Let It Flow: A Formally Verified Compilation Framework for Asynchronous Dataflow

ZHENGYAO LIN, Carnegie Mellon University, USA
YI CAI, University of Maryland, College Park, USA
MILIJANA SURBATOVICH, University of Maryland, College Park, USA

Dataflow architectures have gained renewed interest due to their balance between energy efficiency and performance. In (spatial) dataflow architectures, a program is represented as a set of entirely distributed and dynamically scheduled dataflow operators that communicate through asynchronous channels, which greatly improves data locality and parallelism. However, compiling to dataflow architectures remains an error-prone process, due to the difficulty of maintaining *determinacy* while enabling *pipelining*. Determinacy means that the result of a dataflow program is deterministic and independent of the schedule of operator execution, and pipelining is an important optimization in spatial dataflow that enables parallelism across loop iterations.

In this work, we present Wavelet, the first effort to formally verify a compiler for asynchronous dataflow. We use a mix of techniques to achieve this goal. Our frontend uses a novel capability type system with fences to synchronize conflicting memory accesses and enable *pipelining*. We then verify a Lean formalization of two core compiler passes that translate elaborated programs from the type checker to dataflow graphs, proving important properties of *forward simulation* and *determinacy*. Notably, our formalization semantically propagates the soundness guarantees of the frontend type system, ensuring modularity between simulation and determinacy proofs. In our evaluation, we show that dataflow graphs compiled by Wavelet have comparable quality to those produced by unverified dataflow compilers from RipTide and LLVM CIRCT.

CCS Concepts: • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Architectures**.

Additional Key Words and Phrases: Compiler Verification, Capability Type System, Dataflow Architectures, Coarse-Grained Reconfigurable Arrays, High-Level Synthesis

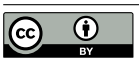
ACM Reference Format:
Zhengyao Lin, Yi Cai, and Milijana Surbatovich. 2026. Let It Flow: A Formally Verified Compilation Framework for Asynchronous Dataflow. *Proc. ACM Program. Lang.* 10, PLDI, Article 185 (June 2026), 25 pages. https://doi.org/10.1145/3808263

1 Introduction

Challenges in decreasing transistor size, memory access bottlenecks, and balancing power and cooling all hinder processor efficiency. Application domains that demand extreme power efficiency, e.g., from ultra-low-power edge computing [15] to high-performance machine learning [42], require hardware designers to look beyond traditional "von Neumann" architecture designs. One attractive alternative is the *spatial dataflow architecture*, which has seen a recent resurgence in interest [1, 11, 14, 16, 20, 42, 45], due to its outstanding balance of power efficiency and performance.

Unlike a traditional von Neumann architecture, where instructions are fetched and executed seemingly according to program order, in spatial dataflow architectures, instructions in a program

Authors' Contact Information: Zhengyao Lin, Carnegie Mellon University, Pittsburgh, USA, zhengyal@cmu.edu; Yi Cai, University of Maryland, College Park, College Park, USA, yicai@umd.edu; Milijana Surbatovich, University of Maryland, College Park, College Park, USA, milijana@umd.edu.



ACM 2475-1421/2026/6-ART185
https://doi.org/10.1145/3808263

are compiled to a set of entirely distributed dataflow *operators* connected with asynchronous channels. At the hardware level, these dataflow operators are mapped to a physical array of "processing elements" (PEs), each serving as an operator (e.g., an arithmetic operation, a memory access operation, or a control-flow operation). These PEs communicate through an on-chip, low-latency network, enabling direct dataflow between operators and significantly reducing data movement and false serialization points compared to traditional von Neumann architectures.

Historically, dataflow computing has seen challenges in maintaining programmability [4, 48, 53], only mapping small fragments of kernels, or being implemented in brittle, fixed-function circuits. Recent work on RipTide [16] has enabled general-purpose programming while retaining efficiency, greatly increasing the viability of dataflow computing. Unfortunately, while the architecture thus supports general-purpose and efficient computing, actually developing the language and compiler infrastructure to realize this potential remains challenging. Since a dataflow program executes as a distributed network of operators, extreme care must be taken to avoid bugs like racing parallel accesses to shared memory, or deadlocks due to a set of operators each waiting on the others to produce the data needed to execute. Indeed, because programming at the low level of dataflow graphs is difficult and unintuitive, in the standard existing workflow, programmers write applications in imperative, *sequential* languages like C, relying on a compiler to translate their traditional sequential program to the distributed, parallel dataflow program. While such a workflow improves programmability, the dataflow compiler is left with the complex task of ensuring memory consistency and equivalence, requiring layers of static analyses to ostensibly convert the original imperative program to an equivalent distributed one, particularly one that remains efficient.

Thus, correct compiler design is an integral, foundational topic in the (re-)emerging area of dataflow architectures. In particular, an ideal compiler for dataflow architectures should produce *determinate* and *pipelined* dataflow graphs: determinacy means that the result of a dataflow graph is deterministic and independent of the schedule of operator execution, precluding the possibility of data races. Meanwhile, pipelining is one of the most important optimizations that utilizes the parallelism in spatial dataflow, where computation of future loop iterations can start before the previous iteration has finished. In essence, pipelining requires removal of unnecessary dependencies between operations, but doing so risks introducing data races and violating determinacy.

We argue that *formal verification* is a crucial step in developing dataflow compilers that generate correct and efficient programs. Rather than relying on more obviously correct but overly conservative compiler behavior, we should design compilers that are mathematically guaranteed to generate dataflow programs that preserve the intended behavior of the sequential application written by the programmer, while not restricting desired parallelism unless needed for correctness.

To that end, we present Wavelet, the first formally verified compiler for asynchronous dataflow. To approach the challenges of determinacy and pipelining, we use a combination of typing and compiler verification techniques. Our core compiler passes that translate from an intermediate sequential language $\mathbb{L}_{let}$ to a dataflow calculus $\mathbb{L}_{flow}$ are formally verified in the Lean 4 theorem prover [40], guaranteeing forward simulation and deadlock-freedom. To enable pipelining, our frontend language is equipped with a novel *capability type system with fences* to ensure consistent usage of shared memory while not over-synchronizing memory operations. The two layers of the compiler work together to achieve determinacy: the type system elaborates memory synchronization with *affine permission tokens*, and our compiler proofs propagate the typing guarantees through simulation proofs, proving the determinacy of the final dataflow program.

Our compiler proofs are designed to be modular and extensible. By interpreting $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$ programs in a common semantic framework, we modularly verify the compilation of a single function and then extend the proof to whole programs using a separate proof for linking dataflow graphs. By formulating the soundness condition of affine permission tokens as label restrictions,

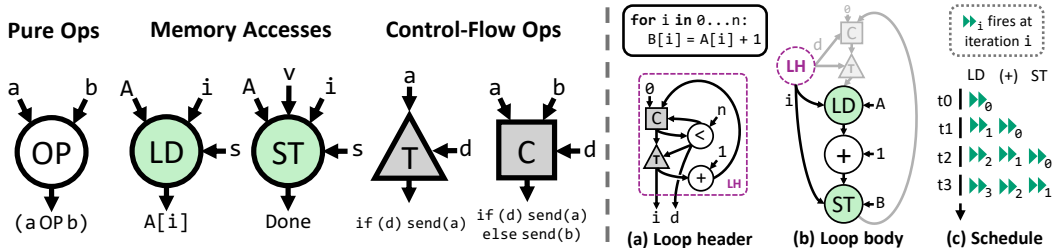


Fig. 1. Overview of key dataflow operators and an example dataflow graph with its pipelined schedule.

we carry the typing guarantees through multiple simulation passes *without modifying them*, and then prove determinacy independently from forward simulations.

In summary, our main contributions are:

- A capability-based type system with fences to ensure race-freedom and facilitate pipelining.
- Formally verified core parallelization passes of a dataflow compiler in Lean, guaranteeing forward simulation and determinacy.
- A novel technique of semantic affine permissions to propagate the frontend typing soundness guarantees throughout simulation and determinacy proofs, greatly improving modularity.

2 Background & Motivation

We explain the basics of compilation for dataflow architectures and the challenges of maintaining correct yet efficient execution, motivating the design of our verified compilation framework.

Dataflow Operators. A dataflow compiler translates code written in a sequential language to a *dataflow program*. These programs are represented as dataflow graphs, where the nodes of the graph are *operators* and the edges are communication *channels* by which data flows between operators. The set of available operators essentially forms the ISA of a particular dataflow architecture. We show representative operators in Fig. 1, based on the RipTide ISA [16]. A pure OP (arithmetic/logic: +, <, etc.) consumes inputs from channels a and b and outputs the operation's result (e.g., $a + b$). Memory operators like load (LD) and store (ST) may modify main memory state. Both take a base address A, an index i, and an optional control signal s. When all inputs arrive, LD outputs the value at $A[i]$; ST stores input v at $A[i]$ and optionally outputs a Done control signal, which is critical for enforcing memory ordering. Because operators fire once inputs are ready, (conflicting) memory operations might execute out-of-order. To prevent this, compilers connect the Done signal of a preceding ST to the signal input s of a subsequent LD or ST. This explicit data dependency prevents the subsequent operation from firing prematurely, transforming implicit sequential memory ordering into explicit dataflow. Finally, control flow is implemented by routing values conditionally. A steer operator (T) acts as a branch filter: given decider d and input a, it forwards a if d is true, and *discards* it otherwise. A carry operator (C) acts as a loop variable: it initially forwards its left input a (the loop's initial value), then on subsequent iterations forwards whichever input decider d selects.

Dataflow Programs. We now step through an example dataflow program (right side of Fig. 1) which increments each element of an array A and stores the result in B. We show the loop header and body subgraphs in separate columns (a) and (b) for clarity. The loop header in column (a) is responsible for generating a stream of values $0, \ldots, n-1$ for the loop variable i. It starts by emitting the initial value 0 from the carry operator, which is then passed to the comparison operator to check the loop condition $i < n$. If this check passes, the steer operator then allows the value of i to flow to both the add operator and the rest of the loop body. The add operator increments i and

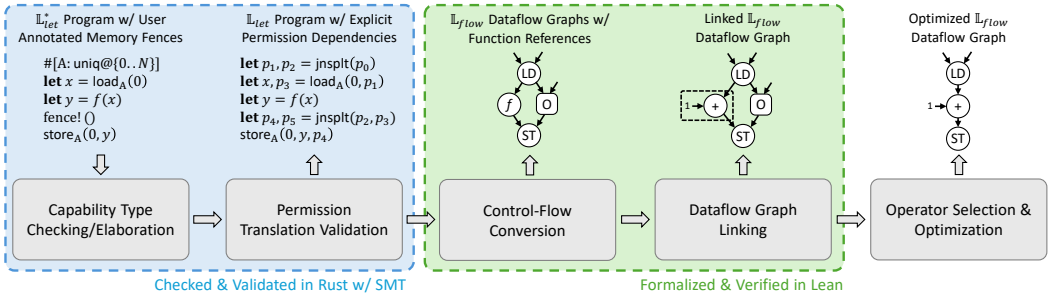


Fig. 2. Compiler pipeline of Wavelet. The compilation transforms the frontend language $\mathbb{L}^*_{let}$ through the elaborated IR $\mathbb{L}_{let}$ to the dataflow calculus $\mathbb{L}_{flow}$.

feeds the new value back to the carry through a back-edge, and the same process repeats until the loop condition fails. Meanwhile, the loop body in column (b) takes a stream of values for i from the loop header, and performs the load of $A[i]$, the increment $A[i]+1$, and the store to $B[i]$ for each i.

Notice that the graph edges in column (a) form a cycle; i has a loop-carried dependence, requiring its updated value to be passed back to the carry operator on each loop iteration. On the other hand, depending on whether A and B are disjoint, the loop body may have no such dependencies, indicating that iterations of the body can be parallelized, as we explain next.

