FlowCert: Translation Validation for Asynchronous Dataflow Programs via Dynamic Fractional Permissions

ANONYMOUS AUTHOR(S)

Coarse-grained reconfigurable arrays (CGRAs) have gained attention in recent years due to their promising power efficiency compared to traditional von Neumann architectures. To program these architectures using ordinary languages such as C, a dataflow compiler must transform the original sequential, imperative program into an equivalent dataflow graph, composed of dataflow operators running in parallel. This transformation is challenging since the asynchronous nature of dataflow graphs allows out-of-order execution of operators, leading to behaviors not present in the original imperative programs.

We address this challenge by developing a translation validation technique for dataflow compilers to ensure that the dataflow program has the same behavior as the original imperative program on all possible inputs and schedules of execution. We apply this method to a state-of-the-art dataflow compiler targeting the RipTide CGRA architecture. Our tool uncovers 8 compiler bugs where the compiler outputs incorrect dataflow graphs, including a data race that is otherwise hard to discover via testing. After repairing these bugs, our tool verifies the correct compilation of all programs in the RipTide benchmark suite.

ACM Reference Format:
Anonymous Author(s). 2024. FlowCert: Translation Validation for Asynchronous Dataflow Programs via Dynamic Fractional Permissions. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2024), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Even as Moore's Law ends, many applications still demand better power efficiency, e.g., for high-performance machine learning or for ultra-low-power environmental monitoring. A flood of new hardware architectures has emerged to meet this demand, but one attractive approach that retains general-purpose programmability is a class of dataflow architectures called *coarse-grained reconfigurable arrays* (CGRAs) [Liu et al. 2019]. In a CGRA architecture, an array of processing elements (PEs) is physically laid out on the chip. Each PE can be configured to function as a high-level *operator* that performs an arithmetic operation, a memory operation, or a control-flow operation. These PEs communicate via an on-chip network, reducing data movement costs compared to traditional von Neumann architectures, thus making a CGRA more power efficient.

Abstractly, CGRAs run *dataflow programs*, or networks of operators that execute independently and asynchronously. While CGRAs are able to run dataflow programs efficiently due to their inherent parallelism, this parallelism also introduces new, subtle correctness issues due to the potential for data races. As a result, dataflow programs are not written directly, but are instead *compiled* from sequential, imperative languages. However, the gap between imperative and dataflow programming means that these compilers are complex; to maximize performance, they must use a variety of program analyses to convert imperative programs into equivalent distributed ones.

As CGRA architectures continue to emerge and rapidly evolve, we argue that *formal verification* — mathematically sound proofs of system correctness — should be integrated early on; past experience

2475-1421/2024/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

teaches us that belatedly retrofitting them may be costly or simply infeasible. Since CGRAs cannot execute as intended without correct dataflow programs, we begin this effort by verifying the translation from imperative programs to dataflow programs.

In this work, we propose using *translation validation* [Pnueli et al. 1998] to certify the results of dataflow compilers. In our case, translation validation amounts to checking that the output dataflow program from the compiler has the same behaviors as the input imperative program. Compared to traditional testing, translation validation provides a much stronger guarantee, as it checks that the imperative and the dataflow programs behave the same on all possible inputs. Additionally, in contrast to full compiler verification, translation validation allows the use of unverified compilation toolchains, which is crucial for the fast-developing area of CGRA architectures.

Our main challenge for leveraging translation validation is managing the asynchrony of dataflow programs. Most translation validation works [Kundu et al. 2009; Necula 2000; Pnueli et al. 1998; Sharma et al. 2013] prove equivalence between input and output programs via *simulation*, or relational invariants between the states of the two programs. Since dataflow programs have exponentially more states than imperative programs (arising from asynchronous scheduling decisions), constructing this simulation relation directly would be quite tricky.

Instead, we utilize a two-phase approach. First, we use symbolic execution to prove that there is a simulation between the input imperative program and the output dataflow program *on a canonical schedule*. In the canonical schedule, the dataflow operators are scheduled to fire in a similar order to their counterparts in the original imperative program.

Next, we prove that any possible schedule of the dataflow program must converge to the same final state as the canonical schedule; i.e., it is *confluent* [Baader and Nipkow 1998]. For our setting, proving confluence requires showing that the dataflow program does not contain any data races. Inspired by fractional permissions [Boyland 2013], we augment our symbolic execution with *linear permission tokens* to track ownership of memory locations. In contrast with interactive program logics (e.g., concurrent separation logic [Brookes 2006]) that typically require manual annotations, we automatically compute *dynamic* permissions that may flow in arbitrary (sound) ways.

Putting together both phases, we prove that the imperative program is equivalent to the dataflow program on all possible inputs and schedules. Furthermore, equivalence with the sequential program also implies liveness and deadlock freedom for the dataflow program, as it enforces that the dataflow program can make progress whenever the sequential program can.

We have implemented this technique in FlowCert, a translation validation system targeting the state-of-the-art CGRA architecture RipTide [Gobieski et al. 2023], which operates via a compiler from LLVM [Lattner and Adve 2004] programs to dataflow programs. Using our tool, we found 8 compiler bugs where the RipTide dataflow compiler generated incorrect dataflow programs. All of these bugs were confirmed by the developers of the RipTide compiler. One of these bugs allows the compiler to emit a dataflow program with data races, which are hard to discover via testing and costly to fix after deployment.

To summarize, our contributions in this work are:

- A novel two-phase translation validation technique to prove that the output dataflow program has equivalent behavior as the input imperative program, capturing both correctness and liveness properties.
- An implementation of our technique, FlowCert[1], targeting the RipTide dataflow compiler.
- An evaluation of FlowCert on the RipTide dataflow compiler, which reveals 8 compilation bugs. Most verification result takes around 10 seconds with a maximum of about 30 seconds.

[1]Source code is available in an anonymized GitHub repository [Anonymous Authors 2024]

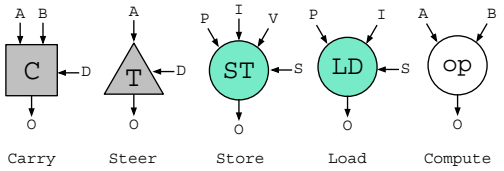


Fig. 1. Examples of operators. `Carry` and `Steer` manage control flow, `Store` and `Load` manipulate the main memory, and op $\in \{+, *, <, \dots\}$ executes pure arithmetic operations.

2 PRELIMINARIES: DATAFLOW PROGRAMS, CGRAS, AND RIPTIDE

Dataflow programming [Johnston et al. 2004] is an old idea dating back to the 1960s and 1970s. Early foundational works include the dataflow architecture by Dennis et al. [Dennis and Misunas 1974], as well as theoretical analysis of models of dataflow programs such as computation graphs [Karp and Miller 1966] and Kahn process networks [Gilles 1974]. The main motivation in these early works is to achieve a greater degree of instruction-level parallelism [Ackerman 1982].

In the intervening years, the community has explored a wide variety of dataflow architectures [Arvind and Culler 1986; Yazdanpanah et al. 2014]. Among these dataflow architectures, *coarse-grained reconfigurable arrays* (CGRAs) [Liu et al. 2019] stand out due to their promising power efficiency: recent work [Gobieski et al. 2021, 2023] is able to support general programmability, while reducing the energy cost by orders of magnitude compared to a von Neumann core.

To utilize the efficiency and parallelism in CGRAs, the predominant general-purpose way to program CGRAs is to compile a sequential, imperative program into a dataflow graph via a dataflow compiler, as seen in CGRA projects such as PipeRench [Goldstein et al. 2000], WaveScalar [Swanson et al. 2003], and more recently, RipTide [Gobieski et al. 2023] and Pipestitch [Serafin et al. 2023].

In this work, we focus on the approach taken by RipTide [Gobieski et al. 2023], which has a dataflow compiler from arbitrary LLVM functions to dataflow programs. We briefly introduce LLVM and dataflow programs below.

LLVM IR. LLVM [Lattner and Adve 2004] is a compiler framework and features an intermediate representation called LLVM IR. Compiler writers can translate a higher-level language such as C or Rust into LLVM IR, and then the LLVM framework can perform optimizations on LLVM IR and produce machine code in various target architectures. For an informal description of the semantics of the LLVM IR, we refer the reader to the official LLVM IR manual [LLVM 2024b]. In this work, we use the subset of LLVM IR supported by the RipTide compiler, including basic control-flow, integer arithmetic, and memory operations. Function calls and floating-point operations are not supported.

Dataflow programs. A *dataflow program* is a Turing-complete representation of programs. It is represented as a dataflow graph in which the nodes are called *operators* and edges are called *channels*. Semantically, operators can be thought of as stateful processes running in parallel, which communicate through channels, or FIFO queues of messages. Repeatedly, each operator: waits to dequeue (a subset of) input channels to get input values; performs a local computation, optionally changing its local state or global memory; and pushes output values to (a subset of) output channels. When an iteration of this loop is done, we say that the operator has executed or *fired*. Since different CGRAs have different strategies for scheduling operators [Liu et al. 2019], we conservatively use asynchrony to model all possible schedules.

In Figure 1, we show the five most important types of operators in RipTide [Gobieski et al. 2023][2]. In Figure 1, reading from left to right, we first have the control flow operators, *carry* (C) and *steer* (T). The carry operator is used to support loop variables. It has two local states: in the initial state, it waits for a value from A, outputs it to O, and transitions to the loop state. In the loop state, it waits for values from B and D (decider); if D is true, it outputs O = B; otherwise it discards B and transitions to the initial state.

The steer operator is used for conditional execution. It waits for values from A and D. If D is true, it outputs O = A; otherwise, it discards the value and outputs nothing.

To utilize the global memory, we have the *load* (LD) and *store* (ST) operators. The load operator waits for values from P (base), I (offset), and an optional signal S for memory ordering; reads the memory location P[I]; and outputs the read value to O. Similarly, the store operator waits for values from P (base), I (offset), V (value), and an optional signal S for memory ordering; stores P[I] = V in the global memory; and optionally outputs a finish signal to O.

All other *compute* operators (e.g., op $\in \{+, *, <\}$) wait for values from input channels, perform the computation, and then output the result to output channels.

To compile an LLVM program to a dataflow program, the RipTide compiler first maps each LLVM instruction to its corresponding dataflow operator, and then it enforces the expected control-flow and memory ordering semantics of the original program. Using a control dependency analysis, it inserts steer operators to selectively enable/disable operators depending on branch/loop conditions; Using a memory ordering analysis, it inserts dataflow dependencies between load/store operators to prevent data races.

These analyses are quite complex and use various optimizations to allow more parallelism than the original LLVM program. As shown in our evaluation in Section 6, this process can easily have bugs. Hence, in our work, we use translation validation to certify the correctness of dataflow compilation.

3 OVERVIEW AND AN EXAMPLE

FlowCert performs translation validation on the RipTide dataflow compiler, which compiles LLVM programs to dataflow programs. Given the input program and the compiled output program (along with some hints generated by the compiler), FlowCert performs two checks: first, a *simulation* check, which verifies that the LLVM program is equivalent to the dataflow program on a restricted, canonical schedule of dataflow operators; and second, a *confluence* check, which proves that the choice of the canonical schedule for the dataflow program is general and all other schedules reach the same final state as the canonical schedule.