Pipelining and Determinacy. Pipelining is a key optimization in dataflow architectures that enables more parallelism. In Fig. 1(b), the grayed-out subgraph introduces a back-edge from the store to the load. If we do *not* have this back-edge, the dataflow graph is *pipeline-able*, i.e., the load in iteration $i+1$ does not have to wait for the store in iteration i to finish. More specifically, consider the schedule of loop body operators in column (c). The horizontal axis represents the three operators in the loop body (LD, +, and ST), and the vertical axis represents abstract time steps. Assume that the loop header can stream one value of i out per time step, with the load of iteration 0 occurring at time t_0. At time t_1, the load of iteration 1 can run in parallel with the add of iteration 0; by time t_3, the three operators in the loop body are simultaneously carrying out computation from three different iterations. At the hardware level, this ideal situation is complicated by the actual timing of each operator, but at the compiler level, we want to maximize such abstract pipelining by not adding unnecessary data dependencies such as the gray part in column (b). A pipelined dataflow graph carries a higher risk of losing *determinacy*, however. For example, the loop body is reading from A and writing to B. If A and B might alias, then the schedule in column (c) could cause data races and non-deterministic results (e.g., if the store of $B[0]$ and load of $A[2]$ at time t_2 are accessing the same location). In this case, the gray back-edge becomes necessary to avoid data races and preserve the original memory dependency in the source sequential program. Therefore, a key challenge in Wavelet is to enable pipelining by eliminating unnecessary memory dependencies through a type system, while preserving correctness and determinacy with formalized and mechanized proofs.

3 System Overview

We present an overview of Wavelet's compilation pipeline in Fig. 2. At a high level, Wavelet takes in a sequential program on the left side, decorated with *capability types* (`#[A: uniq@{0..N}]`) for accessing arrays and user-inserted *fences* (`fence!()`) indicating synchronization points, and translates it into a lower-level dataflow program on the right side that is guaranteed to be data-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 graph; and (5) operator selection and optimization, to lower the graph to an optimized, architecture-specific representation. The soundness of the frontend passes (1) and (2) is certified with translation validation implemented in Rust, while the core control flow conversion and linking passes (3) and (4) are formally verified in Lean. The last operator selection/optimization pass is currently unverified, but can draw on orthogonal work in verifying graph rewrite passes [18].

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 (§4.1), and (2) an elaboration pass that lowers fences into explicit permission tokens that the backend uses to provably preserve the typing guarantee (§4.2).

The main idea of the capability types is that the user can type an array region R as uniq for exclusive write access, or shrd for shared read access. 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.

Since fences are not directly supported by the backend dataflow architecture, which enforces ordering through explicit control signals, 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 $\mathbb{L}_{let}$ (§4.2), 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 (§4.3).

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 Fig. 2) to compile an $\mathbb{L}_{let}$ program to the dataflow calculus $\mathbb{L}_{flow}$.

The first is *control flow conversion* (§5.2), 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 Fig. 1). While this pass is standard in dataflow compilers dating back to TTDA [4], 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 program. Unlike 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.

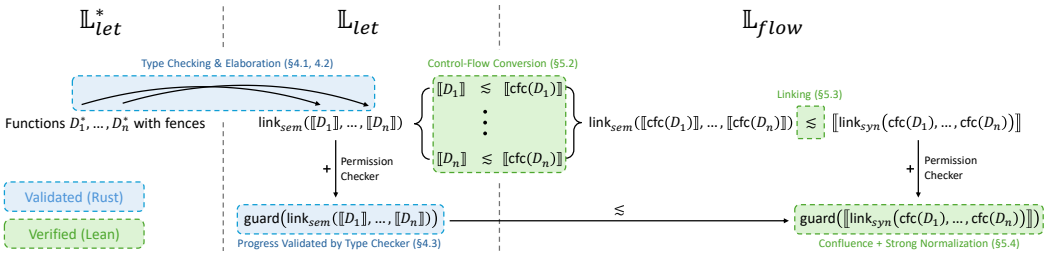


Fig. 3. Proof structure overview of Wavelet. $\lesssim$ denotes simulation; cfc is control flow conversion (§5.2); link_{sem} is the semantic linking of LTSs (§5.3); link_{syn} is the syntactic linking of dataflow graphs (§5.3); guard imposes permission checking on the base semantics (§5.4).

Additionally, for a dataflow graph 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* (§5.3), which connects individually compiled dataflow graphs 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 graphs distribute local variables across channel buffers that may hold values from multiple concurrent function invocations. To tackle these challenges, we place static restrictions (§4.3) on $\mathbb{L}^*_{let}$ to facilitate linking, and use a modular linking semantics inspired by interaction semantics [47].

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 graph has at least one deadlock-free schedule with the same behavior as the input sequential program; and (b) *determinacy*: given a terminating $\mathbb{L}_{let}$ 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.

Fig. 3 gives an overview of the proof structure of our compiler, showing how different guarantees of each component connect to establish our final correctness results. In particular, we design 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 graphs. 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 graph in $\mathbb{L}_{flow}$ that simulates the input $\mathbb{L}_{let}$ program and is determinate. To interface with actual backend hardware (e.g., CGRAs [35] or FPGAs), we lower certain pseudo-operators to architecture-specific operators (instruction selection), and perform unverified optimizations to improve the quality of the dataflow graph. We define these transformations as term rewrites in $\mathbb{L}_{flow}$ (§5.5). Combined with the memory ordering strategy in the type checker/elaborator and the core control-flow passes of the compiler, we show in §6 that we can produce dataflow graphs with comparable quality to existing unverified dataflow compilers in RipTide [16] and LLVM CIRCT [36].

4 Frontend: Capability Types and Fences

In this section, we present the design of Wavelet's frontend, which consists of a capability type system for our source language $\mathbb{L}^*_{let}$ (§4.1), an elaboration pass to an intermediate language $\mathbb{L}_{let}$ (§4.2) with a translation validation phase to ensure that the produced $\mathbb{L}_{let}$ program satisfies the

Op	op	::=	$\mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{ld} \mid \mathbf{st} \mid \cdots$	*Value*	v	::=	$n \mid \mathbf{true} \mid \mathbf{false} \mid ()$
Expr	E	::=	$\mathbf{ret}(\vec{x}) \mid \mathbf{tail}(\vec{x}) \mid \mathbf{let}\ \vec{y} = \mathrm{op}(\vec{x});\ E$	*Type*	τ	::=	$\mathbf{bool} \mid \mathbf{unit} \mid [\tau;\ \mathtt{N}]$
			$\mid \mathbf{let}\ \vec{y} = f(\vec{x});\ E \mid \mathbf{let}\ y = v;\ E$				$\mid \mathbf{u8} \mid \mathbf{u16} \mid \mathbf{u32} \mid \cdots$
			$\mid \mathbf{if}\ x\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \mid -E$	*Cap.*	C	::=	$\mathrm{uniq@}R \mid \mathrm{shrd@}R \mid C_1 \cdot C_2$
Defn	D	::=	$\mathbf{def}\ f(\overrightarrow{x:\tau})\langle\ \overrightarrow{A:C}\ \rangle \to \vec{\tau_o} = E$	*R-expr*	r	::=	$n \mid x \mid r + r \mid r - r \mid r * r$
Program	P	::=	$\vec{D}$	*Region*	R	::=	$r..r$

Fig. 4. Syntax of $\mathbb{L}^*_{let}$.

soundness property assumed by later verified compiler passes (§4.3). While both $\mathbb{L}^*_{let}$ and $\mathbb{L}_{let}$ are sequential languages and do not need memory synchronization by themselves, the goal of the type system and elaboration is to soundly make *implicit* memory dependencies *explicit*, so that we can guarantee the determinacy of the dataflow graph produced by the core verified passes.

4.1 $\mathbb{L}^*_{let}$ with Capability Types and Fences

This section defines $\mathbb{L}^*_{let}$, which is a first-order sequential language with capability types and a construct called a *fence* to explicitly annotate memory dependencies.

Syntax. Fig. 4 shows the syntax of $\mathbb{L}^*_{let}$. A top-level $\mathbb{L}^*_{let}$ program P is made up of a list of function definitions, and the body of each function is an expression consisting of let bindings, conditional expressions, returns, and tail calls. A key feature of $\mathbb{L}^*_{let}$ is that every expression can be optionally preceded by a fence ($-$), intuitively denoting memory dependencies from operations after the fence to operations before it. To help the type system infer suitable capabilities, each $\mathbb{L}^*_{let}$ function is annotated with two lists of typed parameters: (1) ordinary variables $\overrightarrow{x:\tau}$, and (2) capability-annotated array inputs $\overrightarrow{A:C}$ (assumed to be disjoint from $\vec{x}$). A capability type C describes allowed access permissions to a precise *region* of the annotated array. Syntactically, this permission can be shared $\mathrm{shrd@}\{R\}$ or unique $\mathrm{uniq@}\{R\}$, where R is a range $[r_1..r_2)$ of indices.

$$\textsc{Load}\quad \frac{\Gamma(i) = \mathbf{int} \quad \Phi \vDash \mathrm{shrd@}\{i\} \le \Delta[A] \qquad \Pi;\Gamma[y \mapsto \mathbf{int}];\Delta[A \mapsto \Delta[A] \setminus \mathrm{shrd@}\{i\}] \vdash_{f,\Phi} E : \tau}{\Pi;\Gamma;\Delta \vdash_{f,\Phi} \mathbf{let}\ y = \mathbf{ld}(A,i);\ E : \tau}$$

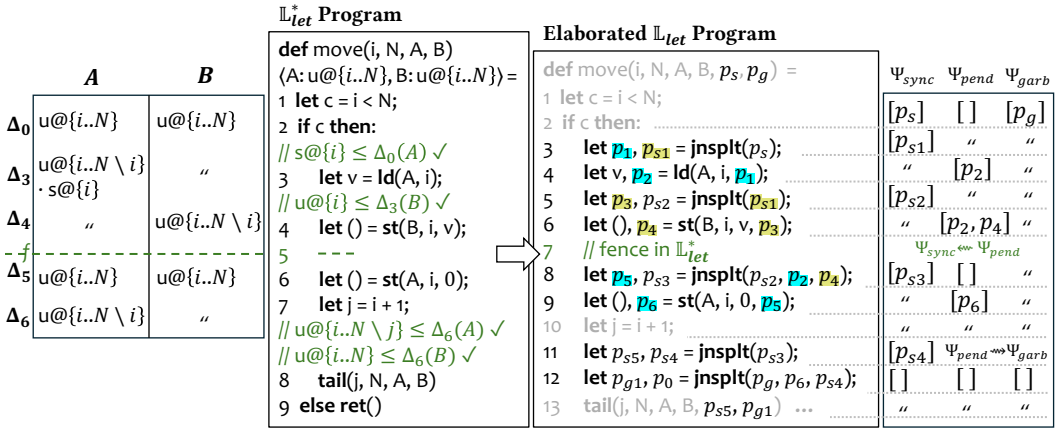
$$\textsc{Fence}\quad \frac{\Pi(f) = \mathbf{def}\ f(\overrightarrow{x_i:\tau_i})\langle\ \overrightarrow{A:C}\ \rangle \to \vec{\tau_o} \qquad \Pi;\Gamma;\overrightarrow{A:C} \vdash_{f,\Phi} E : \tau}{\Pi;\Gamma;\Delta \vdash_{f,\Phi} -E : \tau}$$

$$\textsc{Store}\quad \frac{\Gamma(i) = \mathbf{int} \quad \Gamma \vdash v : \mathbf{int} \quad \Phi \vDash \mathrm{uniq@}\{i\} \le \Delta[A] \qquad \Pi;\Gamma;\Delta[A \mapsto \Delta[A] \setminus \mathrm{uniq@}\{i\}] \vdash_{f,\Phi} E : \tau}{\Pi;\Gamma;\Delta \vdash_{f,\Phi} \mathbf{let}\ () = \mathbf{st}(A,i,v);\ E : \tau}$$

$$\textsc{Tail-Call}\quad \frac{\Pi(f) = \mathbf{def}\ f(\overrightarrow{x_i:\tau_i})\langle\ \overrightarrow{A:C}\ \rangle \to \vec{\tau_o} \qquad \Gamma(\vec{i}) = \vec{\tau_i} \quad \Phi \vDash \forall a \in \vec{A}.\, C[\vec{i}/\vec{x_i}] \le \Delta[a]}{\Pi;\Gamma;\Delta \vdash_{f,\Phi} \mathbf{tail}(\vec{i}) : \vec{\tau_o}}$$

Fig. 5. Selected typing rules for $\mathbb{L}^*_{let}$.

Capability Type System. We equip $\mathbb{L}^*_{let}$ with a capability type system to ensure that user-provided capability annotations correctly describe the program's actual memory access patterns and that the fences in the program soundly overapproximate all memory dependencies. Information inferred from this type system facilitates the translation of $\mathbb{L}^*_{let}$ to $\mathbb{L}_{let}$ (§4.2), which has a more explicit representation of capabilities and is more compatible with later compilation passes. There are two key challenges that our type system must address to be precise enough for the purpose of pipelining (§2): (1) it needs to distinguish between read and write accesses since reads can be done safely in parallel, but writes (to the same location) need to be synchronized; and (2) it needs to distinguish



(a) $\mathbb{L}^*_{let}$ example with its typing contexts **(b) Elaboration and permission contexts**

Fig. 6. An example program `move` in $\mathbb{L}^*_{let}$, along with its typing contexts, elaborated $\mathbb{L}_{let}$ program, and the permission contexts during elaboration. Permissions for arrays A and B are highlighted. u@{...} and s@{...} are abbreviations for uniq@{...} and shrd@{...}, respectively.

accesses to different regions of an array to avoid over-synchronization. As a result, each capability C has two components: a shrd/uniq component for distinguishing read/write accesses, and a region R component for specifying the precise indices.

We now sketch the design of our type system, where the main typing judgment has the form $\Pi; \Gamma; \Delta \vdash_{f,\Phi} E : \tau$. Here, f denotes the current function, Π maps function names to their signatures, Γ is the typing context for ordinary variables, E is the expression being checked, and τ is its final type. Additionally, two important contexts that we need to include in the type judgment are the capability context Δ, which is a map from array names to capabilities C, and the propositional context Φ, which is a collection of logical facts from branches or let bindings that overapproximate the state of ordinary variables. In particular, both Δ and Φ can mention ordinary variables in Γ.

Fig. 5 presents selected key rules, and the full formulation can be found in our appendix [33]. To explain the intuition behind these rules, we use the concrete example in Fig. 6 (a), which is an $\mathbb{L}^*_{let}$ function that copies the contents of array A to B, and then clears A to zero. When loading from or storing to an array, we need to check that the current capability context Δ contains suitable permissions for accessing the target array (shrd for load and uniq for store) at the specific index being accessed. These checks are described by the Load and Store rules in Fig. 5. This is not a purely syntactic check since our capabilities are dependent on ordinary variables (e.g., to track indices), and as a result, premises such as $\Phi \vDash \text{shrd@}\{i\} \le \Delta[A]$ involve SMT queries to check whether we have suitable capabilities under the current propositional context Φ (where we order shrd $\le$ uniq and the ranges are ordered by set inclusion). These SMT queries are represented in green comments in our example in Fig. 6 (a). After the load (resp. store), the suitable permission for the read (resp. written) index is consumed and removed from the capability context Δ when type checking the rest of the expression. One subtlety in removing capabilities is that, to allow parallel loads, we have shrd $\setminus$ shrd = shrd and uniq $\setminus$ shrd = shrd. In our example, consider the load on line 3. Before the load, we have $\Delta_0[A] = \text{uniq@}\{i..N\}$, and after the load, we have $\Delta_3[A] = \text{uniq@}\{i..N \setminus i\} \cdot \text{shrd@}\{i\}$, where $\cdot$ denotes the union of capabilities.

In order to write to i again on line 6, the $\mathbb{L}^*_{let}$ program must insert a fence to indicate a memory dependency. A fenced expression $-E$ in $\mathbb{L}^*_{let}$ intuitively "resets" the capability context to the initial

function context, as formally described by rule Fence. In our example, the fence on line 5 resets the next capability context to $\Delta_5 = \Delta_0$ before type checking the rest.

Finally, when calling another function or doing a tail call, the Tail-Call rule is used to check that suitable capabilities required by the callee function are present in the current capability context.

4.2 Ghost Permissions Inference and Fence-Removal

The language $\mathbb{L}^*_{let}$ only requires light annotations to soundly overapproximate memory dependencies in the program, but dataflow architectures do not have a fence primitive. Instead, each LD and ST operator takes an additional argument (a control signal) that indicates when the operation can safely proceed (§2). Therefore, to bridge this gap, Wavelet lowers $\mathbb{L}^*_{let}$ to an intermediate language called $\mathbb{L}_{let}$ with more explicit forms of memory dependencies.

We demonstrate the lowering using the same example in Fig. 6 (b), which is the lowered $\mathbb{L}_{let}$ program from Fig. 6 (a). There are several differences in this intermediate representation (highlighted in the figure). First, $\mathbb{L}_{let}$ removes all capability types and fences, but extends $\mathbb{L}^*_{let}$ with two "ghost" constructs: ghost variables representing affine permission tokens and the ghost operator jnsplt, which is used to **join** input permission tokens and then **split** them in a suitable way. Semantically, **let** p', $p'' = \mathsf{jnsplt}(p_1, \ldots, p_n)$ combines n permission variables and splits the result into two fresh permissions, representing the needed permission for the operation immediately after this jnsplt and the remaining permissions for subsequent operations. Another main difference is that operators and functions now accept and return explicit permission variables. For example, memory operations now consume and produce one additional permission variable, and functions accept two additional permission arguments p_{sync} and p_{garb} (abbreviated as p_s and p_g, respectively, in Fig. 6 (b)).

While the detailed lowering definition can be found in our appendix [33], we highlight some key aspects. To guide the lowering process, we maintain a permission context $\Psi = (\Psi_{sync}, \Psi_{pend}, \Psi_{garb})$. Ψ_{sync} is initialized with p_{sync}, which holds synchronized permissions available for immediate use—they are inherited from capability types, and are therefore *guaranteed sufficient* up to the next fence point (or the end of the function if there is no fence). Ψ_{garb} is initialized with p_{garb}, which is needed to "garbage collect" permissions that are no longer needed and can be returned to the caller, enabling permission reuse in subsequent operations. Finally, Ψ_{pend} tracks permissions from issued operations awaiting synchronization (moving to Ψ_{sync} at fences) or garbage collection (moving to Ψ_{garb} at tail calls and returns). Together, these contexts let the lowering soundly materialize the capabilities and fences from $\mathbb{L}^*_{let}$ while avoiding unnecessary memory dependencies in the resulting $\mathbb{L}_{let}$ program, which is crucial for safe pipelining in the final $\mathbb{L}_{flow}$ program.

To see how these contexts are used and how synthesized permissions reflect the original typing constraints, consider lines 3-9 in Fig. 6 (b). When translating a ld/st, we synthesize a jnsplt that splits Ψ_{sync} into a permission for the ld/st and remaining permissions. The returned permission from the ld/st moves into Ψ_{pend}, and the lowering process continues with the remaining permissions in Ψ_{sync} until the next fence, which shifts all pending permissions back into Ψ_{sync}. Operationally, these permission variables enforce sound memory synchronization. For example, the ld on line 4 and the st on line 6 depend on *disjoint* permissions and can be executed in parallel, while the st on line 9 needs to wait for the ld on line 4 to produce the permission p_2. Function calls are similar to ld/st except that we synthesize *two* jnsplts for p_{sync} and p_{garb}, respectively (lines 11-12).

4.3 Backend Assumptions

The capability type checking (§4.1) and fence-removal (§4.2) passes of our frontend ensure that implicit memory dependencies (implied by the control flow in $\mathbb{L}^*_{let}$) are explicitly converted to data dependencies on ghost permission variables. Since these passes are currently not formally

verified in Lean, we discuss two important assumptions that our verified backend (§5) places on the frontend to ensure compilation correctness, as well as how they are currently validated.

Soundness of Ghost Permissions. While the ghost permission lowering pass intuitively materializes the memory access guarantees established by $\mathbb{L}^*_{let}$'s type system (§4.1), how do we know that the synthesized permissions indeed preclude any conflicting memory accesses once further lowered to $\mathbb{L}_{flow}$? For example, a buggy lowering pass might ignore capabilities entirely and synthesize a jnsplt that produces more permissions than it consumes. This would allow parallel writes to the same memory location, introducing data races in the final $\mathbb{L}_{flow}$ program. To address this, our frontend uses *translation validation* to certify the soundness of the inferred permission tokens, which our verified backend ultimately relies on to ensure determinacy (§5.4). Note that the translation validator itself is not formally verified, and it serves as an additional safeguard for the frontend.

To formalize the concept of permissions, we rely on *partial commutative monoids* (PCMs) [22], which provide a mathematical framework for reasoning about resource sharing and disjointness. For our frontend, we use the PCM instance of *fractional permissions* [8]: resources are associated with a fraction $f \in [0, 1]$, where $f = 1$ means exclusive access, and any fraction $0 < f < 1$ represents shared access. We define dependent permission specifications for each operator and function in $\mathbb{L}_{let}$, each of which includes a precondition pre (mapping input values to required permissions) and a postcondition post (mapping input and output values to returned permissions). For example, the store operator st has $\text{pre}_{\text{st}}(A, i) = 1@\text{A}\{\text{i}\}$ and $\text{post}_{\text{st}}(A, i, v) = 1@\text{A}\{\text{i}\}$ (i.e., exclusive access to $A[i]$). For jnsplt, we specify that the PCM sum of the input and output permissions must be equal, ensuring that it does not create or destroy permissions.

The *guarded semantics* later defined in §5.4 dynamically checks that the permission specification above is satisfied and gets stuck otherwise. Hence, the exact soundness property validated in this pass (assumed later in Theorem 5.7) is the progress of the $\mathbb{L}_{let}$ program under the guarded semantics.

Static Restrictions on $\mathbb{L}^*_{let}$. The dataflow model that we target in this work uses FIFO queues as channels and requires a static dataflow graph topology. These architectural constraints complicate the compilation of function calls: when a function has multiple (parallel) callers, the compiler must merge their input arguments and route return values to the correct caller. A static topology requires these routing decisions to be fixed ahead of time, while FIFO channels make arbitrary interleavings of parallel calls difficult because operators must process their inputs in order.

While there are multiple (unverified) solutions to this challenge (§7), we adopt an approach similar to RipTide [16], where the control flow of $\mathbb{L}_{let}$ is assumed to satisfy two static restrictions: (1) a function can be statically called at most once outside its definition, but multiple tail-recursive calls within the function definition are allowed; and (2) no mutual recursion is allowed. These restrictions are validated by the backend when parsing the input $\mathbb{L}_{let}$ program from the frontend.

Requirement (1) allows a function to tail-call itself multiple times, such as $\textbf{def}\ f(x) = \textbf{if}\ x \leq 1\ \textbf{then ret}(x)\ \textbf{else}\ (\textbf{if}\ x\%2 = 0\ \textbf{then tail}(x/2)\ \textbf{else tail}(3x+1))$. However, *outside* its own definition, there can be at most one static call site to f. Requirement (2) forbids mutual recursion. In most dataflow architectures, FIFO channels have bounded buffers, and they block the sender if they are full (i.e., backpressure). As a result, non-tail mutual recursion such as $\textbf{def}\ f(x) = g(x)+1; \textbf{def}\ g(x) = f(x) + 1$ causes a *deadlock*, because it induces an unbounded "stack."

These restrictions do not limit the expressiveness of $\mathbb{L}_{let}$ and $\mathbb{L}^*_{let}$, since any structured control flow (i.e., with branching and loops) can be transformed to satisfy these restrictions: branching is built-in, and loops can be transformed to tail-recursive calls.