If a pair of input/output programs passes both checks, the two programs are equivalent (in the sense of Definition 2). Equivalence implies both *safety* — the dataflow program returns the correct values — and *liveness* — the dataflow program always terminates correctly, and does not contain communication-related deadlocks.

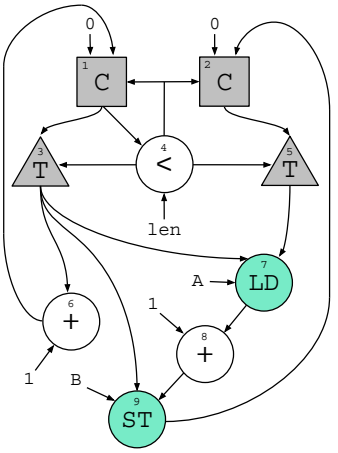
Example. In Figure 2, we have an example input and output program from the dataflow compiler. The input LLVM function `@test` in Figure 2a contains a single loop with loop header block `%header` and loop body block `%body`. The loop increments variable `%i` from `0` to `%len`, and at each iteration, updates `%B[%i] = %A[%i] + 1`. The LLVM function `@test` is compiled to the dataflow program in Figure 2b.

In addition to compute and memory operators, the compiler also inserts operators to faithfully implement the sequential semantics of the LLVM program. The steer operators marked with T enforce that the operators corresponding to the loop body only execute when the loop condition is

[2]FlowCert supports all RipTide operators, except for the *Stream* operator, as described in Section 7.

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

(a) Input LLVM function.



(b) Output dataflow program.

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

true. The carry operator 1 is used for the loop induction variable `%i` (line 5). The carry operator 2 is used for a loop variable inserted by the compiler to enforce a memory dependency: since arrays `A` and `B` could overlap, we have an additional dataflow path along operators $9 \longrightarrow 2 \longrightarrow 5 \longrightarrow 7$ in order to enforce that the load in the next iteration waits until the previous store has finished.

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

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

To automatically check that the proposed relation in Figure 4 is indeed a simulation, we perform *symbolic execution* [Baldoni et al. 2018] from all pairs of cut points (except for $\bot$) until they reach another pair of cut points. On the LLVM side, we first symbolically execute cut point 1, which turns into two symbolic branches: reaching `%header` after one loop iteration (cut point 2), or failing the loop condition and reaching `%end` (cut point $\bot$). For LLVM cut point 2, we also have two branches: one reaching cut point 2 again after a loop iteration, the other failing the loop condition and reaching cut point $\bot$.

For the dataflow side, we have a correspondence between a subset of LLVM instructions and a subset of dataflow operators, which is automatically generated by the dataflow compiler. We use this mapping to infer the order in which we fire the dataflow operators, i.e., the canonical schedule.

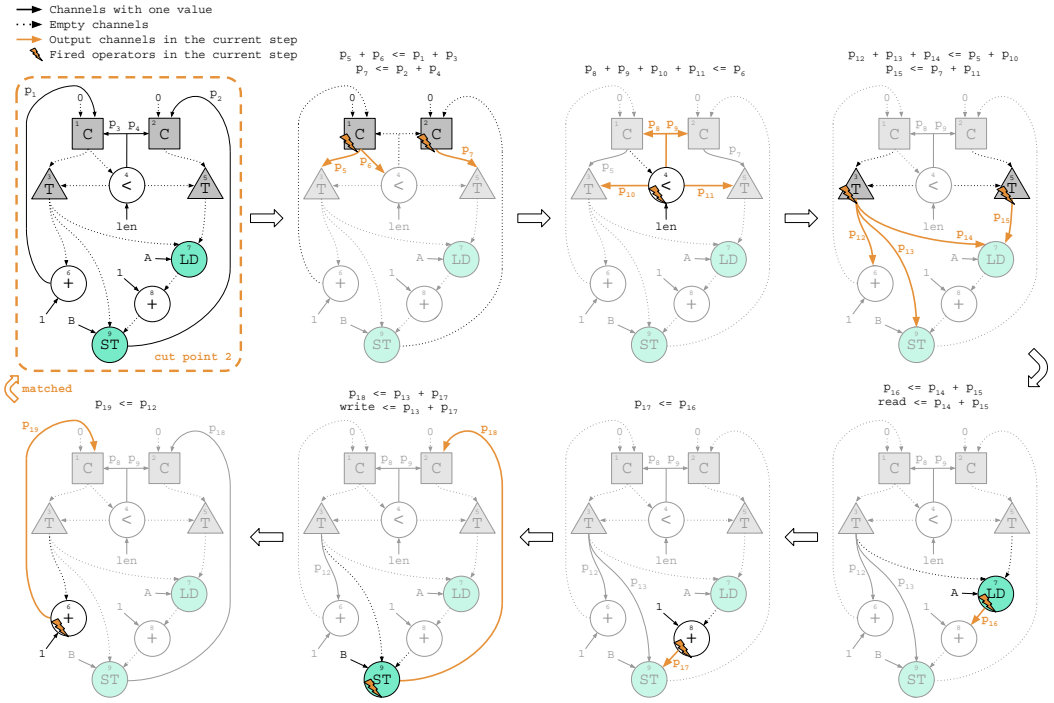


Fig. 3. This plot shows a trace of execution from dataflow cut point 2 (upper left corner). The operators fired in this trace are 1, 2, 4, 3, 5, 7, 8, 9, 6 (which are inferred from the LLVM trace from cut point 2 to cut point 2). At the end of the trace, the configuration matches cut point 2 again. In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

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

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

For example, from LLVM cut point 2, there are two branches reaching cut point 2 and cut point $\perp$ respectively. These two branches have two traces of LLVM instructions (excluding `br`):

- For the first branch to cut point 2: lines 5, 6, 9 - 14;

- For the second branch to cut point $\bot$: lines 5, 6.

For the first branch, the LLVM instructions map to dataflow operators 1, 2, 4, 7, 8, 9, 6 (excluding ones that do not have a corresponding operator; operator 2 corresponds to a `phi` instruction implicitly inserted during compilation for memory ordering). LLVM instructions in the second branch map to dataflow operators 1, 2, and 4. Therefore, for dataflow cut point 2, we execute these two traces of dataflow operators (and also any steer operator that can be executed), which gives us two dataflow configurations. Figure 3 shows the trace of execution of the first branch from cut point 2 to itself. At the end of the execution, we can see that the configuration matches the cut point 2 we started with, except with a different memory state and symbolic variables (e.g. `i + 1` vs. `i` in the channel from operator 6 to 1).

After a branch is matched to a cut point, we check the validity of the correspondence equations at the target cut points (given the assumptions made in the source cut point). For example, for the branch shown in Figure 3, we check that given the source cut point correspondence ($\mathsf{Mem}_{\mathrm{llvm}} = \mathsf{Mem}_{\mathrm{df}} \wedge \texttt{\%i} = \texttt{i}$), the target cut point correspondence holds:

$$\mathsf{Mem}_{\mathrm{llvm}}[\texttt{\%B[\%i]} \mapsto \texttt{\%A[\%i]} + 1] = \mathsf{Mem}_{\mathrm{df}}[\texttt{B[i]} \mapsto \texttt{A[i]} + 1] \wedge \texttt{\%i + 1} = \texttt{i + 1}$$

If this check succeeds for all cut points, we soundly conclude that we have obtained a simulation relation between the LLVM program and the dataflow program on the canonical schedule.

Confluence Check. Once we have established a simulation relation between the LLVM program and a canonical schedule of the dataflow program, we check that the dataflow program is *confluent*: all possible schedules converge to the canonical schedule in the final state (or synchronize with it infinitely often if the program is non-terminating). This confluence result ensures that our simulation result for the canonical schedule is fully general for all other schedules.

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

In general, as defined in Section 5, we allow `write` to split into k tokens of `read`, for a predetermined value of k, and we have one permission token for every memory region, such as the array `A` or `B`. The intuition is that to write to a memory location, an operator needs to have exclusive write ownership of that location and no other operator should have `write` or `read` permissions; while to read a memory location, we allow potentially k parallel reads to happen independently. If all reads are done, and a write needs to occur, we merge all `read` tokens into one `write` token.

These permission tokens are passed between operators via channels, and operators are not allowed to duplicate or create new tokens (i.e., permission tokens are used linearly). We attach permissions to values flowing through the dataflow program, instead of creating separate channels for communicating permission tokens.

To determine the exact assignment and flow of permission tokens, we first attach a permission variable (representing a permission token that we do not know a priori) to each value in the channels of dataflow configurations in each symbolic branch. Figure 3 shows an example of this attachment of permission variables and permission constraints. At the starting state cut point 2, we attach free permission variables p_1, p_2, p_3, p_4 to values in the configuration. After the first step in which operators 1 and 2 fire, these permission variables are consumed by the two carry operators and we generate fresh permission variables p_5, p_6, p_7 for the new output values. We require that the

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

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

If there is a satisfying assignment of the permission variables with permission tokens, then intuitively, the operators in the dataflow program are consistent in memory accesses, and there will not be any data races. In Section 5, we show that this implies that any possible schedule will converge to the canonical schedule we use. For the example in Figure 3, a satisfying assignment is $p_2 = p_7 = p_{15} = p_{16} = p_{17} = p_{18} = \texttt{write}$ with all other variables set to 0. This assignment essentially passes the write permission across the loop of operators $2 \to 5 \to 7 \to 8 \to 9 \to 2$.

4 EQUIVALENCE ON A CANONICAL SCHEDULE VIA CUT-SIMULATION

We detail the first half of our translation validation technique, which proves via *cut-simulation* that the LLVM program is equivalent to the dataflow program when restricted to a canonical schedule of dataflow operators.

Most of our formulation is independent of the source language. Hence, instead of using LLVM directly, we assume some sequential and imperative input language called *Seq* such that 1) Seq is deterministic; 2) Seq has a sequentially consistent memory model (as do dataflow programs); 3) Seq programs operate on fixed-width integer values (as do dataflow programs).

Throughout this section, we fix a dataflow program and a Seq program, and we denote their operational semantics as transition systems $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \to_{\text{df}})$ and $(\mathcal{I}_{\text{seq}}, C_{\text{seq}}, \to_{\text{seq}})$, respectively, where $\mathcal{I}_{\text{df}}$ and $\mathcal{I}_{\text{seq}}$ are the sets of initial configurations, and C_{df} and C_{seq} are the sets of all configurations.

We use $\mathsf{Mem}_{\text{df}}(c_{\text{df}})$ to denote the state of the memory in a dataflow configuration c_{df}, and similarly $\mathsf{Mem}_{\text{seq}}(c_{\text{seq}})$ for the memory in a Seq configuration c_{seq}. We will omit the subscripts when it is clear from the context. We assume that the memory map is modeled in the same way in both semantics, so that we can directly compare them by $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$. Additionally, both the dataflow and Seq programs have input parameters (e.g., the pointer `%A` in Figure 2); thus, we assume a relation $\sim_{\mathcal{I}} \subseteq \mathcal{I}_{\text{df}} \times \mathcal{I}_{\text{seq}}$ on initial configurations that guarantees equal values on all input parameters.