They do, however, reduce the ergonomics of writing $\mathbb{L}_{let}$ directly, because users must currently encode control flow in this normalized form. In future work, we plan to automate this process with desugaring/normalization passes, or to use $\mathbb{L}_{let}$ as an IR for dataflow compilation.

$$((\textbf{def}\ f(\vec{y}) = E), \mathsf{init}, \sigma) \xrightarrow{\mathsf{input}(\vec{v})} ((\textbf{def}\ f(\vec{y}) = E), E, \{\vec{y} \mapsto \vec{v}\})$$

$$(D, \textbf{ret}(\vec{x}), \sigma) \xrightarrow{\mathsf{output}(\sigma(\vec{x}))} (D, \mathsf{init}, \emptyset)$$

$$((\textbf{def}\ f(\vec{y}) = E), \textbf{tail}(\vec{x}), \sigma) \xrightarrow{\tau} ((\textbf{def}\ f(\vec{y}) = E), E, \{\vec{y} \mapsto \sigma(\vec{x})\})$$

$$(D, (\textbf{let}\ \vec{y} = \mathsf{op}(\vec{x});\ E), \sigma) \xrightarrow{\mathsf{yield}(\mathsf{op}, \sigma(\vec{x}), \vec{v})} (D, E, \sigma[\vec{y} \mapsto \vec{v}])$$

$$(D, \textbf{if}\ x\ \textbf{then}\ E_{\textbf{true}}\ \textbf{else}\ E_{\textbf{false}}, \sigma) \xrightarrow{\tau} (D, E_{\sigma(x)}, \sigma) \quad \text{if } \sigma(x) \in \{\textbf{false}, \textbf{true}\}$$

Fig. 7. Small-step operational semantics of an $\mathbb{L}_{let}$ definition D, denoted $[\![D]\!] := ((D, \mathsf{init}, \emptyset), \rightarrow) \in \mathsf{LTS}^{Label_O}_{C^{let}}$, with the set of configurations $C^{let} := \mathit{Defn} \times (\mathit{Expr} \sqcup \{\mathsf{init}\}) \times (\mathit{Variable} \rightharpoonup \mathit{Value})$. For a list of variables $\vec{x} = x_1, \ldots, x_n$, we use $\sigma(\vec{x})$ to denote the list of values $\sigma(x_1), \ldots, \sigma(x_n)$, assuming they are all defined.

$$\begin{array}{lll} \mathit{Proc} \quad \pi & ::= \mathsf{op}(\vec{c}) \rightarrow \vec{d} & \mathsf{op} \in O \\ & \mid \mathsf{merge}_s(c_1, \vec{c_2}, \vec{c_3}) \rightarrow \vec{d} & s \in \{\mathsf{init}, \mathsf{left}, \mathsf{right}\} \\ & \mid \mathsf{steer}_f(c_1, \vec{c_2}) \rightarrow \vec{d} & f \in \{\bot, \top\} \\ & \mid \mathsf{fork}(c) \rightarrow \vec{d} \mid \mathsf{order}(\vec{c}) \rightarrow d \mid \mathsf{sink}(\vec{c}) & \\ & \mid \mathsf{forward}(\vec{c}) \rightarrow \vec{d} \mid \mathsf{const}_v(c) \rightarrow \vec{d} & \\ & \mid \pi_1 \mid\mid \pi_2 \mid \nu c.\ \pi & \end{array}$$

$$(\pi, \sigma) \xrightarrow{\mathsf{input}(\vec{v})} (\pi, \mathsf{push}(\overrightarrow{c_{in}}, \vec{v}, \sigma)) \qquad \frac{\mathsf{pop}(\overrightarrow{c_{out}}, \sigma) = (\vec{v}, \sigma')}{(\pi, \sigma) \xrightarrow{\mathsf{output}(\vec{v})} (\pi, \sigma')} \qquad \frac{(\pi_1, \sigma) \xrightarrow{l} (\pi_1', \sigma')}{(\pi_1 \mid\mid \pi_2, \sigma) \xrightarrow{l} (\pi_1' \mid\mid \pi_2, \sigma')}$$

$$\frac{\mathsf{pop}(\vec{c}, \sigma) = (\overrightarrow{v_{in}}, \sigma')}{(\mathsf{op}(\vec{c}) \rightarrow \vec{d}, \sigma) \xrightarrow{\mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})} (\mathsf{op}(\vec{c}) \rightarrow \vec{d}, \mathsf{push}(\vec{d}, \overrightarrow{v_{out}}, \sigma'))}\ \textsc{Sync-Op}$$

Fig. 8. Syntax and selected rules of the semantics of $\mathbb{L}_{flow}$ processes. We denote the semantics of a process π as $[\![\pi]\!]_{\overrightarrow{c_{in}}, \overrightarrow{c_{out}}} := ((\pi, \emptyset), \rightarrow) \in \mathsf{LTS}^{Label_O}_{C^{flow}}$, with the set of configurations $C^{flow} := \mathit{Proc} \times (\mathit{Channel\ Name} \rightharpoonup \mathit{Value}^*)$. Here, $\overrightarrow{c_{in}}$ and $\overrightarrow{c_{out}}$ are distinguished lists of input and output channels. push and pop are auxiliary functions for pushing and popping values from channel buffers.

5 Compiler Design and Verification

In this section, we detail the design and formal verification of the core compilation passes of Wavelet, covering the verified portions of our proof structure overview in Fig. 3.

In §5.1, we introduce our dataflow calculus $\mathbb{L}_{flow}$ and the semantics of $\mathbb{L}_{let}$. In §5.2 and §5.3, we formally define the two core passes of control flow conversion (cfc) and linking ($\mathsf{link}_{sem}/\mathsf{link}_{syn}$), and sketch their forward simulation proofs. In §5.4, we define the guarded semantics (guard) and prove the determinacy of the final dataflow graph. Finally, in §5.5, we briefly cover our optimization passes as graph rewrites on $\mathbb{L}_{flow}$ processes.

5.1 Dataflow Calculus $\mathbb{L}_{flow}$ and the Semantics of $\mathbb{L}_{let}$

We first introduce a simple calculus $\mathbb{L}_{flow}$ as our target language. Its syntax can be found in Fig. 8. In essence, an $\mathbb{L}_{flow}$ process is a parallel composition of *atomic processes*, where each atomic process is a recursive process that waits for inputs, performs computation, and produces some outputs.

We classify atomic processes into two categories: *asynchronous* and *synchronous* operators. An operator is asynchronous if its input/output behavior may depend on its input values and local state (e.g., steer in §2), whereas a synchronous operator reads and then produces exactly one value from/to each input/output channel (e.g., arithmetic or memory operators).

Asynchronous operators such as steer and merge are built-in, because our core passes depend on their semantics to convert control flow and perform optimizations, and we will discuss their semantics shortly. Besides asynchronous operators, the syntax and semantics of $\mathbb{L}_{flow}$ are parametric in a customizable set O of synchronous operators (previously specialized to Op in Fig. 4). These operators do not need special treatment in compilation due to their more uniform behavior. This separation of built-in asynchronous operators and custom synchronous operators allows Wavelet to easily target dataflow architectures with different instruction sets. Further, the similar interface of a synchronous operator and a function in $\mathbb{L}_{let}$ also allows Wavelet to treat function calls modularly as operator calls and perform linking in a separate pass.

Finally, atomic processes in $\mathbb{L}_{flow}$ can be used in parallel composition $\pi_1 \mid\mid \pi_2$ and channel restriction $\nu c.\ \pi$, with the usual structural congruence rules of commutativity and associativity of $\mid\mid$, scope extrusion, and alpha renaming.

Semantics of $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$. We use labeled transition systems (LTSs) to model the semantics of both $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$. As notation, we use LTS_C^L as a shorthand for the set of relations $R \subseteq C \times L \times C$, with states C and labels L, and a distinguished initial state $c_0 \in C$. For any $(c_0, \rightarrow) \in \mathsf{LTS}_C^L$, we use $c \xrightarrow{ls} c'$ to denote zero or more steps from c to c' with some string ls of labels.

The small-step operational semantics of $\mathbb{L}_{let}$ definitions and $\mathbb{L}_{flow}$ processes are shown in Fig. 7 and Fig. 8. Both LTSs share four types of labels:

$$Label_O \ni l ::= \tau \mid \mathsf{input}(\vec{v}) \mid \mathsf{output}(\vec{v}) \mid \mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}}) \quad \mathsf{op} \in O$$

The silent label τ denotes an internal step without external effects; an $\mathsf{input}(\vec{v})$ label provides initial input $\vec{v}$ to the program/process; an $\mathsf{output}(\vec{v})$ label denotes final output $\vec{v}$ produced by the program/process; and finally, a $\mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})$ label denotes calling either an operator or function with inputs $\overrightarrow{v_{in}}$, and transitioning to a state assuming the outputs to be $\overrightarrow{v_{out}}$.

The transitions of $\mathbb{L}_{let}$ are mostly straightforward. Initially, a function accepts an input label and initializes the parameters. A return expression produces an output label with the return values. A tail call produces a silent label and re-initializes the current function with tail call arguments. Note, however, that function and operator calls are left uninterpreted in the $\mathbb{L}_{let}$ semantics: they simply generate yield labels and non-deterministically transition to a state assuming the return values of the call. This is intended to make our proofs more modular, as we can separately prove forward simulations of individual function compilation (§5.2) and linking (§5.3). This is also similar to uninterpreted events in interaction trees [52].

On the $\mathbb{L}_{flow}$ side (Fig. 8), we show selected rules for inputs/outputs, parallel composition, and synchronous operators, and describe the semantics of built-in operators in the text below. The $\mathsf{steer}_f(c_1, \vec{c_2}) \rightarrow \vec{d}$ operator reads a Boolean value from the "decider" channel c_1 and input values from $\vec{c_2}$. If the decider equals the filter f, the input values are forwarded to $\vec{d}$; otherwise, the input values are discarded. The $\mathsf{merge}_s(c_1, \vec{c_2}, \vec{c_3}) \rightarrow \vec{d}$ operator has three states. At the initial state $s = \mathsf{init}$, it reads the decider value from c_1. If the decider is true, it transitions to left, and otherwise it transitions to right. Then, at state left (resp. right), it forwards the input from $\vec{c_2}$ (resp. $\vec{c_3}$) to $\vec{d}$, and then transitions back to init. The carry operator in Fig. 1 for representing loop variables is exactly $\mathsf{merge}_{\mathsf{left}}$. The fork operator duplicates its input to multiple outputs. The order operator is a low-level synchronization primitive that waits for all its inputs to be ready and then forwards the first input. The sink operator consumes its input without any output. The forward operator simply

forwards its input to its output. Finally, $\mathsf{const}_v(c)$ pushes the constant v to the output whenever it receives any value from the input c.

To conclude our semantics setup, we define a notion of simulation $\lesssim$ for LTSs with label set $Label_O$. Intuitively, this says that if the LTS $(c_0, \to)$ can perform one step, then a related state in $(c_0', \to')$ can also make multiple steps to reach another related state, with some restrictions such as no silent steps before an input label. This definition is stricter than the usual notion of weak simulation [43], due to technical reasons in the simulation proofs of linking.

Definition 5.1 (IO-Restricted Weak Simulation). Given two transition systems $(c_0, \to) \in \mathsf{LTS}_C^{Label_O}$ and $(c_0', \to') \in \mathsf{LTS}_{C'}^{Label_O}$, a relation $R \subseteq C \times C'$ is an *IO-restricted weak simulation* if $(c_0, c_0') \in R$; and for every $(c, c') \in R$ and $l \in Label_O$ with $c \xrightarrow{l} c_1$, there exists c_1' with $(c_1, c_1') \in R$ such that: (1) if $l = \tau$, then $c' \xrightarrow{\tau^*} c_1'$; (2) if $l = \mathsf{input}(\vec{v})$, then $c' \xrightarrow{l\tau^*} c_1'$; (3) if $l = \mathsf{output}(\vec{v})$, then $c' \xrightarrow{\tau^* l} c_1'$; (4) if $l = \mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})$, then $c' \xrightarrow{l} c_1'$. If there exists such a relation R, we say that $(c_0, \to) \lesssim (c_0', \to')$.

5.2 Control Flow Conversion

In this section, we describe the core control flow conversion (CFC) pass of Wavelet, which, in essence, eliminates all control flow in the input $\mathbb{L}_{let}$ function, and produces an $\mathbb{L}_{flow}$ process with only dataflow. We first define the compilation of each $\mathbb{L}_{let}$ construct, and then sketch a proof of its forward simulation. Throughout, we assume that we are compiling a single $\mathbb{L}_{let}$ function, parametric in the operator set O (which can contain base operators or other $\mathbb{L}_{let}$ function symbols).

Compiling a Function Definition. The compiled $\mathbb{L}_{flow}$ process of an $\mathbb{L}_{let}$ definition D, denoted $\mathsf{cfc}(D, \vec{c}, \vec{d})$, is defined below, where $\vec{c}$ (resp. $\vec{d}$) are free channels in $\mathsf{cfc}(D, \vec{c}, \vec{d})$ to represent input (resp. output) channels of the entire process, and bound names in ν are distinct and fresh.

$$
\begin{aligned}
\mathsf{cfc}(\mathbf{def}\ f(\vec{x}) = E, \vec{c}, \vec{d}) := \nu\ \vec{c}', \vec{d}', t, \vec{r}, \vec{r}'.\ & \mathsf{merge}_{\mathsf{left}}(t, \vec{c}, \vec{r}) \to \vec{c}' \ \|\ && \text{(Merge inputs/tail call)} \\
& \mathsf{cfc}(E, \{\vec{x} \mapsto \vec{c}'\}, t, \vec{d}', \vec{r}') \ \|\ && \text{(Compile body)} \\
& \mathsf{steer}_{\bot}(t, \vec{d}') \to \vec{d} \ \|\ \mathsf{steer}_{\top}(t, \vec{r}') \to \vec{r} && \text{(Route outputs)}
\end{aligned}
$$

The definition consists of three parts, where the first and last lines are for initialization and cleanup. The compilation of the function body E in the middle is denoted $\mathsf{cfc}(E, \Sigma, t, \vec{d}, \vec{r})$, where Σ is a map from unused $\mathbb{L}_{let}$ variables to channel names; t is a channel name used for sending a Boolean flag indicating whether the expression ends with a return or a tail call, which we will call the *tail condition* below; $\vec{d}$ are channels used for sending return values; and $\vec{r}$ are channels used for sending tail call arguments.

The trickiness of compilation comes from determining the correct input and output channel routing, which depends on the different ways a function is called and terminates. Recall that an $\mathbb{L}_{let}$ function can receive (non-recursive) external calls or tail-recursive calls within the function body (a design choice due to the difficulty with dataflow function calls discussed in §3). To produce a dataflow graph that handles both cases, we need a merge operator at the beginning to combine call arguments from both situations. The initial state of the merge is set to left, so that initially, the function will directly forward the input arguments to the function body, without requiring a decider value. In turn, to handle outputs, Wavelet must suitably route the output channels $\vec{d}'$ and $\vec{r}'$ based on the tail condition t: if t is true (i.e., tail call), then the tail arguments $\vec{r}'$ are routed to the initial merge for recursion and $\vec{d}'$ are discarded via a steer; if t is false (i.e., return), then $\vec{d}'$ are routed to the final outputs $\vec{d}$, and $\vec{r}'$ are discarded by the second steer.

Compiling an Operator Call. Handling operator (or function) calls is comparatively simple; an $\mathbb{L}_{let}$ operator is mapped to the corresponding atomic process in $\mathbb{L}_{flow}$, and suitable output channels are added to the context, where $\Sigma(\vec{x}) \equiv \Sigma(x_1), \ldots, \Sigma(x_n)$ (assuming all defined) for $\vec{x} \equiv x_1, \ldots, x_n$.

$$\mathsf{cfc}((\textbf{let } \vec{y} = \mathsf{op}(\vec{x});\ E), \Sigma, t, \vec{d}, \vec{r}) := \nu\, \vec{d}'.\mathsf{op}(\Sigma(\vec{x})) \to \vec{d}' \mid\mid \mathsf{cfc}(E, \Sigma[\vec{y} \mapsto \vec{d}'], t, \vec{d}, \vec{r})$$

Note that if op is a function, the static restrictions we put on $\mathbb{L}^*_{let}$ (§4.3) force this case to be a non-recursive call, and it is subsequently handled in the linking stage by essentially substituting in the subgraph compiled from the body of op.

Compiling Branching. Dataflow operators execute whenever their inputs are available, so Wavelet needs to ensure that the body of either branch does not directly use the channels from outside the branch. Otherwise, an operator in the "wrong" branch may be triggered. Therefore, Wavelet inserts steer operators as gates between live variables outside the branch and their uses inside each branch, so that the variables only flow to the path triggered by the branch condition. Naturally, after the steers and branch bodies, the compiler needs to combine the sets of outputs from both branches, which is done by three merge operators, one for each output channel. Formally, the definition is as follows, where $\vec{\Sigma}$ means treating Σ as a list of channels indexed by live variables, and $\vec{\Sigma_1}, \vec{\Sigma_2}$ are two lists of fresh channel names for each variable in Σ:

$$\begin{aligned}
&\mathsf{cfc}(\textbf{if } x \textbf{ then } E_1 \textbf{ else } E_2, \Sigma, t, \vec{d}, \vec{r}) := \nu\, \vec{\Sigma_1}, \vec{\Sigma_2}, t_1, t_2, \vec{d_1}, \vec{d_2}, \vec{r_1}, \vec{r_2}.\\
&\quad \mathsf{steer}_\top(\Sigma(x), \vec{\Sigma}) \to \vec{\Sigma_1} \mid\mid \mathsf{steer}_\bot(\Sigma(x), \vec{\Sigma}) \to \vec{\Sigma_2} \mid\mid && \text{(Steer live variables)}\\
&\quad \mathsf{cfc}(E_1, \Sigma_1, t_1, \vec{d_1}, \vec{r_1}) \mid\mid \mathsf{cfc}(E_2, \Sigma_2, t_2, \vec{d_2}, \vec{r_2}) \mid\mid && \text{(Compile branch bodies)}\\
&\quad \mathsf{merge}_{\mathsf{init}}(\Sigma(x), t_1, t_2) \to t \mid\mid && \text{(Merge tail condition)}\\
&\quad \mathsf{merge}_{\mathsf{init}}(\Sigma(x), \vec{d_1}, \vec{d_2}) \to \vec{d} \mid\mid \mathsf{merge}_{\mathsf{init}}(\Sigma(x), \vec{r_1}, \vec{r_2}) \to \vec{r} && \text{(Merge return and tail call)}
\end{aligned}$$

Compiling a Return or Tail Recursion. Returning in $\mathbb{L}_{flow}$ means sending values to the output channels ($\vec{d}$), while a tail call means sending the tail call arguments back to the arguments of the current function through back edges ($\vec{r}$). Additionally, these processes should set the tail condition t, and send dummy values to the unused channels ($\vec{r}$ in the case of return, and $\vec{d}$ in the case of a tail call) so that merge operators compiled from any branching proceed correctly. In other words:

$$\mathsf{cfc}(\textbf{ret}(\vec{x}), \Sigma, t, \vec{d}, \vec{r}) := \mathsf{const}_\bot \to t \mid\mid \mathsf{const}_{\textbf{unit}} \to \vec{r} \mid\mid \mathsf{forward}(\Sigma(\vec{x})) \to \vec{d} \mid\mid \mathsf{sink}(\vec{\Sigma} \setminus \vec{x})$$

$$\mathsf{cfc}(\textbf{tail}(\vec{x}), \Sigma, t, \vec{d}, \vec{r}) := \mathsf{const}_\top \to t \mid\mid \mathsf{const}_{\textbf{unit}} \to \vec{d} \mid\mid \mathsf{forward}(\Sigma(\vec{x})) \to \vec{r} \mid\mid \mathsf{sink}(\vec{\Sigma} \setminus \vec{x})$$

Unlike the case of a normal non-recursive function call ($\mathsf{cfc}(\textbf{let } \vec{y} = \mathsf{op}(\vec{x});\ E)$), our static restrictions (§4.3) do *not* prevent multiple tail calls in the same function. This is acceptable because if we have multiple tail calls in different branches of a function, the tail call arguments will eventually be merged by the merge gates compiled in the branching case above, and routed back to the start of the body with back edges.

This also alludes to the reason why we put the static restrictions in $\mathbb{L}^*_{let}$ in the first place. Within the same function body, the condition for merging arguments to a function is exactly the tail condition t, since the arguments can either come from an external caller, or from any of the tail calls within the function. However, if we do allow multiple external call sites to a function, the conditions for merging these external sets of arguments are not *local*, and we must sacrifice the compositionality of the compiler definition to perform more complex global analyses.

Forward Simulation. We now sketch our proof of forward simulation of the CFC pass (the full version has been formalized in Lean), which intuitively means that the output dataflow graph has at least one schedule that behaves the same as the input $\mathbb{L}_{let}$ function.

The main difficulty of establishing a simulation relation for this translation is that a single use of a variable in $\mathbb{L}_{let}$ can explode into a tree of channels in $\mathbb{L}_{flow}$. This is largely due to steering in a

dataflow graph, where any live variable needs to be routed to the correct branch, and the results of both branches need to be merged at the end. As an example, consider the simple conditional expression below: the variable z has three uses, but the compiled dataflow graph has at least 12 channels (those named c_{z_i}) that receive the value of z at some point in the execution!

$$E \equiv \textbf{let } z = \mathrm{op}(x, y);\ \textbf{if } z \textbf{ then } \mathbf{ret}(x, z) \textbf{ else } \mathbf{ret}(y, z)$$

$$\begin{aligned}
&\mathrm{cfc}(E, \emptyset, t, \vec{d}, \vec{r}) = \mathrm{op}(c_x, c_y) \to c_{z_1} \mid\mid \mathrm{fork}(c_{z_1}) \to c_{z_2} \dots c_{z_{10}} \mid\mid && \text{(Run op and copy } z\text{)}\\
&\quad \mathrm{steer}_\top(c_{z_2}, c_x) \to c_{x'} \mid\mid \mathrm{steer}_\top(c_{z_3}, c_{z_4}) \to c_{z_{11}} \mid\mid && \text{(Route to true branch)}\\
&\quad \mathrm{steer}_\bot(c_{z_5}, c_y) \to c_{y'} \mid\mid \mathrm{steer}_\bot(c_{z_6}, c_{z_7}) \to c_{z_{12}} \mid\mid \dots \mid\mid && \text{(Route to false branch)}\\
&\quad \mathrm{merge}(c_{z_8}, \dots) \to \vec{d} \mid\mid \mathrm{merge}(c_{z_9}, \dots) \to \vec{r} \mid\mid \mathrm{merge}(c_{z_{10}}, \dots) \to t && \text{(Merge outputs)}
\end{aligned}$$

To alleviate the issue and make the simulation proof tractable, we make two important design decisions. First, we enforce affinity (i.e., no re-use) of variables and require no shadowing of variable names (§4), essentially requiring $\mathbb{L}_{let}$ programs to be in an "affine" SSA form [2]. This "well-formedness" restriction greatly reduces the complexity of tracking the liveness of a variable.

Given this restriction, the second part of the solution is to use a consistent naming scheme in the compiler definition, so that each use of a variable x in the source program corresponds to a *unique* channel name that in essence tracks provenance.

Following is a snippet of our Lean definition of channel names.

```
inductive ChanName X where | var (base : X) (pathConds : List (Bool x ChanName X))
                           | steer_cond (chan : ChanName X) -- Other 10 cases omitted
```

The parameter `X` is the type of variable names in the source program, and after CFC, all variables in the compiled $\mathbb{L}_{flow}$ process have type `ChanName X`. The most common case of channel names is the `var` case, where `base` is the original variable name, and `pathConds` keeps track of the sequence of branching decisions required to reach the program point at which the variable occurs. Path conditions can in turn inductively contain `ChanName`, because the branching conditions themselves may be "gated" by other branching conditions. Other cases such as `steer_cond` are used when the variable is the decider of a steer, merge, etc. In total, we use 12 cases of `ChanName` to disambiguate all channel names in the produced dataflow graph.

With the SSA form and channel naming scheme, the rest of the simulation proof is relatively standard. The simulation relation keeps track of the current path condition (a list of branching conditions used and their Boolean value) and of the key correspondence between the local variable store of $[\![\mathbb{L}_{let}]\!]$ and channel buffers of $[\![\mathbb{L}_{flow}]\!]$. The simulation proof shows that this relation is preserved after one step of the input $\mathbb{L}_{let}$ function $[\![D]\!]$ and potentially multiple steps of $[\![\mathrm{cfc}(D)]\!]$.