For a transition relation $\to \in \{\to_{\text{df}}, \to_{\text{seq}}\}$, we use $\to^*$ for the reflexive-transitive closure of $\to$, and $\to^+$ for the transitive closure of $\to$. A configuration c is *terminating/final* if there exists no configuration c' such that $c \to c'$. A *trace* is a finite or infinite sequence of configurations $\tau = c_1 \ldots c_i \ldots$ such that $c_i \to c_{i+1}$ for any i. A trace τ is *complete* if τ is infinite or the last configuration of τ is final. We define $\text{Traces}_{\text{df}}$ and $\text{Traces}_{\text{seq}}$ to be the set of all traces in the dataflow and Seq semantics, respectively. For the dataflow semantics, we use $\to^o_{\text{df}}$ to denote the deterministic firing of a particular operator o.

One important tool that we use is symbolic execution, and in the following, we establish some notation for symbolic execution. Let $\mathbf{x}$ be a list of variables $\mathbf{x} = (x_1, \ldots, x_n)$. Let V be the set of all bit-vectors of a predetermined length (for modeling fixed-width integers). We use $\text{Terms}(\mathbf{x})$

to denote the set of *symbolic expressions* constructed from constants in V, variables in $\mathbf{x}$, and any bit-vector operation we could perform in our semantics (e.g., a 32-bit zero bit-vector 0_{32}, or a term like $x_1 + 1_{32}$ are in $\mathsf{Terms}(\mathbf{x})$). For $s \in \{\mathrm{df}, \mathrm{seq}\}$, we use $C_s(\mathbf{x})$ to represent the set of *symbolic configurations with free variables* $\mathbf{x}$ in the semantics s. A symbolic configuration $c(\mathbf{x}) \in C_s(\mathbf{x})$ is similar to a configuration but over symbolic expressions $\mathsf{Terms}(\mathbf{x})$ instead of only V (for example, a symbolic dataflow configuration can have symbolic expressions such as $x_1 + 1_{32}$ in the channels). If we have a substitution $\sigma : \{x_1, \ldots, x_n\} \to \mathsf{Terms}(\mathbf{y})$ from $\mathbf{x}$ to symbolic expressions over some other list of variables $\mathbf{y}$, we denote $c(\sigma) \in C_s(\mathbf{y})$ to be the result of the substitution. We use $c(\mathbf{x}) \to_s c'(\mathbf{x}) \mid \varphi(\mathbf{x})$ to denote symbolic execution from $c(\mathbf{x})$ to $c'(\mathbf{x})$ with path condition $\varphi(\mathbf{x})$.

A dataflow program does not have a notion of a return value, and the only observable state after finishing execution is the modified global memory. Thus we first define when two traces are equivalent based on their memory state.

Definition 1 (Memory-synchronizing traces)**.** Let $\tau_{\mathrm{df}} = c^1_{\mathrm{df}} \ldots c^i_{\mathrm{df}} \ldots$ and $\tau_{\mathrm{seq}} = c^1_{\mathrm{seq}} \ldots c^j_{\mathrm{seq}} \ldots$ be two complete traces starting from c^1_{df} and c^1_{seq}, respectively. The traces τ_{df} and τ_{seq} are *memory-synchronizing* iff $\mathsf{Mem}(c^1_{\mathrm{df}}) = \mathsf{Mem}(c^1_{\mathrm{seq}})$, and

- If τ_{df} is finite, then τ_{seq} is finite and $\mathsf{Mem}(c'_{\mathrm{df}}) = \mathsf{Mem}(c'_{\mathrm{seq}})$, where c'_{df} and c'_{seq} are the final states of τ_{df} and τ_{seq}, respectively.
- If τ_{df} is infinite, then τ_{seq} is infinite, and for any $n \in \mathbb{N}$, there are $i, j \geq n$ such that $\mathsf{Mem}(c^i_{\mathrm{df}}) = \mathsf{Mem}(c^j_{\mathrm{seq}})$.

In other words, two complete dataflow and Seq traces are memory-synchronizing if they are both finite and have the same memory state in the initial and final configurations (i.e., both programs terminate and have the same final memory state); or they are both infinite and they synchronize in the same memory state infinitely often.

Now, we may define *program equivalence*, which describes when two matching initial configurations for dataflow and Seq exhibit the same observable behaviors:

Definition 2 (Program equivalence)**.** Two programs $(\mathcal{I}_{\mathrm{df}}, C_{\mathrm{df}}, \to_{\mathrm{df}})$ and $(\mathcal{I}_{\mathrm{seq}}, C_{\mathrm{seq}}, \to_{\mathrm{seq}})$ are *equivalent* if for any pair of initial states $c_{\mathrm{df}} \sim_{\mathcal{I}} c_{\mathrm{seq}}$ with $\mathsf{Mem}(c_{\mathrm{df}}) = \mathsf{Mem}(c_{\mathrm{seq}})$, if we take any complete dataflow trace $\tau_{\mathrm{df}} = c_{\mathrm{df}} \ldots$ and the unique complete Seq trace $\tau_{\mathrm{seq}} = c_{\mathrm{seq}} \ldots$ (unique since Seq is deterministic), then τ_{df} and τ_{seq} are memory-synchronizing.

Intuitively, the dataflow program and Seq program are equivalent if any pair of complete traces from corresponding initial states are memory-synchronizing; i.e., they either both terminate with the same memory state, or both run forever and synchronize infinitely often.

Our final goal is to achieve Definition 2. We will do so by first verifying a weaker property of equivalence along a particular, deterministic *canonical* schedule of the dataflow program. Then, in Section 5, we show that this canonical schedule suffices to prove equivalence for all possible dataflow schedules.

First, we introduce canonical schedules, which determinize dataflow programs using the information contained in Seq traces:

Definition 3 (Canonical schedule)**.** A *canonical schedule* is a function $\mathsf{Canon} : C_{\mathrm{df}} \times \mathsf{Traces}_{\mathrm{seq}} \to \mathsf{Traces}_{\mathrm{df}}$ that selects a dataflow trace using a Seq trace, such that

- For any $c_{\mathrm{df}} \in C_{\mathrm{df}}$ and any non-empty Seq trace τ_{seq}, $\mathsf{Canon}(c_{\mathrm{df}}, \tau_{\mathrm{seq}})$ is a dataflow trace starting in c_{df}.
- If τ_{seq} is finite, then $\mathsf{Canon}(c_{\mathrm{df}}, \tau_{\mathrm{seq}})$ is finite.
- For any $c_{\mathrm{df}} \in C_{\mathrm{df}}$, if τ'_{seq} is a prefix of τ_{seq}, then $\mathsf{Canon}(c_{\mathrm{df}}, \tau'_{\mathrm{seq}})$ is a prefix of $\mathsf{Canon}(c_{\mathrm{df}}, \tau_{\mathrm{seq}})$.

We then specialize program equivalence (Definition 2) by restricting our attention to those canonical dataflow schedules:

Definition 4 (Program equivalence on a canonical schedule)**.** Let Canon be a canonical schedule. Two programs $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \to_{\text{df}})$ and $(\mathcal{I}_{\text{seq}}, C_{\text{seq}}, \to_{\text{seq}})$ are *equivalent on the canonical schedule* Canon if for any pair of initial states $c_{\text{df}} \sim_{\mathcal{I}} c_{\text{seq}}$ with $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$, if we take the unique complete Seq trace $\tau_{\text{seq}} = c_{\text{seq}} \ldots$, then $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ is a complete dataflow trace, and $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ and τ_{seq} are memory-synchronizing.

That is, instead of enforcing that all complete dataflow traces are memory-synchronizing to the complete Seq trace, we only require that there exists one such trace selected by some canonical schedule Canon. Later in Section 5, we separately verify the *confluence* property (Definition 7), so that Definition 7 and Definition 4 together imply Definition 2 (Proposition 1).

Canonical Schedules for RipTide. In our case study of verifying compilation from LLVM programs to RipTide dataflow programs, the specific canonical schedule Canon is a heuristic designed for LLVM and the RipTide compiler. We instrument the RipTide compiler to output a partial map $\mathit{hint} : \{I_1, \ldots, I_m\} \rightharpoonup \{o_1, \ldots, o_n\}$, where $I_1, \ldots, I_m$ are LLVM instructions in the LLVM program, and $o_1, \ldots, o_n$ are operators in the dataflow program. We require that the image of *hint* covers all dataflow operators except for steer operators (which are control-flow operators and have no direct correspondence to an LLVM instruction). The canonical schedule $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{llvm}})$ is then inductively constructed following the LLVM instructions executed in τ_{llvm}:

$$\mathsf{Canon}(c_{\text{df}}, \tau_{\text{llvm}}) := \begin{cases} \epsilon & \text{If } \tau_{\text{llvm}} = \epsilon \\ \tau_{\text{df}} \cdot \mathsf{Canon}(c'_{\text{df}}, \tau'_{\text{llvm}}) & \text{If } \tau_{\text{llvm}} = c_{\text{llvm}} \cdot \tau'_{\text{llvm}} \end{cases}$$

where ϵ is the empty trace, $\cdot$ denotes trace concatenation, I is the LLVM instruction executed at c_{llvm}, and $\tau_{\text{df}} = c_{\text{df}} \ldots c'_{\text{df}}$ is a dataflow trace firing $\mathit{hint}(I)$ and then any fireable steer operators (which may be fired in any order, as they all commute). If $\mathit{hint}(I)$ is not defined or not fireable in c_{df}, then $\tau_{\text{df}} = \epsilon$ and $c'_{\text{df}} = c_{\text{df}}$.

In the following sections, we outline the steps to verify that the dataflow and Seq programs satisfy Definition 4. In Section 4.1, we first describe our general method of using a coinductive invariant called cut-simulation that implies the property in Definition 4. In Section 4.2, we show how cut-simulation can be symbolically represented by cut points. In Section 4.3, we present the algorithm to check that a set of cut points does form a cut-simulation. Finally in Section 4.4, we describe our heuristics to infer cut points for the compilation from LLVM to dataflow programs.

4.1 Proving Equivalence on the Canonical Schedule via Cut-Simulation

Traces from $\mathcal{I}_{\text{df}}$ and $\mathcal{I}_{\text{seq}}$ are unbounded in general, so to prove the equivalence on a canonical schedule (Definition 4), we need to verify a coinductive invariant about the pairs of traces starting from $(\mathcal{I}_{\text{df}}, \mathcal{I}_{\text{seq}})$, and such an invariant is a *cut-simulation* relation [Kasampalis et al. 2021].

Definition 5 (Cut-simulation)**.** Let $\mathcal{R} \subseteq C_{\text{df}} \times C_{\text{seq}}$ be a binary relation. We say that $\mathcal{R}$ is a *cut-simulation* between $(\mathcal{I}_{\text{seq}}, C_{\text{seq}}, \to_{\text{seq}})$ and $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \to_{\text{df}})$ iff for any $(c_{\text{df}}, c_{\text{seq}}) \in \mathcal{R}$, if $c_{\text{seq}} \to_{\text{seq}} c'_{\text{seq}}$ for some c'_{seq}, then there are $(c'_{\text{df}}, c''_{\text{seq}}) \in \mathcal{R}$ such that $c'_{\text{seq}} \to^*_{\text{seq}} c''_{\text{seq}}$ and $c_{\text{df}} \to^+_{\text{df}} c'_{\text{df}}$.

Intuitively, a cut-simulation states that if the two transition systems are synchronized in $\mathcal{R}$, then if we can take a step on the Seq side, we can step the two transition systems further to re-synchronize them in $\mathcal{R}$.

Note that in our setting, we are verifying that the dataflow program simulates the Seq program (i.e., the target program simulates the source program), instead of the other way around which

is common in compiler verification work. This is because our target dataflow program has more nondeterministic behavior than the source Seq program, whereas in many other cases, the source program has more behavior (such as undefined behavior in C) than the concrete target program.

To satisfy our desired equivalence properties, we look for cut-simulations that are *memory-synchronizing*:

Definition 6. A binary relation $\mathcal{R} \subseteq C_{\text{df}} \times C_{\text{seq}}$ is *memory-synchronizing* iff

- For any $(c_{\text{df}}, c_{\text{seq}}) \in \mathcal{R}$, $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$.
- For any initial configurations $c_{\text{df}} \sim_{\mathcal{I}} c_{\text{seq}}$ with $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$, $(c_{\text{df}}, c_{\text{seq}}) \in \mathcal{R}$.
- For any $(c_{\text{df}}, c_{\text{seq}})$ with $c_{\text{df}}, c_{\text{seq}}$ final and $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$, $(c_{\text{df}}, c_{\text{seq}}) \in \mathcal{R}$.

The second and third conditions enforce that $\mathcal{R}$ should include all pairs of corresponding initial states, as well as final states; thus $\mathcal{R}$ covers all complete traces.

If there exists a memory-synchronizing cut-simulation $\mathcal{R}$, then we can conclude via a coinductive argument that the two programs are equivalent on a canonical schedule (Definition 4).

4.2 Describing a Cut-Simulation via Cut Points

To finitely describe a proposed cut-simulation, we use a list of pairs of dataflow and Seq *cut points*. A cut point is a pair $P = (c_{\text{df}}(\mathbf{x}), c_{\text{seq}}(\mathbf{y})) \mid \varphi(\mathbf{x}, \mathbf{y})$ of symbolic dataflow and Seq configurations constrained by the correspondence condition $\varphi(\mathbf{x}, \mathbf{y})$. In our case, the condition $\varphi(\mathbf{x}, \mathbf{y})$ is a conjunction of equalities between variables in $\mathbf{x}$ and $\mathbf{y}$.

Semantically, each cut point describes the binary relation $\mathcal{R}(P) := \{(c_{\text{df}}(\mathbf{x}), c_{\text{seq}}(\mathbf{y})) \in C_{\text{df}} \times C_{\text{seq}} \mid \mathbf{x}, \mathbf{y} \in V, \varphi(\mathbf{x}, \mathbf{y})\}$. For a list of n cut points

$$\{P^i = (c^i_{\text{df}}(\mathbf{x^i}), c^i_{\text{seq}}(\mathbf{y^i})) \mid \varphi^i(\mathbf{x^i}, \mathbf{y^i})\}_{i=1}^n,$$

the relation they describe is the union of the subrelation each describes: $\mathcal{R}(P^1, \ldots, P^n) := \bigcup_{i=1}^n \mathcal{R}(P^i)$.

To ensure that the cut-simulation we propose satisfies the three conditions in Definition 6, we only construct cut points such that:

- For any cut point P^i, the correspondence constraint $\varphi^i(\mathbf{x}^i, \mathbf{y}^i)$ implies equal memory $\mathsf{Mem}(c^i_{\text{df}}(\mathbf{x}^i)) = \mathsf{Mem}(c^i_{\text{seq}}(\mathbf{y}^i))$.
- The first cut point exactly represents the pairs of equivalent, initial configurations with equal memory; i.e., $\mathcal{R}(P^1) = \{(c_{\text{df}}, c_{\text{seq}}) \in \sim_{\mathcal{I}} \mid \mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})\}$.
- The last cut point P^n exactly represents the pairs of final configurations with equal memory; i.e., $\mathcal{R}(P^n) = \{(c_{\text{df}}, c_{\text{seq}}) \in C_{\text{df}} \times C_{\text{seq}} \mid c_{\text{df}}, c_{\text{seq}} \text{ final}, \mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})\}$.

Then $\mathcal{R}(P^1, \ldots, P^n)$ satisfies Definition 6.

4.3 Checking Cut-Simulation

Now that we have a way to symbolically describe a relation $\mathcal{R}(P^1, \ldots, P^n)$ with cut points satisfying Definition 6, we need to check that it is indeed a cut-simulation.

To achieve this, we perform symbolic execution from each cut point and try to reach another cut point. Suppose we are checking cut point $P^i = (c^i_{\text{df}}(\mathbf{x^i}), c^i_{\text{seq}}(\mathbf{y^i})) \mid \varphi^i(\mathbf{x^i}, \mathbf{y^i})$. We perform the following three steps.

Step 1. Symbolic execution of Seq cut points. We perform symbolic execution from $c^i_{\text{seq}}(\mathbf{y^i})$ and get a list of k_i symbolic branches with path conditions denoted by ψ:

$$c^i_{\text{seq}}(\mathbf{y^i}) \to^+_{\text{seq}} c^{i,1}_{\text{seq}}(\mathbf{y^i}) \mid \psi_{\text{seq},1}(\mathbf{y^i})$$
$$\ldots$$
$$c^i_{\text{seq}}(\mathbf{y^i}) \to^+_{\text{seq}} c^{i,k_i}_{\text{seq}}(\mathbf{y^i}) \mid \psi_{\text{seq},k_i}(\mathbf{y^i})$$

where each right hand side $c^{i,l}_{\text{seq}}(\mathbf{y^i})$ for $l \in \{1, \ldots, k_i\}$ is an instance $c^j_{\text{seq}}(\sigma)$ of some Seq cut point $c^j_{\text{seq}}(\mathbf{y^j})$ with substitution $\sigma : \mathbf{y^j} \to \text{Terms}(\mathbf{y^i})$. That is, at the end of each branch, the configuration is matched to some cut point.

Note that since the initial symbolic configuration $c^i_{\text{seq}}(\mathbf{y^i})$ does not have any constraints, the final path conditions should cover all cases of $\mathbf{y^i}$; in other words, $\bigvee_{l=1}^{k_i} \psi_{\text{seq},l}(\mathbf{y^i})$ is valid.

Step 2. Symbolic execution of dataflow cut points on the canonical schedule. For the dataflow cut point $c^i_{\text{df}}(\mathbf{x^i})$, since the dataflow semantics is nondeterministic, the choice of operators to fire is unclear. Therefore, we follow a canonical schedule Canon (Definition 3) to replicate the execution of Seq branches on the dataflow side.

For each Seq branch $l \in \{1, \ldots, k_i\}$, we have a Seq trace $\tau_{\text{seq}} = c^i_{\text{seq}}(\mathbf{y}^i) \ldots c^{i,l}_{\text{seq}}(\mathbf{y}^i)$. By mapping this trace through the canonical schedule $\tau_{\text{df}} := \text{Canon}(c^i_{\text{df}}(\mathbf{x^i}), \tau_{\text{seq}})$, we get a list of dataflow operators $(o^l_1, \ldots, o^l_{n_l})$ fired in τ_{df}. Then we symbolically execute the dataflow cut point $c^i_{\text{df}}(\mathbf{x^i})$ on the schedule $(o^l_1, \ldots, o^l_{n_l})$ with the path condition $\psi_{\text{df},l}(\mathbf{x^i}) := \exists \mathbf{y^i} \varphi^i(\mathbf{x^i}, \mathbf{y^i}) \wedge \psi_{\text{seq},l}(\mathbf{y^i})$:

$$c^i_{\text{df}}(\mathbf{x^i}) \to^{o^l_1}_{\text{df}} \cdots \to^{o^l_{n_l}}_{\text{df}} c^{i,l}_{\text{df}}(\mathbf{x^i}) \mid \psi_{\text{df},l}(\mathbf{x^i})$$

The new path condition $\psi_{\text{df},l}(\mathbf{x^i})$ amounts to replacing variables in the Seq path condition $\psi_{\text{seq},l}(\mathbf{y^i})$ with corresponding variables in the correspondence constraint $\varphi^i(\mathbf{x^i}, \mathbf{y^i})$. With the additional path condition imposed, the symbolic execution is not likely to branch. If it does, the cut-simulation check will fail; in which case, a more precise heuristic is needed for the canonical schedule.

By doing this for each Seq branch, we get a list of dataflow branches:

$$\begin{aligned} c^i_{\text{df}}(\mathbf{x^i}) &\to^+_{\text{df}} c^{i,1}_{\text{df}}(\mathbf{x^i}) \mid \psi_{\text{df},1}(\mathbf{x^i}) \\ &\ldots \\ c^i_{\text{df}}(\mathbf{x^i}) &\to^+_{\text{df}} c^{i,k_i}_{\text{df}}(\mathbf{x^i}) \mid \psi_{\text{df},k_i}(\mathbf{x^i}) \end{aligned} \tag{1}$$

In our case, these branches should cover all possible values of $\mathbf{x}^i$ (i.e., $\bigvee_{l=1}^{k_i} \psi_{\text{df},l}(\mathbf{x^i})$ is valid), since in the path condition $\psi_{\text{df},l}(\mathbf{x^i})$, the correspondence constraint $\varphi^i(\mathbf{x^i}, \mathbf{y^i})$ only contains a conjunction of equalities between variables in $\mathbf{x^i}$ and $\mathbf{y^i}$, and the union of all Seq path conditions $\bigvee_{l=1}^{k_i} \psi_{\text{seq},l}(\mathbf{y^i})$ is valid. In general, however, if the correspondence constraint $\varphi^i(\mathbf{x^i}, \mathbf{y^i})$ is more complex (e.g. containing formulas such as $x_1 = 2y_2$), we may need to additionally verify that $\bigvee_{l=1}^{k_i} \psi_{\text{df},l}(\mathbf{x^i})$ is valid to ensure that all possible concrete configurations represented by $c^i_{\text{df}}(\mathbf{x^i})$ is covered in the symbolic execution.