Theorem 5.2. *For a well-formed $\mathbb{L}_{let}$ definition D, $[\![D]\!] \lesssim [\![\mathrm{cfc}(D)]\!]$. Furthermore, the initial state of $[\![D]\!]$ is only related to the initial state of $[\![\mathrm{cfc}(D)]\!]$ in the simulation.*

The second part of the statement guarantees that after D returns, $\mathrm{cfc}(D)$ restores to its initial state, which ensures deadlock-freedom even when the function is called multiple times.

5.3 Linking

We can now lower a sequential $\mathbb{L}_{let}$ function to a parallel $\mathbb{L}_{flow}$ process that is guaranteed to simulate the source semantics. However, the compiler still has not handled calls to other functions, and instead treats them as uninterpreted synchronous operators. This intentional design decision drastically improves proof modularity; if Wavelet performs linking as a part of CFC, the final simulation relation has to relate the entire call stack of an $\mathbb{L}_{let}$ program to the channel buffers of $\mathbb{L}_{flow}$, which complicates the already involved channel naming scheme in §5.2. Instead, by splitting

$$\frac{c_0 \xrightarrow{l}_0 c_0' \qquad l \in \{\mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}}) \mid \mathsf{op} \in O\} \cup \{\mathsf{input}(\overrightarrow{v}), \mathsf{output}(\overrightarrow{v}), \tau\}}{(0, c_0, c_1, \ldots, c_n) \xrightarrow{l} (0, c_0', c_1, \ldots, c_n)}\ \textsc{Base}$$

$$\frac{c_0 \xrightarrow{\mathsf{yield}(f_i, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})}_0 c_0' \quad c_i \xrightarrow{\mathsf{input}(\overrightarrow{v_{in}})}_i c_i'}{(0, c_0, \ldots, c_i, \ldots) \xrightarrow{\tau} (i, c_0, \ldots, c_i', \ldots)}\ \textsc{Spawn} \qquad \frac{c_0 \xrightarrow{\mathsf{yield}(f_i, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})}_0 c_0' \quad c_i \xrightarrow{\mathsf{output}(\overrightarrow{v_{out}})}_i c_i'}{(i, c_0, \ldots, c_i, \ldots) \xrightarrow{\tau} (0, c_0', \ldots, c_i', \ldots)}\ \textsc{Ret}$$

Fig. 9. Selected transition rules of $\mathsf{link}_{sem}(S, S_1, \ldots, S_n)$ for linking a base $S \in \mathsf{LTS}_C^{Label_{O \sqcup \{f_1, \ldots, f_n\}}}$, and dependent LTSs $\{S_i \in \mathsf{LTS}_{C_i}^{Label_O}\}_{i=1}^n$. Note that only the base LTS can "call" the dependent LTSs. The configuration of the linked LTS is $\mathbb{N} \times C \times C_1 \times \cdots \times C_n$, with the first index denoting the currently active LTS.

the two phases, the proofs in §5.2 only need to focus on a single call frame and the corresponding dataflow subgraph, leaving linking as a separate reasoning layer, which we explain now.

To compile the whole program, we must reason about function linking in both $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$, in a way that provably connects their semantics. Thus, Wavelet uses two notions of linking for this pass: (1) *semantic* linking link_{sem}, which gives semantics to function calls in $\mathbb{L}_{let}$; and (2) *syntactic* linking link_{syn}, which "stitches" compiled $\mathbb{L}_{flow}$ processes together into a single $\mathbb{L}_{flow}$ process.

Semantic Linking. The high-level idea of semantic linking is to take the semantics (i.e., LTS) of each individual function definition (with uninterpreted function symbols), and construct a single LTS in which each component function can call one another. We leverage the fact that we enforce an acyclic call graph in $\mathbb{L}_{let}$ programs (§3) to have a (relatively) simple definition where a *base* LTS can make calls to a list of *dependent* LTSs without mutual recursion. We show a snippet of our definition in Fig. 9, which is partially inspired by *interaction semantics* in compositional CompCert [47], though more relaxed due to our call graph restrictions.

The definition revolves around a base LTS S that can call operators in O as well as function symbols $f_1, \ldots, f_n$, which correspond to dependent LTSs $\{S_i\}_{i=1}^n$ that only call operators in O. The linked LTS, denoted $\mathsf{link}_{sem}(S, S_1, \ldots, S_n)$, is then an LTS that only calls operators in O, with function calls "internalized" as silent steps. This linked LTS has its own transition relation. Intuitively, this relation treats all LTSs as parallel threads, with the current active thread indicated by the first index in the state. If a thread produces a label that does not involve interaction between the threads, the label is simply passed through. However, if the base LTS has a potential transition to produce $\mathsf{yield}(f_i, \overrightarrow{v_{in}}, \ldots)$, and a dependent LTS S_i can accept an $\mathsf{input}(\overrightarrow{v_{in}})$ label, then they synchronize and the active thread switches to S_i (Spawn). Once S_i finishes execution and produces an $\mathsf{output}(\overrightarrow{v_{out}})$ label, the base LTS resumes execution (Ret). One point to note is that no component LTS ever resets to its initial state init_i. While the state of a function "resetting" after returning makes sense in the sequential $\mathbb{L}_{let}$ setting, this is not generally true in the $\mathbb{L}_{flow}$ semantics (even though it is satisfied by a process produced by CFC). Therefore, we define the linked semantics this way to avoid dependency on the simulation proof of CFC and preserve modularity.

As an example of semantic linking, suppose we have the following two simple functions

$$\textbf{def}\ f(x) = g(x) + 1; \textbf{def}\ g(x) = x + 1$$

The LTS $[\![f]\!]$ can have labels $\mathsf{yield}(g, \overrightarrow{v_{\text{in}}}, \overrightarrow{v_{\text{out}}})$ and $\mathsf{yield}(+, \overrightarrow{v_{\text{in}}}, \overrightarrow{v_{\text{out}}})$, while the LTS $[\![g]\!]$ can only generate $\mathsf{yield}(+, \overrightarrow{v_{\text{in}}}, \overrightarrow{v_{\text{out}}})$ (besides input/output/silent labels). Then $\mathsf{link}_{sem}([\![f]\!], [\![g]\!])$ would act like the *parallel composition* of the two LTSs that interact when a call happens: whenever $[\![f]\!]$ yields to (calls) g, an input label will be sent to $[\![g]\!]$; whenever $[\![g]\!]$ returns, the return values are passed back to $[\![f]\!]$ through the yield label $\mathsf{yield}(g, \overrightarrow{v_{\text{in}}}, \overrightarrow{v_{\text{out}}})$ and the execution of f continues.

In this semantic version of linking, we prove a congruence lemma of link_{sem} with respect to $\lesssim$.

Lemma 5.3. *Given two sets of LTSs $S_0, \ldots, S_n$ and $S'_0, \ldots, S'_n$ such that $S_i \lesssim S'_i$ for all $i = 0, \ldots, n$, $\mathsf{link}_{sem}(S_0, \ldots, S_n) \lesssim \mathsf{link}_{sem}(S'_0, \ldots, S'_n)$.*

Syntactic Linking. We can use Lemma 5.3 to prove forward simulation of semantically linked processes, i.e., $\mathsf{link}_{sem}(\llbracket D_1 \rrbracket, \ldots, \llbracket D_n \rrbracket) \lesssim \mathsf{link}_{sem}(\llbracket \mathsf{cfc}(D_1) \rrbracket, \ldots, \llbracket \mathsf{cfc}(D_n) \rrbracket)$. This result has a crucial gap, however, as the final compilation result of a function should be a single dataflow graph, not a collection of function graphs. Therefore, we define a *syntactic linking* operation for $\mathbb{L}_{flow}$ processes, denoted $\mathsf{link}_{syn}(\pi, \pi_1, \ldots, \pi_n)$, that replaces any operator call to f_i in π with the corresponding process π_i. This is defined inductively on an $\mathbb{L}_{flow}$ process, and we show the key rule below, where $\pi_i(\vec{c}) \to \vec{d}$ denotes replacing the free input (output) channels of π_i with $\vec{c}$ ($\vec{d}$).

$$\mathsf{link}_{syn}((\mathsf{op}(\vec{c}) \to \vec{d}), \pi_1, \ldots, \pi_n) := \begin{cases} \pi_i(\vec{c}) \to \vec{d} & \text{if } \mathsf{op} = f_i \\ \mathsf{op}(\vec{c}) \to \vec{d} & \text{otherwise} \end{cases}$$

To connect the semantic and syntactic notions of linking, we prove the following lemma saying that a syntactically linked $\mathbb{L}_{flow}$ process includes the behavior of semantically linking the processes.

Lemma 5.4 (Linking Simulation). $\mathsf{link}_{sem}(\llbracket \pi \rrbracket, \llbracket \pi_1 \rrbracket, \ldots, \llbracket \pi_n \rrbracket) \lesssim \llbracket \mathsf{link}_{syn}(\pi, \pi_1, \ldots, \pi_n) \rrbracket$

Final Simulation. By connecting Theorem 5.2 and Lemmas 5.3 and 5.4, we can now state our final forward simulation theorem from $\mathbb{L}_{let}$ programs to $\mathbb{L}_{flow}$ processes, which shows that the combined passes of CFC and linking produce a dataflow graph that *includes* the behavior of the $\mathbb{L}_{let}$ program.

For an $\mathbb{L}_{let}$ program $P = D_1, \ldots, D_n$, we say that it is *well-formed* if each definition D_i is well-formed (i.e., affine variable usage and no shadowing of variable names), and P satisfies the static restrictions in §4.3. We denote the semantics of P as $\llbracket P \rrbracket := \mathsf{link}_{sem}(\llbracket D_1 \rrbracket, \ldots, \llbracket D_n \rrbracket)$ and the final compilation pass $\mathsf{comp}(P) := \mathsf{link}_{syn}(\mathsf{cfc}(D_1), \ldots, \mathsf{cfc}(D_n))$.

Theorem 5.5 (Compiler Forward Simulation). *For a well-formed program P, $\llbracket P \rrbracket \lesssim \llbracket \mathsf{comp}(P) \rrbracket$.*

5.4 Guarded Semantics and Determinacy

The final forward simulation (Theorem 5.5) shows that the produced dataflow graph will have *at least one* schedule that behaves the same as the source $\mathbb{L}_{let}$ program. However, due to the non-deterministic execution of $\mathbb{L}_{flow}$ processes and shared memory between operators, execution may diverge from this "good" schedule and produce different results or termination behaviors.

In other words, we need to prove *determinacy*: assuming termination, the compiled program produces the same result regardless of the schedule. To establish this, we re-introduce ghost permission tokens from §4.2. We add a permission-checking layer to both $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$ semantics that verifies that operators are called with correct permissions, getting stuck on violations. This approach enables us to prove two key properties: (1) if one "good" terminating trace of an $\mathbb{L}_{flow}$ process is found satisfying the permission specification, then the $\mathbb{L}_{flow}$ process is strongly normalizing in *all* schedules; (2) compilation preserves permission-satisfying, terminating traces, so that it is sufficient to check the permissions at the $\mathbb{L}_{let}$ level (§4).