Step 3. Cut point subsumption. Finally, we check that each pair of dataflow and Seq branches $(c^{i,l}_{\text{df}}(\mathbf{x^i}), c^{i,l}_{\text{seq}}(\mathbf{y^i}))$ is contained in some target cut point $P^j = (c^j_{\text{df}}(\mathbf{x^j}), c^j_{\text{seq}}(\mathbf{y^j})) \mid \varphi^j(\mathbf{x^j}, \mathbf{y^j})$:

$$\{(c^{i,l}_{\text{df}}(\mathbf{x^i}), c^{i,l}_{\text{seq}}(\mathbf{y^i})) \mid \varphi^i(\mathbf{x^i}, \mathbf{y^i}) \wedge \psi_{\text{df},l}(\mathbf{x^i}) \wedge \psi_{\text{seq},l}(\mathbf{y^i})\} \subseteq \mathcal{R}(P^j) \tag{2}$$

or equivalently, there are substitutions $\sigma^{i,l}_{\text{df}} : \mathbf{x^j} \to \text{Terms}(\mathbf{x^i})$, $\sigma^{i,l}_{\text{seq}} : \mathbf{y^j} \to \text{Terms}(\mathbf{y^i})$ such that the following conditions hold:

- The dataflow branch is an instance of the target dataflow cut point: $c^{i,l}_{\text{df}}(\mathbf{x^i}) = c^j_{\text{df}}(\sigma^{i,l}_{\text{df}})$.
- The Seq branch is an instance of the target Seq cut point: $c^{i,l}_{\text{seq}}(\mathbf{y^i}) = c^j_{\text{seq}}(\sigma^{i,l}_{\text{seq}})$.
- The correspondence condition at the target cut point is satisfied given the source cut point correspondence and the path conditions:

$$\varphi^i(\mathbf{x^i}, \mathbf{y^i}) \wedge \psi_{\text{df},l}(\mathbf{x^i}) \wedge \psi_{\text{seq},l}(\mathbf{y^i}) \to \varphi^j(\sigma^{i,l}_{\text{df}}, \sigma^{i,l}_{\text{seq}})$$

If the steps in this section are passed for all cut points i, we can conclude that $\mathcal{R}(P^1, \ldots, P^n)$ is indeed a cut-simulation.

4.4 Heuristics to Infer Cut Points

In this section, we describe our heuristics to infer cut points specific to LLVM and dataflow programs. In general, program equivalence for Turing-complete models like LLVM and dataflow programs is undecidable, so our best hope is to have good heuristics to infer the cut points for the specific input/output programs of the compiler.

The placement of cut points is important for the symbolic execution to terminate. It is in theory correct (for terminating dataflow and LLVM programs) to have exactly two cut points: one at the initial configurations, and one for all final configurations. But the symbolic execution will try to simulate all possible traces together, which are unbounded, and the check may not terminate. To ensure termination, on the LLVM side, we place a cut point at the entry of the function (i.e., the initial configuration), the back edge of each loop header block, and the exit program point.

For the dataflow program, we need to place the first cut point at the initial configuration of the dataflow program, and the last cut point at the final configuration. For loops, ideally, we want to place the cut points at the corresponding "program points" at loop headers. However, due to the lack of control-flow in the dataflow program, the form of the corresponding dataflow configurations at these "program points" is hard to infer statically. Instead of generating the dataflow cut points ahead of time, we dynamically infer them during the symbolic execution of dataflow branches. For example, after mirroring the LLVM branch execution in a dataflow branch such as in Equation (1), if the dataflow cut point corresponding to the LLVM cut point has not been inferred yet, we generalize the symbolic configuration on the RHS of the dataflow branch Equation (1) to a dataflow cut point by replacing symbolic expressions with fresh variables. The correspondence condition is then inferred using compiler hints to match fresh variables in the new dataflow cut point to an LLVM variable defined by some LLVM instruction.

Note that we do not need these heuristics to be sound. As long as the cut points for the initial and final dataflow/LLVM configurations are generated correctly and the cut-simulation check in Section 4.3 succeeds, the soundness of the cut-simulation follows. While we do not prove that these heuristics are complete, they work well in practice and cover all test cases in our evaluation (Section 6).

If the heuristics generate incorrect cut points, the simulation check would still terminate but fail. To see this, on the LLVM side, our placement of the cut points guarantees termination because we place one at each loop header. On the dataflow side, since the execution is mirroring the LLVM execution and no additional search is done, the algorithm would terminate but may fail to match the resulting dataflow configuration against the cut point.

5 CONFLUENCE CHECKING USING LINEAR PERMISSION TOKENS

After checking the equivalence between the LLVM and dataflow programs on a canonical dataflow schedule, we still need to prove that this canonical schedule is general. Indeed, since the dataflow program is asynchronous and nondeterministic, other choices of schedule may lead to data races and inconsistent final states.

Therefore, in addition to checking the equivalence on a canonical schedule, we want to prove a *confluence* property on the dataflow program:

Definition 7. We say $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ is *(ground) confluent* iff for any initial configuration $c \in \mathcal{I}_{\text{df}}$, if $c \rightarrow^*_{\text{df}} c_1$ and $c \rightarrow^*_{\text{df}} c_2$, then there is $c' \in C_{\text{df}}$ with $c_1 \rightarrow^*_{\text{df}} c'$ and $c_2 \rightarrow^*_{\text{df}} c'$.

If the pair of dataflow and LLVM programs satisfies both Definition 4 and Definition 7, then we can conclude the desired equivalence property of Definition 2.

Proposition 1. If $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ and $(\mathcal{I}_{\text{llvm}}, C_{\text{llvm}}, \rightarrow_{\text{llvm}})$ are equivalent on a canonical schedule (Definition 4) and $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ is confluent, then $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ and $(\mathcal{I}_{\text{llvm}}, C_{\text{llvm}}, \rightarrow_{\text{llvm}})$ are equivalent (Definition 2).

Proof Sketch. Assume $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ and $(\mathcal{I}_{\text{llvm}}, C_{\text{llvm}}, \rightarrow_{\text{llvm}})$ are equivalent on some canonical schedule Canon and $(\mathcal{I}_{\text{df}}, C_{\text{df}}, \rightarrow_{\text{df}})$ is confluent. Let $c_{\text{df}} \sim_{\mathcal{I}} c_{\text{seq}}$ be any pair of corresponding initial states with $\mathsf{Mem}(c_{\text{df}}) = \mathsf{Mem}(c_{\text{seq}})$, and let $\tau_{\text{seq}} = c_{\text{seq}} \ldots$ be the unique complete trace. Let $\tau'_{\text{df}} = c_{\text{df}} \ldots$ be any complete trace from c_{df}. We need to show that τ'_{df} and τ_{seq} are memory-synchronizing.

By equivalence on Canon, we know that $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ is a complete trace and $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ and τ_{seq} are memory-synchronizing. We have two cases:

- If $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ is finite, then by confluence, τ'_{df} is finite and has the same final state as $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ (since τ'_{df} is complete).
- If $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ is infinite, then by confluence, $\mathsf{Canon}(c_{\text{df}}, \tau_{\text{seq}})$ and τ'_{df} synchronize infinitely often.

Therefore τ'_{df} and τ_{seq} are memory-synchronizing. □

In this section, we present a technique using what we call *linear permission tokens* to ensure that the dataflow program is confluent. Linear permission tokens are, in essence, a dynamic variant of fractional permissions [Boyland 2013]. In our case, the exact placement of permission tokens is resolved dynamically via symbolic execution.

We associate each value flowing through a dataflow program with a permission token in the form of `write` l, `read` l, or any formal sum of them, for any memory location l. These tokens are used by dataflow operators in a linear fashion: they cannot be replicated or generated, and they are disjoint in the initial state. To support independent parallel reads and enforce that writes must be exclusive, a `write` token can be split into k many `read` tokens (for a predetermined parameter k), and inversely, k of such `read` tokens can be merged to get back a `write` token.

We show in this section that if there is a consistent assignment of these linear permission tokens to values, then we can conclude that the execution of dataflow program will be confluent. This is done in two steps:

- In Section 5.1, we prove that the technique works in a bounded setting for a concrete trace τ: if τ has a consistent assignment of permission tokens, τ' is another trace from the same initial configuration, and τ' is bounded by τ (meaning that for each operator o, the number of times o fired in τ' is less than or equal to the number of times o fired in τ), then there is a valid trace from the final configuration of τ' to the final configuration of τ; i.e., τ' will converge to the final configuration of τ.
- In Section 5.2, we explain how this bounded confluence check can be extended to a full confluence check for the entire dataflow program by using the cut points that we have placed for the dataflow program in Section 4.

Finally in Section 5.3, we show that even though we assume the channels in a dataflow program are unbounded, the cut-simulation and confluence checks allow us to extend the verification results to a bounded-channel model of dataflow programs.

5.1 Confluence on Bounded Traces

In this section, we show that if we have a finite trace $c_1 \rightarrow_{\text{df}}^{o_1} \cdots \rightarrow_{\text{df}}^{o_n} c_{n+1}$ (where $o_1, \ldots, o_n$ are the operators fired in each step) with a consistent assignment of permission tokens to each value

in each configuration, then for any other trace $c_1 \xrightarrow{o'_1}_{\text{df}} \cdots \xrightarrow{o'_m}_{\text{df}} c_{m+1}$ (not necessarily with a consistent assignment of permission tokens) such that the multiset $\{o'_1, \ldots, o'_m\} \subseteq \{o_1, \ldots, o_n\}$, we have $c_{m+1} \rightarrow^*_{\text{df}} c_{n+1}$.

Let us first describe the permission tokens and their assignment to values. The permission tokens are drawn from the *permission algebra* defined below.

Definition 8 (Permission Algebra)**.** Let L be a set of memory locations and k be a positive integer. The *permission algebra* is a partial algebraic structure $(P(L,k), 0, +, \leq)$ where

- $P(L,k)$ is the set of permissions such that $(P(L,k), 0, +)$ forms a partially commutative monoid. It consists of formal sums $\texttt{read}\ l_1 + \cdots + \texttt{read}\ l_n$, with $l_1, \ldots, l_n \in L$, modulo the commutativity and associativity of $+$. We require that for a location l, at most k copies of $\texttt{read}\ l$ are present in the formal sum. We denote $N(l,p)$ to be the number of occurrences of $\texttt{read}\ l$ in p. We call $p+q$ the *(disjoint) sum* of $p, q \in P(L,k)$. In particular, for any $l \in L$, $p+q$ is defined iff for all l, $N(l,p) + N(l,q) \leq k$.
  We also denote
  $$\texttt{write}\ l := \underbrace{\texttt{read}\ l + \cdots + \texttt{read}\ l}_{k \text{ copies}}$$
- $P(L,k)$ is partially ordered by $\leq$ with $p \leq q$ (reads q *contains* p) iff for any l, the number of occurrences of $\texttt{read}\ l$ in p is less than or equal to that in q.

Now we define when two or more permissions are considered *disjoint*, which is crucial for enforcing linearity.

Definition 9. We say that the permissions $p_1, \ldots, p_n \in P(L,k)$ are *disjoint* (write *disjoint*$(p_1, \ldots, p_n)$) iff for any $l \in L$ and $i \neq j$, $N(l,p_i) + N(l,p_j) \leq k$.

For example, in $P(\{\texttt{A}, \texttt{B}\}, 2)$, we have that
$$\texttt{read B} + \texttt{read A} + \texttt{read A} = \texttt{read B} + \texttt{write A},$$
while $\texttt{read A} + \texttt{write A}$ is not defined (i.e., `read A` and `write A` are not disjoint).

Intuitively, the reason to allow a `write` to split into k copies of `read` is to allow the following two valid scenarios:

- A store operator can write to a memory location if it has exclusive write permission to the location (i.e., no other operator has a `write` or `read` permission to that memory location);
- A load operator can read from a memory location if it has a `read` permission and all other currently existing permissions in the system are also `read`.

Furthermore, once (at most k) parallel reads are performed, a store operator can perform a memory write by reclaiming all existing `read` permissions and merging them into a `write`; hence we define `write` as the sum of k copies of `read`.

Now let us define the assignment of permission tokens to values in a configuration. Let E denote the set of channel names. For a configuration $c \in C_{\text{df}}$ and a channel $e \in E$, let $\text{Channel}(c,e)$ denote the state of the channel e in c, which is a string over V representing the values in the channel e.

Definition 10 (Permission Augmentation)**.** Let $c \in C_{\text{df}}$ be a dataflow configuration. A *permission augmentation* of c is a partial map $t : E \times \mathbb{N} \rightharpoonup P(L,k)$ such that for any $e \in E$, $t(e,n)$ is defined iff $n \in \{1, \ldots, |\text{Channel}(c,e)|\}$. We call (c,t) a *permission-augmented configuration.*

Intuitively, $t(e,n)$ is the permission we attach to the n-th value in the channel e. Permission augmentations defined above can assign arbitrary permissions to values. However, to ensure race-freedom and confluence, we need to restrict them so that, for example, two operators cannot share

a `write` token. In the following definitions, we define when a configuration, a transition, and a trace are *consistently augmented with permissions*, meaning that permissions are used linearly.

Definition 11 (Consistent Augmentation)**.** We call (c, t) a *consistent (permission-augmented) configuration* if the permissions in the image of t are disjoint.

Definition 12 (Consistent Transition)**.** Let $c \to^{o}_{\text{df}} c'$ be a transition. Let t, t' be two consistent permission augmentations for c, c', respectively. Let $p_1, \ldots, p_n$ be the permissions in t attached to input values of o and $q_1, \ldots, q_m$ be the permissions in t' attached to output values of o. We say $(c, t) \to^{o}_{\text{df}} (c', t')$ is a *consistent (permission-augmented) transition* iff:

- (c, t) and (c', t') are consistent.
- $t(e, n) = t'(e, n)$ for all channel e and position n except for those changed by o.
- (Linearity) $q_1 + \cdots + q_m \leq p_1 + \cdots + p_n$.
- (Load Permission) If o is a load operator on a memory location $l \in L$, then $\texttt{read}\ l \leq p_1 + \cdots + p_n$.
- (Store Permission) If o is a store operator on a memory location $l \in L$, then $\texttt{write}\ l \leq p_1 + \cdots + p_n$.

Definition 13 (Consistent Trace)**.** Let $c_1 \to_{\text{df}} \cdots \to_{\text{df}} c_n$ be a trace. Let $t_1, \ldots, t_n$ be permission augmentations to $c_1, \ldots, c_n$, respectively. We say that $(c_1, t_1) \to_{\text{df}} \cdots \to_{\text{df}} (c_n, t_n)$ is a *consistent (permission-augmented) trace* iff for all $i \in \{1, \ldots, n\}$, $(c_i, t_i) \to_{\text{df}} (c_{i+1}, t_{i+1})$ is consistent.

We now show some facts about a consistently permission-augmented transition or trace, leading to the final Theorem 3. First, we show that two consistent steps commute.

Lemma 1. *Let $(c_1, t_1) \to^{o_1}_{\text{df}} (c_2, t_2) \to^{o_2}_{\text{df}} (c_3, t_3)$ be a consistent trace. Suppose $c_1 \to^{o_2}_{\text{df}} c_2'$ for some configuration c_2'. Then $c_2' \to^{o_1}_{\text{df}} c_3$ and there is a permission augmentation t_2' such that $(c_1, t_1) \to^{o_2}_{\text{df}} (c_2', t_2') \to^{o_1}_{\text{df}} (c_3, t_3)$ is consistent.*

Proof Sketch. If one of o_1 and o_2 is neither load nor store, or if both of them are loads, they can trivially commute with the same result. Otherwise, one of o_1 and o_2 is a store operator (without loss of generality, assume o_1 is a store). For $i \in \{1, 2\}$, let l_i be the memory location accessed by o_i and p_i be the sum of o_i's input permissions. Since they can both fire at c_1, they should have suitable permissions: $\texttt{write}\ l_1 \leq p_1$ and $\texttt{read}\ l_2 \leq p_2$. Since p_1 and p_2 are disjoint, we have $l_1 \neq l_2$ Therefore, they are accessing different memory locations, and should give the same result no matter in which order we execute them. □

Inductively applying this lemma, we can swap the execution of an operator all the way to the beginning if it can fire and has not fired yet.

Lemma 2. *Let $(c_1, t_1) \to^{o_1}_{\text{df}} \cdots \to^{o_n}_{\text{df}} (c_{n+1}, t_{n+1})$ be a consistent trace. Suppose o_n is not in the set $\{o_1, \ldots, o_{n-1}\}$ and $c_1 \to^{o_n}_{\text{df}} c_2'$ for some c_2', then $c_1 \to^{o_n}_{\text{df}} c_2' \to^{o_1}_{\text{df}} \cdots \to^{o_{n-1}}_{\text{df}} c_k$ for some c_2' and there is a consistent augmentation for this trace.*

Applying the lemma above inductively, we have the following.

Theorem 3 (Bounded Confluence for Consistent Traces). *Let $(c_1, t_1) \to^{o_1}_{\text{df}} \cdots \to^{o_n}_{\text{df}} (c_{n+1}, t_{n+1})$ be a consistent trace. Let $c_1 \to^{o_1'}_{\text{df}} \cdots \to^{o_m'}_{\text{df}} c_{m+1}$ be another trace. If we have the* multiset *inclusion $\{o_1', \ldots, o_m'\} \subseteq \{o_1, \ldots, o_n\}$, then $c_{m+1} \to^{*}_{\text{df}} c_{n+1}$.*

Although we have omitted much of the technical details of the proof, we have formalized a machine-checkable proof of Theorem 3 in Verus [Lattuada et al. 2023], a verification language, and the proof can be found in our anonymized GitHub repository [Anonymous Authors 2024].

Practically, with this bounded confluence theorem, we can do bounded model checking for the confluence property: we perform symbolic execution to obtain a trace of symbolic configurations $c_1(\mathbf{x}) \to^{o_1}_{\text{df}} \cdots \to^{o_n}_{\text{df}} c_{n+1}(\mathbf{x})$, and then we can solve for a consistent augmentation $(t_1, \ldots, t_{n+1})$ for this trace using an SMT solver. If there is a solution, then it follows from Theorem 3 that any other trace (firing operators only in the multiset $\{o_1, \ldots, o_n\}$, counting multiplicity) would still converge to $c_{n+1}(\mathbf{x})$ in the end.

Algorithm 1 Full confluence checking algorithm (Section 5.2).

Require: Inputs:
- $c^1_{\text{df}}(\mathbf{x}^1), \ldots, c^n_{\text{df}}(\mathbf{x}^n)$: a list of dataflow cut points;
- $P(L, k)$: a permission algebra.

Ensure: If ConfluenceCheck succeeds, then the dataflow program is confluent.

1: **procedure** ConfluenceCheck
2: $\psi \leftarrow \top$ ▷ Permission constraints to be accumulated.
3: **for** $i \in \{1, \ldots, n\}$ **do** ▷ Initialize permission variables at cut points
4: $t_i \leftarrow$ **empty_map**() ▷ $t_i : E \times \mathbb{N} \rightharpoonup PV$ (Definition 10)
5: **for** each channel e and $m \in \{1, \ldots, |\text{Channel}(c^i_{\text{df}}(\mathbf{x}^i), e)|\}$ **do**
6: $t_i(e, m) \leftarrow$ **fresh_variable**()
7: $\psi \leftarrow \psi \wedge \mathit{disjoint}(t_i(e, m)$ for all $e, m)$ ▷ Definition 11
8: **for** $i \in \{1, \ldots, n\}$ **do**
9: $Q \leftarrow$ **empty_queue**()
10: **enqueue**$(Q, (c^i_{\text{df}}(\mathbf{x}^i), t_i))$
11: **while** Q is non-empty **do**
12: $(c_{\text{df}}(\mathbf{x}^i), t) \leftarrow$ **dequeue**(Q)
13: **for** $(c'_{\text{df}}(\mathbf{x}^i), t') \in$ **step_canonical**$(c_{\text{df}}(\mathbf{x}^i), t)$ **do** ▷ For each symbolic branch
14: ▷ Constraints in Definition 12, where o is the fired operator.
15: $p_{\text{in}} \leftarrow$ sum of o's input permission variables in t
16: $p_{\text{out}} \leftarrow$ sum of o's output permission variables in t'
17: $\psi \leftarrow \psi \wedge (p_{\text{out}} \le p_{\text{in}})$
18: **if** o reads memory location $l \in L$ **then**
19: $\psi \leftarrow \psi \wedge (\texttt{read}\ l \le p_{\text{in}})$
20: **else if** o writes to memory location $l \in L$ **then**
21: $\psi \leftarrow \psi \wedge (\texttt{write}\ l \le p_{\text{in}})$
22: **for** $j \in \{1, \ldots, n\}$ **do**
23: **if** $\exists \sigma : \mathbf{x}^j \to V(\mathbf{x}^i).\ c'_{\text{df}}(\mathbf{x}^i) = c^j_{\text{df}}(\sigma)$ **then** ▷ Matched to a cut point
24: **for** each channel e and $m \in \{1, \ldots, |\text{Channel}(c^j_{\text{df}}(\mathbf{x}^j), e)|\}$ **do**
25: $\psi \leftarrow \psi \wedge (t'(e, m) = t_j(e, m))$
26: **continue** Line 13
27: **enqueue**$(Q, (c'_{\text{df}}(\mathbf{x}^i), t'))$
28: **succeed** iff ψ is satisfiable in $P(L, k)$

5.2 Full Confluence Checking using Cut Points

To extend the bounded confluence results in Section 5.1 to checking confluence for the entire dataflow program with unbounded traces, we need to finitely describe and check a consistent pattern of permission augmentations for all traces. For this purpose, we use the cut points selected in Section 4 and perform symbolic execution from them not only for symbolic values, but also for permissions.

Let $P^1, \ldots, P^n$ be a list of cut points from the cut-simulation check (Section 4.3), where each $P^i = (c^i_{\text{df}}(\mathbf{x}^i), c^i_{\text{seq}}(\mathbf{y}^i)) \mid \varphi^i(\mathbf{x}^i, \mathbf{y}^i)$ is a pair of symbolic dataflow and Seq configurations with an additional correspondence constraint. During the cut-simulation check in Section 4.3, we verify that starting from any dataflow cut point $c^i_{\text{df}}(\mathbf{x}^i)$, any symbolic execution branch reaches an instance of another cut point $c^j_{\text{df}}(\mathbf{x}^j)$. A consequence is that the symbolic branches from all dataflow cut points "cover" any complete trace in the canonical schedule (Definition 3). In other words, any complete concrete trace in the canonical schedule is the concatenation of instances of symbolic branches.

Thus, if we are able to find a consistent permission augmentation for the trace of each dataflow symbolic branch during the cut-simulation check, then any concrete trace in the canonical schedule would have a consistent permission augmentation, which implies that any other schedule will converge to the canonical schedule, and the confluence of the entire dataflow program follows.