Guarded Semantics. Given a permission specification for all operators (§4.2), we want to have a semantic "permission checker" that checks whether each operator call satisfies the permission specification and gets stuck otherwise. As a first step, we need a "hook" to reason about permissions within our semantics, so we slightly augment our operator set O to create an extended set $\mathsf{ext}(O)$. The key difference of $\mathsf{ext}(O)$ is that each operator is given an additional input and output channel specifically for ghost permission tokens (mirroring $\mathbb{L}_{let}$'s structure in §4.2), so that permission

$$\frac{t_{pre} = \mathsf{pre}(\mathsf{op}, \overrightarrow{v_{in}}) \qquad t_{post} = \mathsf{post}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})}{\mathsf{yield}(\mathsf{op}, (\overrightarrow{v_{in}}, t_{pre}), (\overrightarrow{v_{out}}, t_{post})) \triangleright_{\mathsf{spec}} \mathsf{yield}(\mathsf{op}, \overrightarrow{v_{in}}, \overrightarrow{v_{out}})}\ \textsc{Guard-Yield}$$

$$\frac{f(\overrightarrow{v}) \cdot t' = t_1 \cdot t_2 \cdot \cdots \cdot t_k}{\mathsf{yield}(\mathsf{jnsplt}_{k,f}, (t_1, \ldots, t_k, \overrightarrow{v}), (f(\overrightarrow{v}), t')) \triangleright_{\mathsf{spec}} \tau}\ \textsc{Guard-Join} \qquad \frac{l \in \{\mathsf{input}(\overrightarrow{v}), \mathsf{output}(\overrightarrow{v}), \tau\}}{l \triangleright_{\mathsf{spec}} l}\ \textsc{Other}$$

Fig. 10. Semantic guard for interpreting $\mathsf{ext}(O)$ with respect to a specification $\mathsf{spec} = (\mathsf{pre}, \mathsf{post})$. This relation interprets the extended operator set $\mathsf{ext}(O)$ as operations in the base operator set O.

tokens will be included in the yield labels produced by both $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$ semantics. Additionally, $\mathsf{ext}(O)$ contains the ghost operator jnsplt in §4.2, which is used to split and join permission tokens.

Now that we have a semantics with permission tokens threaded through, we can create a checker to make sure that any operator call satisfies the permission spec. To do so, we define a notion of a *guarded* semantics, which is in essence an LTS with a restricted and translated label set.

Definition 5.6 (Guarded Semantics). Given a relation $G \subseteq Label_{O'} \times Label_O$ which we call *guard*, and an $S = (\mathsf{init}, \rightarrow) \in \mathsf{LTS}_C^{Label_{O'}}$, $\mathsf{guard}_G(S) = (\mathsf{init}_G, \rightarrow_G) \in \mathsf{LTS}_C^{Label_O}$ is another LTS such that $\mathsf{init}_G := \mathsf{init}$, and for any $c, c' \in C$, $c \xrightarrow{l}_G c'$ iff there exists l' such that $c \xrightarrow{l'} c'$ and $(l', l) \in G$.

We now define a guard relation $\mathsf{guard}_{\mathsf{spec}}(S)$ in Fig. 10 for checking the operator calls ($\mathsf{yield}(\ldots)$) against the permission specification. In the Guard-Yield rule, we check that the additional permission token input and output of the operator op satisfy the permission specification, and translate the label back to the base operator set O. In the Guard-Join rule, we take the sum of the input permissions and then split it as specified by the $\mathsf{jnsplt}_{k,f}$ operator, where k is the number of input permissions, and f determines how the two output permissions should be split, potentially depending on input values $\overrightarrow{v}$. Since this is a ghost operator, it produces no external yields and emits a τ label. Finally, in the Other rule, we pass through any other labels unchanged.

To reason about the base unguarded semantics, we also define a trivial guard $\mathsf{guard}_{\mathsf{triv}}$ (omitted for space), which is similar to $\mathsf{guard}_{\mathsf{spec}}$ but skips all permission checks.

Strong Normalization. With our permission checker, we can prove properties about how the guarded semantics of $\mathbb{L}_{flow}$ processes relate to the unguarded semantics. Namely, if we can find a trace in the guarded semantics that leads to a final state in both semantics, then even without the guards, any execution trace in the unguarded semantics will be "eventually consistent" with the same final state. Furthermore, the unguarded semantics in this case will be strongly normalizing, as all traces will be bounded by the length of the terminating, guarded trace.

Theorem 5.7 (Guarded Weak Normalization Implies Unguarded Strong Normalization). *Let $c \in C^{flow}$ be an $\mathbb{L}_{flow}$ configuration, and let $\rightarrow_{\mathsf{spec}}$ and $\rightarrow_{\mathsf{triv}}$ be the guarded/unguarded semantics of $\mathbb{L}_{flow}$, respectively. If there exists a trace $c \xrightarrow{\tau^k}_{\mathsf{spec}} c_\perp$ (of k steps) such that $c_\perp$ is final with respect to $\rightarrow_{\mathsf{triv}}$, then for any trace $c \xrightarrow{\tau^{k_1}}_{\mathsf{triv}} c'$ in the unguarded semantics, there exists a trace $c' \xrightarrow{\tau^{k_2}}_{\mathsf{triv}} c_\perp$ in the unguarded semantics to the same final state $c_\perp$, and $k_1 + k_2 = k$.*

Note that the statement only considers τ labels. This is because in a step omitted from the paper for brevity, we concretely interpret all previously uninterpreted operator calls, so the resulting semantics only generates input/output/silent labels but no yield labels.

Connecting with Forward Simulation. We have now established good properties at two ends of the compilation pipeline: (1) via the type system in §4, we know that well-typed $\mathbb{L}^*_{let}$ programs can

be elaborated to $\mathbb{L}_{let}$ programs that will satisfy the permission checker; (2) via Theorem 5.7, we know that if the compiled $\mathbb{L}_{flow}$ process satisfies the permission checker and has a "good" terminating trace, then it is strongly normalizing. To bridge the gap between (1) and (2), we use the following key congruence lemma that preserves the *modularity* between forward simulation and determinacy.

Lemma 5.8. *For any guard relation G and LTSs S and S', $S \lesssim S'$ implies* $\mathsf{guard}_G(S) \lesssim \mathsf{guard}_G(S')$.

Therefore, by carrying a terminating, guarded trace of $\mathbb{L}_{let}$ programs (guaranteed by the type system) through the forward simulation proofs and applying Theorem 5.7, we can prove the following determinacy result, which says that the compiled $\mathbb{L}_{flow}$ process $\mathsf{comp}(P)$ is strongly normalizing and, in particular, produces the same output values $\overrightarrow{v_{out}}$ as the input $\mathbb{L}_{let}$ program P.

Theorem 5.9 (Compiler Determinacy). *Let P be a well-formed $\mathbb{L}_{let}$ program and* $\mathsf{comp}(P)$ *its compiled $\mathbb{L}_{flow}$ process. Suppose that there is a terminating trace in the guarded $\mathbb{L}_{let}$ semantics:*

$$\mathsf{init}(P) \xrightarrow{\mathsf{input}(\overrightarrow{v_{in}})}_{\mathsf{spec}} c_1 \xrightarrow{\tau^k}_{\mathsf{spec}} c_2 \xrightarrow{\mathsf{output}(\overrightarrow{v_{out}})}_{\mathsf{spec}} c_\perp$$

Then there exists $k' \in \mathbb{N}$, such that: for any c_2' with $\mathsf{init}(\mathsf{comp}(P)) \xrightarrow{\mathsf{input}(\overrightarrow{v_{in}})\ \tau^{k_1}}_{\mathsf{triv}} c_2'$, *we have* $c_2' \xrightarrow{\tau^{k_2}\ \mathsf{output}(\overrightarrow{v_{out}})}_{\mathsf{triv}} c_\perp'$ *with $k_1 + k_2 + 2 = k'$ and $c_\perp$ and $c_\perp'$ have the same final memory state.*

5.5 Operator Selection and Optimization

As the final step in Wavelet, we implement a term rewriter on the produced $\mathbb{L}_{flow}$ process to lower "pseudo-operators" to actual hardware operators and optimize for a smaller graph size. In particular, any ghost permission tokens become simple control signals, and a jnsplt is lowered to an order operator and a fork operator, which together wait for all inputs and then output two control signals.

In total, we have 54 rewrite rules, and they are currently unverified. The formal verification of these rewrites is an interesting problem on its own (§7). For instance, we have the rewrite rule $\mathsf{merge}(c_0, c_1, c_2) \to d \mid\mid \mathsf{steer}_\top(c_0, d) \to d' \Rightarrow \mathsf{forward}(c_2) \to d'$, i.e., if a merge and a steer have the same decider (c_0), then c_2 can be directly forwarded to the final output d'. This rewrite may not preserve the original behavior if, for example, c_2 is ready, but the decider c_0 is always empty. Hence, the correctness of this rule relies on the liveness property that any value will eventually be consumed, which is implied by the termination assumption (Theorem 5.7) and our simulation relation that enforces empty channels in the final dataflow configuration.

6 Evaluation

In this section, we evaluate Wavelet in terms of compilation quality (§6.1), ergonomic/compiler overhead of $\mathbb{L}_{let}^*$ (§6.2), and human effort in verifying Wavelet (§6.3). All evaluations are performed on a machine with Apple M1 Pro CPU and 32 GiB of RAM.

6.1 Compilation Quality

Wavelet targets CGRAs such as RipTide [16] or FPGAs as a part of a dynamically-scheduled high-level synthesis (HLS) pipeline [21, 36]. Therefore, we compare the performance and resource usage of dataflow graphs compiled from Wavelet and two unverified dataflow compilers from RipTide [16] and LLVM CIRCT [36] (for HLS) on 10 benchmark programs from RipTide.

Comparison with RipTide. The RipTide project [16] features an LLVM-based dataflow compiler targeting the RipTide CGRA. The output dataflow graphs 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 Fig. 11. Simulators of Wavelet and RipTide are both not cycle-accurate (every operator takes one cycle), but their relative performance is still informative.

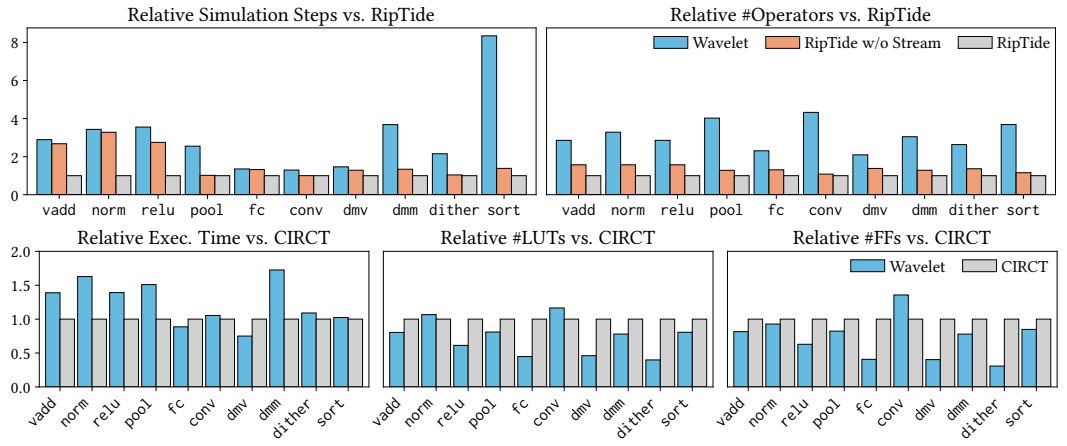


Fig. 11. Compilation quality: the first row compares simulation performance and graph size of Wavelet and RipTide; the second row compares execution time and resource usage of Wavelet and CIRCT in dynamic HLS.

Compared to RipTide, Wavelet's dataflow graphs are 2.62× slower and 3.04× larger in geometric mean. Without optimizations in §5.5, Wavelet's dataflow graphs are 5.79× slower and 8.26× larger. One advantage of RipTide is that it has a "streamification" optimization to replace a loop header similar to Fig. 1 (a) with a specialized "stream" operator, which we do not yet support in Wavelet. With streamification disabled in RipTide for a fairer comparison, Wavelet's dataflow graphs are 1.69× (3.74× if unoptimized) slower and 2.26× (6.13× if unoptimized) larger.

The main downside of Wavelet is that it produces significantly more control-flow operators (e.g., merge, steer, etc.). In the benchmarks, Wavelet-compiled graphs have 78% control-flow operators on average (93% if unoptimized), 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 (§5.4). 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 produces graphs with 2× as many operators as RipTide (without streamification), the final performance is still similar due to pipelining.

Comparison with CIRCT. CIRCT [36] is an MLIR-based compiler framework for hardware design. We compare Wavelet against a dynamic HLS pipeline in CIRCT, which compiles MLIR's structured control-flow dialect (`scf` [38]) to dataflow graphs (CIRCT's `handshake` dialect [37]). For Wavelet, we implement an unverified lowering pass from $\mathbb{L}_{flow}$ to the `handshake` dialect.