This full confluence check algorithm is procedurally described in Algorithm 1. The algorithm is parameterized with a list of dataflow cut points $c^1_{\text{df}}(\mathbf{x}^1), \ldots, c^n_{\text{df}}(\mathbf{x}^n)$ and a predetermined permission algebra $P(L, k)$. The algorithm first generates a fresh permission variable for each value present in the channels of the dataflow cut points. Then it performs symbolic execution from each cut point, carrying constraints for both symbolic values and their corresponding permission variables, until the symbolic branch is subsumed by another cut point. In the end, it checks for the satisfiability of the permission constraints. If they are satisfiable, the confluence of the full program follows.

In more detail, the first loop at Line 3 assigns fresh permission variables to each value present in each cut point, representing an unknown concrete permission token to be solved for. We also add disjointness constraints at Line 7 to enforce that these permission augmentations are consistent (Definition 11) at the corresponding cut point configurations.

Then in the main loop at Line 8, we symbolically execute from each dataflow cut point, trying to match each symbolic branch against other cut points. The step function **step_canonical** uses a deterministic canonical schedule (Definition 3) we constructed from Seq execution traces. For example, in the case of LLVM and RipTide, we use the canonical schedule described in Section 4. **step_canonical** is also instrumented to take a permission augmentation t for the current configuration $c_{\text{df}}(\mathbf{x}^i)$, and output a modified permission augmentation t' for the stepped configuration $c'_{\text{df}}(\mathbf{x}^i)$, in which fresh variables are generated for all new output values in $c'_{\text{df}}(\mathbf{x}^i)$, and permissions corresponding to input values are removed. In each step, for each symbolic branch, we also accumulate permission constraints (Lines 14-21) to enforce that the augmented transition $(c_{\text{df}}(\mathbf{x}^i), t)$ to $(c'_{\text{df}}(\mathbf{x}^i), t')$ is a consistent transition (Definition 12).

At each symbolic step, we check if a symbolic branch is matched to another cut point $c^j_{\text{df}}(\mathbf{x}^j)$ (Line 23). This check is performed by finding a substitution $\sigma : \mathbf{y}^j \to \text{Terms}(\mathbf{x}^i)$ such that the current symbolic branch $c'_{\text{df}}(\mathbf{x}^i) = c^j_{\text{df}}(\sigma)$. If the symbolic branch matches the cut point, we add equality constraints (Line 25) to enforce that the permissions in t' should converge back to the original permissions t_j at the matched j-th cut point. Note that the substitution σ is not later used because we do not need to check correspondence constraints (Section 4.3).

The main loop at Line 8 should terminate, because if the cut-simulation check (Section 4) passes, any branch from any dataflow cut point should eventually match another cut point.

Finally, we check if the accumulated permission constraint ψ is satisfiable in the permission algebra $P(L, k)$ via an SMT query. If ψ is satisfiable, then any concrete trace (of unbounded length) in the canonical schedule has a consistent augmentation. Thus, by Theorem 3, any other possible schedule would eventually converge to the canonical schedule, and the dataflow program is confluent. If ψ is not satisfiable, then there are different possibilities: 1) the dataflow program is not confluent and may have data races (which require manual inspection to confirm); 2) the choice of k in the permission algebra $P(L, k)$ is not large enough; or 3) the confluence property is not provable using this method.

5.3 Liveness in the Bounded-Channel Model

In our current semantics for dataflow programs, channels between operators are modeled as *unbounded* queues. In real CGRA architectures, however, the channels have a fixed buffer size, and an operator may block if one of the output channels is full. As a result, a dataflow program in the bounded-channel model may have a deadlock that is not possible in the unbounded-channel model.

For example, consider Figure 7, where operator A has two dataflow paths to B: one path is full, while the other is empty due to a Steer operator T discarding its inputs. This situation puts us in deadlock. Operator B cannot fire without tokens along both paths, while operator A cannot fire before more space is made along the full path.

To verify that this scenario does not occur, we lift our cut-simulation and confluence results in the unbounded-channel model to the bounded-channel model via two observations:

- The symbolic branches in the dataflow cut point execution (Section 4.3) have finitely many intermediate symbolic configurations, so the maximum channel size is bounded by some integer K in the canonical schedule.
- The confluence results in Section 5.1 and Section 5.2 are still valid in the bounded-channel model, which we have formalized and verified in Verus as well.

Therefore, if the cut-simulation and the confluence checks pass in the unbounded-channel model, the same results hold in the bounded-channel model if channels have a buffer size of at least K. As a result, in the bounded-channel model, the dataflow program is still equivalent to the input imperative program, which guarantees the liveness property of the dataflow program that it will always make progress if the sequential imperative program is able to make progress. Furthermore, since the sequential program is deadlock-free, the dataflow program is also deadlock-free.

6 IMPLEMENTATION AND EVALUATION

We have implemented our translation validation technique in the tool FlowCert targeting the RipTide compiler from LLVM programs to dataflow programs. It is publicly available in an anonymized GitHub repository [Anonymous Authors 2024]. We implemented symbolic executors for (a subset of) LLVM and dataflow programs in Python. Based on these, we implemented the cut-simulation check described in Section 4 and the confluence check in Section 5. We also instrumented the RipTide compiler to output hints for constructing the canonical schedule (Section 4).

FlowCert uses the Z3 SMT solver [De Moura and Bjørner 2008] to discharge the verification conditions it generates. In the simulation check, FlowCert encodes feasibility of path conditions and validity of the correspondence conditions at each cut point into Z3 queries. In the confluence check, FlowCert produces a list of permission constraints and checks if these constraints are satisfiable with respect to the finite permission algebra (Definition 8) by encoding each permission variable as a set of Boolean variables and the list of permission constraints as an SMT query.

In order to evaluate FlowCert's capability to certify compilation to real dataflow programs, we have applied FlowCert to a benchmark of 21 programs consisting of the benchmark programs

Name	LOC	#OP	#PC	Sim.	Conf.	Description
nn_vadd	5	15	244	0.02	0.17	Vector addition (RipTide)
nn_norm	10	19	316	0.02	0.19	Neural network normalization (RipTide)
nn_relu	11	19	844	0.03	0.45	Neural network ReLU layer (RipTide)
smv	12	25	1103	0.05	2.27	Sparse-dense matrix-vector mult. (Pipestitch)
dmv	10	31	1234	0.06	1.64	Dense-dense matrix-vector mult. (RipTide)
Dither	19	31	2553	0.13	2.29	Dithering (Pipestitch)
SpSlice	30	31	2616	× 0.13	7.94	Sparse matrix slicing (Pipestitch)
nn_fc	23	36	2632	0.09	3.35	Neural network fully-connected layer (RipTide)
SpMSpVd	45	40	3182	0.15	8.38	Sparse-sparse matrix-vector mult. (Pipestitch)
bfs	48	48	3634	0.15	7.68	Breadth-first search (RipTide)
dfs	45	49	3762	0.17	7.76	Depth-first search (RipTide)
smm	22	49	3967	0.16	11.2	Sparse-dense matrix mult. (RipTide)
nn_pool	32	52	8165	0.23	7.14	Neural network pooling layer (RipTide)
dmm	16	56	4220	0.19	6.16	Dense-dense matrix mult. (RipTide)
sconv	22	56	4272	0.2	10.37	Sparse convolution (RipTide)
sort	23	57	6554	× 0.14	3.31	Radix sort (RipTide)
SpMSpMd	39	60	7693	0.33	26.68	Sparse-sparse matrix mult. (Pipestitch)
nn_conv	42	61	12154	0.2	15.32	Neural network convolution layer (RipTide)
dconv	24	65	8134	0.25	11.51	Dense convolution (RipTide)
fft	29	70	2690	0.12	5.77	Fast Fourier transform (RipTide)
sha256	81	181	3548	× 0.28	12.6	SHA-256 hash

Fig. 5. Evaluation results on 21 test cases sorted by the number of dataflow operators. From left to right, the columns are the name of the test case (Name), lines of code in the original C program (LOC), number of dataflow operators (#OP), number of permission constraints (#PC), time spent for simulation check in seconds (Sim.), and time spent for confluence check in seconds (Conf.). All test cases are originally written in C and then compiled to LLVM via Clang [LLVM 2024a]. Compiler bugs cause the simulation checks to fail in SpSlice, sort, and sha256. In all other cases, the simulation and confluence checks succeed.

in RipTide [Gobieski et al. 2023] and Pipestitch [Serafin et al. 2023], as well as some programs implementing neural network inference and an implementation of SHA-256 [Conte 2024]. Figure 5 shows some statistics about the benchmark programs and how FlowCert performs. All tests are performed on a laptop with an Apple M1 processor and 64 GiB of RAM. The time spent in the RipTide compiler (including hint generation) is less than 0.1 seconds for all of the benchmark programs, so it is omitted from the table.

We now evaluate FlowCert in three aspects: generality of cut-point heuristics, performance, and size of the trusted computing base.

Generality of the heuristics. Both simulation and confluence checks pass for most of the benchmark programs. This shows that our heuristics for inferring dataflow cut points work well on the canonical schedule, and our confluence check is general enough for common compilation patterns. In test cases sort, SpSlice, and sha256, the simulation check fails and correctly reveals two compilation bugs (see Section 6.1 for more detail). After fixing these bugs, the simulation and confluence checks succeed for these two programs.

Performance. FlowCert spends most of its time on the confluence check, mainly consisting of checking satisfiability for permission constraints. The simulation check takes a relatively short amount of time, because the input LLVM program and the output dataflow program match well

```
%xor = xor i1 %b, 1
%conv = zext i1 %xor to i32
store i32 %conv, i32* %A
```

```
for (int i = 0; i < n; i++) {
  A[i] = i;
  for (int j = i + 1;
       j < n; j++)
    A[j] = A[j - 1];
}
```

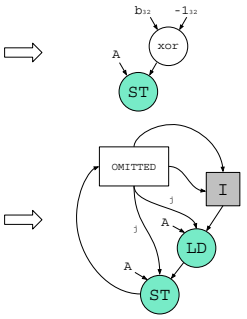


Fig. 6. Examples of two compiler bugs.

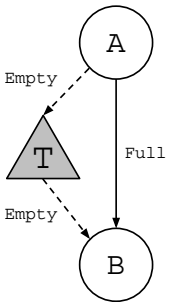


Fig. 7. Deadlock example.

on the canonical schedule and thus the verification conditions are simple to solve. Overall, the verification time is within 15 seconds for most benchmark programs, and we believe it is feasible to enable FlowCert in real production scenarios.

Trusted Computing Base. The original RipTide compiler has ~21,000 lines of C++ code, excluding comments. The instrumented hint generation in the RipTide compiler is implemented in less than 50 lines of C++. Compared to the RipTide compiler, the trusted computing base of FlowCert has 3,121 lines of Python with three main components: ~800 lines of dataflow semantics; ~1,000 lines of LLVM semantics; and ~750 lines for simulation and confluence checks.

6.1 Compiler Bugs

While testing FlowCert, we found 8 bugs in the RipTide compiler, confirmed by RipTide authors. Four of these bugs were discovered on the test cases `sort`, `SpSplice`, `sha256`, and their variations; and the remaining four bugs were found on test cases not included in the evaluation. We now discuss two representative bugs:

Incorrect signed extension. FlowCert found a compilation bug involving integer signed vs. unsigned extension in the `sort` test case. A simplified LLVM program used to reproduce this issue is shown in the top left of Figure 6, and the output dataflow program is shown in the top right of Figure 6. The LLVM code performs a 1-bit xor with a 1-bit constant 1, and it zero-extends the result to a 32-bit integer. On the dataflow side, the dataflow compiler converted all operations into 32-bits (word length of RipTide). However, during the process, the compiler assumes that all constants should be sign-extended. As a result, it extends the 1-bit 1 to a 32-bit `-1`, which is incorrect in this case. This difference causes our simulation check to fail, as the final memory states fail to match.

Incorrect memory ordering with potential data races. Our confluence check catches a compilation bug when compiling from the C program shown in the bottom left of Figure 6 to the dataflow program in the bottom right of Figure 6. The C program has a nested loop. In each iteration of the outer loop, it first sets `A[i]` = i, and then in the inner loop that follows, it copies `A[i]` into `A[i + 1]`, ..., `A[n - 1]`. In the output dataflow program, the compiler allows the load and store operators in the inner loop to pipeline using the invariant operator (marked `I`) to repeatedly send the signal to enable load. However, this causes a data race, because in the inner loop, a load in the second iteration accesses the same memory location as the store in the first iteration.

The confluence check fails as the generated permission constraints are unsatisfiable. FlowCert also outputs an unsat core of 57 constraints, a subset of the 420 constraints generated, that causes the unsatisfiability. In particular, the unsat core includes two constraints: `read A` is in the input

tokens of the inner loop load, and `write A` is in the input tokens of the inner loop store. This indicates that the main conflict stems from these two operators.

7 LIMITATIONS AND FUTURE WORK

Our translation validation technique still has some limitations to address in future work.

Optimized dataflow graphs. Our technique may fail on more optimized programs. The RipTide compiler employs some optimizations currently not supported by FlowCert:

- Deduplication of operators, which combines arithmetic operators with the same inputs.
- Streamification, which uses a single stream operator to generate a loop induction variable (as opposed to using, e.g., carry, comparison, and addition operators).
- Array dependency analysis, which removes unnecessary orderings between loads and stores across iterations that access different indices of an array.

The first two optimizations are not supported by our simulation check because we require a one-to-one correspondence between LLVM instructions and dataflow operators for the canonical schedule. Since these optimizations involve reasoning about the equivalence between dataflow programs, we believe that a modular approach is to use FlowCert to verify the input LLVM against the unoptimized dataflow program, and use another verification pass in future work to soundly optimize the dataflow program.

The third optimization may lead to a dataflow program failing our confluence check. To solve this, we need finer granularity in permission tokens. Currently, the `read` and `write` permissions are for the entire memory region, such as an array `A`. To allow memory operations on `A[i]` and `A[i + 1]` to run in parallel, where `i` is the loop induction variable, we need to re-formulate the permission algebra to allow slices of a permission token on a specific range of indices. This also requires a form of "dependent" permission tokens that can depend on the values in the channels.

Scalability of the confluence check. FlowCert currently spends most of its time on the confluence check (Section 6). We believe this is primarily due to a naive encoding of permission constraints in SMT. For better performance in future work, one could design a custom solver for permission constraints to better utilize the structure of a permission algebra, or use a preprocessing step to simplify permission constraints.

Lack of architecture-specific details. This work targets the compilation from imperative programs to an abstract version of dataflow programs. While our technique can ensure the correctness of this pass, we do not verify later passes, such as the procedure to optimally configure the dataflow program on the actual hardware.

8 RELATED WORK

As a compiler verification technique, the translation validation approach taken in our work is a less labor-intensive alternative to full compiler verification, such as CompCert [Leroy 2009]. Translation validation was first proposed by Pnueli et al. [1998], and there are numerous following works with novel translation validation or program equivalence checking techniques for various languages and compilers [Kundu et al. 2009; Necula 2000; Sharma et al. 2013; Tate et al. 2009]. For LLVM-related works on translation validation, we have Alive [Lopes et al. 2021, 2015], which performs bounded translation validation for internal optimizations in LLVM. LLVM-MD [Govereau 2012; Tristan et al. 2011] is another translation validation tool for LLVM optimizations. LLVM-MD translates the LLVM code into an intermediate representation called synchronous value graphs (SVG) [Govereau 2012], which is a similar representation to a dataflow program. The difference is that the semantics of SVG is synchronous (in the sense that values are not buffered in the channels), and they also trust

the compilation from LLVM to SVG, whereas in our case, such translation is exactly what we are trying to validate. Our formulation of cut-simulation in our simulation check is an adaptation from the work on Keq [Kasampalis et al. 2021], which aims to be a more general program equivalence checker parametric in the operational semantics of input/output languages.

Our use of linear permission tokens is a dynamic variant of the fractional permissions proposed by Boyland [Boyland 2013] and used in various separation logics [Brookes 2006] to reason about read-only sharing of references. Tools based on separation logics such as Iris [Jung et al. 2018] and Viper [Müller et al. 2016] usually require manual annotations for permission passing in the pre-/post-conditions, whereas our permission tokens are synthesized fully automatically. Furthermore, separation logic is not a suitable formalism to directly reason about dataflow programs due to the lack of structured control-flow in a dataflow program and the asynchrony of dataflow operators.

Various models of dataflow programming [Johnston et al. 2004] have been studied, such as computation graphs [Karp and Miller 1966] and Kahn process networks [Gilles 1974]. An important aspect of these works is determinacy; i.e., these models produce a deterministic result regardless of the schedule of execution. There are also verification tools for classical dataflow languages such as Lustre [Bourke et al. 2017; Hagen and Tinelli 2008; Halbwachs et al. 1991] and Esterel [Bouali 1998]. However, dataflow programs used by CGRA compilers often make use of shared global memory, which requires stronger techniques for guaranteeing determinacy.

Process calculi such as CSP [Roscoe 1997] and π-calculus [Milner 1999] present an alternative formalism for message-passing concurrency. To manage the inherent complexity and nondeterminism in process calculi, session types can enforce confluence and deadlock-freedom by construction [Sangiorgi and Walker 2001]. However, similar to separation logic, session types require manual typing annotations that would be difficult to synthesize automatically, and we likely still need some form of fractional permissions to handle shared memory.

Our strategy for analyzing nondeterministic dataflow programs is reminiscent of verification strategies for distributed systems [Bakst et al. 2017; Kragl et al. 2020; von Gleissenthall et al. 2019] which analyze message-passing protocols (e.g., Paxos [Lamport 2001]) via reductions to sequential programs. However, the dataflow domain requires significantly different techniques: we verify *equivalence properties* with arbitrary specification programs, and use *linear permissions* to enable safe memory accesses in the presence of asynchrony.

In the context of term rewriting systems [Baader and Nipkow 1998], the standard approach for confluence checking is the use of critical pairs [Durán et al. 2020; Knuth and Bendix 1970], which proves ground confluence in the case when the rewriting system is terminating. This algorithm is implemented in various systems, such as the Church-Rosser checker in Maude [Clavel et al. 2002]. However, this technique is difficult to apply in our case, since the semantics of a dataflow program is neither terminating nor, in general, ground confluent.

9 CONCLUSION

In this work, we develop a technique for translation validation between imperative source programs and asynchronous dataflow target programs, with an implementation for the RipTide CGRA architecture. Our technique simplifies analysis on dataflow by first verifying that the target program simulates the source program along a canonical schedule, and then using linear memory permissions to show that this canonical schedule is general. Our verification procedure ensures both functional correctness and liveness for the target dataflow program.

In future work, we aim to support optimizations on dataflow programs (e.g., special-purpose operators for pipelining loops), and extending our verification methodology to concrete instantiations of CGRAs in hardware.

REFERENCES

Ackerman. 1982. Data Flow Languages. *Computer* 15, 2 (1982), 15–25. https://doi.org/10.1109/MC.1982.1653938

Anonymous Authors. 2024. Anonymized GitHub Repository. https://github.com/flowcert/flowcert.

Arvind and David E. Culler. 1986. *Dataflow Architectures.* Annual Reviews Inc., USA, 225–253.

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press, UK. https://doi.org/10.1017/CBO9781139172752

Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. *Proceedings of the ACM on Programming Languages* 1 (2017), 1 – 27. https://api.semanticscholar.org/CorpusID:23362468

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

Amar Bouali. 1998. Xeve, an Esterel verification environment. In *Computer Aided Verification*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 500–504.

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. https://doi.org/10.1145/3062341.3062358

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

Stephen Brookes. 2006. Variables as Resource for Shared-Memory Programs: Semantics and Soundness. *Electronic Notes in Theoretical Computer Science* 158 (2006), 123–150. https://doi.org/10.1016/j.entcs.2006.04.008 Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII).

M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J.F. Quesada. 2002. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science* 285, 2 (2002), 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0 Rewriting Logic and its Applications.

Brad Conte. 2024. Basic implementations of standard cryptography algorithms. https://github.com/B-Con/crypto-algorithms.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Jack B. Dennis and David P. Misunas. 1974. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA '75)*. Association for Computing Machinery, New York, NY, USA, 126–132. https://doi.org/10.1145/642089.642111

Francisco Durán, Jose Meseguer, and Camilo Rocha. 2020. Ground confluence of order-sorted conditional specifications modulo axioms. *Journal of Logical and Algebraic Methods in Programming* 111 (2020), 100513. https://doi.org/10.1016/j.jlamp.2019.100513

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

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. https://doi.org/10.1109/ISCA52012.2021.00084

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. https://doi.org/10.1109/MICRO56248.2022.00046

S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer* 33, 4 (2000), 70–77. https://doi.org/10.1109/2.839324

Paul Govereau. 2012. *Denotational Translation Validation.* Ph. D. Dissertation. Harvard University, USA. Advisor(s) Morrisett, John G. AAI3495610.

George Hagen and Cesare Tinelli. 2008. Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques. In *2008 Formal Methods in Computer-Aided Design*. IEEE Press, USA, 1–9. https://doi.org/10.1109/FMCAD.2008.ECP.19

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *ACM Comput. Surv.* 36, 1 (mar 2004), 1–34. https://doi.org/10.1145/1013208.1013209

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Richard M. Karp and Rayamond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. https://doi.org/10.1137/0114108 arXiv:https://doi.org/10.1137/0114108

Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1004–1019. https://doi.org/10.1145/3445814.3446751

Donald E. Knuth and Peter B. Bendix. 1970. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*. Pergamon, Oxford, UK, 263–297. https://doi.org/10.1016/B978-0-08-012975-4.50028-X

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. 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, 227–242. https://doi.org/10.1145/3385412.3385980

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/1542476.1542513

Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* 4, 32 (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

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. https://doi.org/10.1145/3586037

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

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. https://doi.org/10.1145/3357375

LLVM. 2024a. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

LLVM. 2024b. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

Robin Milner. 1999. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, United States.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62.

George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

A. W. Roscoe. 1997. *The Theory and Practice of Concurrency.* Prentice Hall PTR, USA.

Davide Sangiorgi and David Walker. 2001. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, USA.

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. https://doi.org/10.1145/3613424.3614283

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

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.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages* 3 (2019), 1 – 30. https://api.semanticscholar.org/CorpusID:57757310

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. https://doi.org/10.1109/TPDS.2013.125