We then compile the produced `handshake` dataflow graphs to SystemVerilog RTL designs via CIRCT, and use Verilator [51] for simulation. To estimate cycle periods and resource utilization, we synthesize the designs targeting the Lattice ECP5 FPGA [44] using Yosys [55] and nextpnr [54].

The resulting comparison is shown in the second row of Fig. 11, including relative execution time and resource utilization of the generated designs from Wavelet and CIRCT. The results show that

Wavelet's dataflow graphs, 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× (2.04× if unoptimized) slower in execution time, and have 0.69× (1.22× if unoptimized) LUTs and 0.67× (1.27× if unoptimized) FFs compared to CIRCT's output.

During evaluation, we actually found two issues in CIRCT [30, 31], which further highlight the challenge of building a correct dataflow compiler.

6.2 $\mathbb{L}^*_{let}$ Type System Overhead

To guarantee dataflow determinacy and to enable modular compiler proofs, our frontend language $\mathbb{L}^*_{let}$ adopts a capability type system (§4) and two static restrictions (§4.3). In this section, we evaluate both the compiler-performance and ergonomic overhead of the type system.

Fig. 12 shows statistics of our benchmarks and compiler performance. In terms of usability, benchmarks written in our $\mathbb{L}^*_{let}$ DSL have on average 4.86× more lines of code than the original versions in C. While the ratio is high, this is primarily due to the lack of automated desugaring passes in our frontend. Our DSL enforces individual operations in separate statements to aid compilation (e.g., `x = a + b * c` is written as `t = b * c; x = a + t`), and the two static restrictions in §4.3 require users to manually convert their control flow to branching and tail recursion.

	Source LoC			Compiler Time (s)			
Test	C	DSL	Ann	TC	TV	Opt	Total
`vadd`	5	26	2	0.03	0.67	0.02	0.72
`norm`	5	27	2	0.02	0.33	0.03	0.39
`relu`	7	30	3	0.02	0.39	0.06	0.47
`pool`	27	135	5	0.06	1.07	1.31	2.45
`fc`	16	71	4	0.09	1.76	0.23	2.08
`conv`	36	179	5	0.11	3.94	2.94	7.00
`dmv`	9	47	3	0.07	1.64	0.09	1.80
`dmm`	16	85	4	0.10	2.41	0.55	3.06
`dither`	19	75	7	0.09	1.26	0.30	1.66
`sort`	23	117	12	0.04	1.77	1.97	3.79

Fig. 12. Benchmark lines of code (in the **C** version from RipTide [16], in Wavelet's **DSL**, and for capability **Ann**otations), along with their compilation time in various passes of Wavelet: type checking (**TC**), translation validation (**TV**), optimization (**Opt**). Omitted CFC and linking took less than 10 ms.

Our capability types (§4) place a relatively small burden on the programmer. Fig. 12 shows that users only have to add a few lines of annotations. While precise capabilities and fences do require careful memory reasoning, this overhead is reasonable given the strong assurance from Wavelet.

Most compilation time is spent on translation validation (§4.3) and dataflow optimizations (§5.5). For translation validation, some test programs with more complex array index computation can lead to inefficient non-linear integer arithmetic in the SMT queries. For dataflow optimizations, the performance issue is largely due to inefficient internal representations of dataflow graphs.

6.3 Verification Effort

In total, the Lean formalization of Wavelet has 797 lines of verified executable code, 1,534 lines of specifications, and 14,819 lines of proofs, with a proof-to-code/spec ratio of about 6.4:1. The type checker and permission validator are implemented in 11,546 lines of Rust.

The most difficult proofs of Wavelet are for control flow conversion (~2 person-months and 6,353 lines of proofs) and determinacy (~2 person-months and 5,862 lines of proofs). Defining the channel naming schemes (§5.2) and static restrictions (§4.3) is the main difficulty of control flow conversion. The determinacy proofs require careful coordination with the frontend in order to achieve modularity, and the exact formulation of the guarded semantics also took some iterations.

7 Related and Future Work

The idea of dataflow is ubiquitous, and has been studied in various contexts including dataflow architectures, stream processing [23, 39], and synchronous data flow [27]. We discuss related formal verification efforts in these areas and highlight exciting future directions for Wavelet.

Dataflow and HLS. FlowCert [34] targets CGRA compiler correctness and inspired our modular determinacy proof. The main difference is that FlowCert is entirely a translation validation approach and does not provide the same level of formal guarantee. Wavelet also improves upon FlowCert with fine-grained capabilities, enabling more pipelining than FlowCert's simple permission tokens.

Our type system is inspired by Dahlia [41], but we extend memory capabilities with dependency on values, which is crucial for pipelining. Vericert [19] is a formally verified HLS compiler in Rocq [50], although it cannot be directly applied to asynchronous dataflow due to static scheduling.

The Dynamatic [21] framework for dynamically-scheduled HLS also uses asynchronous dataflow extensively. Law et al. [26] formalize two Dynamatic-inspired dataflow models in Rocq. Their dataflow calculus Cilan is an alternative to $\mathbb{L}_{flow}$, but they do not allow shared memory between dataflow components, which simplifies their determinacy proof. They also do not verify our main focus of dataflow compilation from a sequential program. Graphiti [18] and ElasticMiter [12] are two prior efforts on verifying dataflow graph rewrites for Dynamatic, which are complementary to our work. Connecting the formalizations of Wavelet and Graphiti would be an interesting future direction, since our proof structure supports future extensions to some extent: rewrites with verified forward simulation can be added after linking without affecting the determinacy proof (Lemma 5.8).

Synchronous Data Flow. The more restricted model of synchronous data flow [27] has been extensively studied, with verification efforts including Kind 2 [9], a model checker for Lustre [17], and Vélus [6, 7], a verified compiler in Rocq from Lustre to CompCert's Clight [28]. *Asynchronous* dataflow in our case presents a different set of challenges, e.g., determinacy and shared memory.

Stream Processing. Stream processing and functional reactive programming [13] are closely related paradigms. Recent works such as Flo [23] and stream types [10] share similar goals, including determinism and liveness, but they target more abstract features like higher-order streams rather than low-level concerns like shared memory, making them not directly applicable to our challenges.

Affine Types and Ghost Permissions. Affine types and permission-based systems have been explored in languages such as Rust [49] and Linear Haskell [5], and in various program logics and verification tools, such as Verus [24, 25], Linear Dafny [29], and Iris [22]. Our capability type system in $\mathbb{L}^*_{let}$ is inspired by these prior works, but tailored specifically for reasoning about memory access ordering and pipelining in dataflow architectures. Exploring richer capability types and more powerful program logics for dataflow would be an interesting future direction.

Compiler Verification. We build on ideas from a rich body of work on compositional compiler verification [46, 47, 56]. Our definition of semantic linking is partially inspired by linking interactive semantics in compositional CompCert [47] and interaction trees [52].

Handling Complex Control Flow. The dataflow model formalized in this work is also known as ordered dataflow [16], where channels have the semantics of FIFO queues. Prior (unverified) work [16, 21] in this model only handles single-function compilation. In our case, the entire $\mathbb{L}^*_{let}$ program with multiple functions can be thought of as a single program in SSA form [3]. Our approach of using the two static restrictions (§4.3) is inspired by RipTide [16], which normalizes the input SSA program to a form with nested loops and branching. Another approach [21, 36] is to use non-deterministic merge operators, but they further complicate the determinacy proof due to the additional non-determinism [26]. We leave for future work the verification of *tagged dataflow* [1, 4], which can support more expressive control flow (e.g., higher-order functions in Id [4]).

Acknowledgments

We sincerely thank Joshua Gancher and Bryan Parno for their guidance on the early stage of this project. We are grateful to Stephanie Balzer, Bas van den Heuvel, Peter Thiemann, Jan Hoffmann, and Cesare Tinelli for discussions on deadlocks in asynchronous dataflow. We thank Souradip Ghosh, Nathan Serafin, Mitchell Fream, Brandon Lucia, and Nathan Beckmann for their help on the RipTide and Tyr CGRAs. We thank the anonymous reviewers for their time and detailed feedback. This work is supported in part by the National Science Foundation under Grant No. 2403144. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Data Availability Statement

The source code and proofs of Wavelet can be found in our GitHub repository: https://github.com/plum-umd/wavelet. Our Docker image for artifact evaluation [32] contains all scripts and instructions to reproduce the evaluation results in §6.

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, Austin, TX, USA, 1184–1200. doi:10.1109/MICRO61859.2024.00089

[2] Alfred V. Aho and Jeffrey D. Ullman. 1977. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Andrew W. Appel. 1998. SSA is functional programming. *SIGPLAN Not.* 33, 4 (April 1998), 17–20. doi:10.1145/278283.278285

[4] 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

[5] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[6] 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

[7] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2019. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. ACM Program. Lang.* 4, POPL, Article 44 (Dec. 2019), 29 pages. doi:10.1145/3371112

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

[9] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 Model Checker. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 510–517. doi:10.1007/978-3-319-41540-6_29

[10] Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2024. Stream Types. *Proc. ACM Program. Lang.* 8, PLDI, Article 204 (June 2024), 25 pages. doi:10.1145/3656434

[11] 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

[12] 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

[13] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing

Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973
[14] 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
[15] 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
[16] 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
[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language Lustre. *Proc. IEEE* 79, 9 (1991), 1305–1320. doi:10.1109/5.97300
[18] 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* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 821–837. doi:10.1145/3779212.3790166
[19] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (Oct. 2021), 30 pages. doi:10.1145/3485494
[20] 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
[21] 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
[22] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (jan 2015), 637–650. doi:10.1145/2775051.2676980
[23] Shadaj Laddad, Alvin Cheung, Joseph M. Hellerstein, and Mae Milano. 2025. Flo: A Semantic Foundation for Progressive Stream Processing. *Proc. ACM Program. Lang.* 9, POPL, Article 9 (Jan. 2025), 30 pages. doi:10.1145/3704845
[24] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952
[25] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. doi:10.1145/3586037
[26] 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
[27] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. doi:10.1109/PROC.1987.13876
[28] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. doi:10.1145/1538788.1538814
[29] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear types for large-scale systems verification. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–28.
[30] Zhengyao Lin. 2026. CIRCT Issue 9807. https://github.com/llvm/circt/issues/9807.
[31] Zhengyao Lin. 2026. CIRCT PR 9587. https://github.com/llvm/circt/pull/9587.
[32] Zhengyao Lin, Yi Cai, and Milijana Surbatovich. 2026. *Let It Flow: A Formally Verified Compilation Framework for Asynchronous Dataflow*. doi:10.5281/zenodo.19393250
[33] Zhengyao Lin, Yi Cai, and Milijana Surbatovich. 2026. Supplementary Material. doi:10.1145/3808263
[34] 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

[35] 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
[36] LLVM. 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.
[37] LLVM. 2026. CIRCT: 'handshake' dialect. https://circt.llvm.org/docs/Dialects/Handshake/.
[38] LLVM. 2026. MLIR: 'scf' dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/.
[39] Konstantinos Mamouras. 2020. Semantic Foundations for Deterministic Dataflow and Stream Processing. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 394–427. doi:10.1007/978-3-030-44914-8_15
[40] 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
[41] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. doi:10.1145/3385412.3385974
[42] 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
[43] Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, USA.
[44] Lattice Semiconductor. 2026. Lattice ECP5 FPGA family. https://www.latticesemi.com/en/Products/FPGAandCPLD/ECP5.
[45] 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
[46] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. doi:10.1145/3371091
[47] 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
[48] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, USA, 291.
[49] The Rust team. 2026. Rust programming language. https://rust-lang.org/.
[50] The Rocq Development Team. 2025. The Rocq Theorem Prover. https://rocq-prover.org/.
[51] Verilator. 2026. Verilator open-source SystemVerilog simulator and lint system. https://github.com/verilator/verilator.
[52] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119
[53] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. 2014. Hybrid Dataflow/von-Neumann Architectures. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1489–1509. doi:10.1109/TPDS.2013.125
[54] Yosys. 2026. nextpnr portable FPGA place and route tool. https://github.com/YosysHQ/nextpnr.
[55] Yosys. 2026. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.
[56] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (Jan. 2024), 31 pages. doi:10.1145/3632914

Received 2025-11-14; accepted 2026-04-03