A Compiler Infrastructure for Accelerator Generators

Rachit Nigam*
Cornell University
USA

Samuel Thomas*
Cornell University
USA

Zhijing Li
Cornell University
USA

Adrian Sampson
Cornell University
USA

ABSTRACT

We present Calyx, a new intermediate language (IL) for compiling high-level programs into hardware designs. Calyx combines a hardware-like structural language with a software-like control flow representation with loops and conditionals. This split representation enables a new class of hardware-focused optimizations that require both structural and control flow information which are crucial for high-level programming models for hardware design. The Calyx compiler lowers control flow constructs using finite-state machines and generates synthesizable hardware descriptions.

We have implemented Calyx in an optimizing compiler that translates high-level programs to hardware. We demonstrate Calyx using two DSL-to-RTL compilers, a systolic array generator and one for a recent imperative accelerator language, and compare them to equivalent designs generated using high-level synthesis (HLS). The systolic arrays are 4.6× faster and 1.11× larger on average than HLS implementations, and the HLS-like imperative language compiler is within a few factors of a highly optimized commercial HLS toolchain. We also describe three optimizations implemented in the Calyx compiler.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**.

KEYWORDS

Intermediate Language, Accelerator Design

ACM Reference Format:
Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators . In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446712

*Equally contributing authors.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446712

1 INTRODUCTION

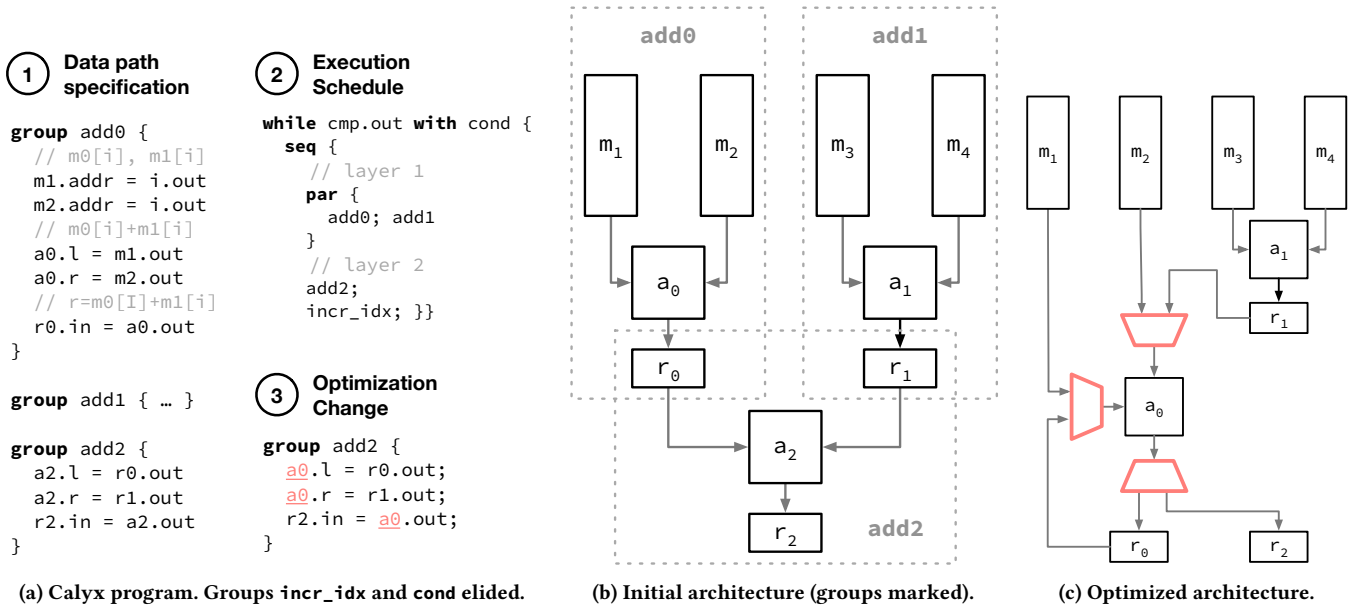
Hardware design is a language problem. While custom hardware accelerators are economically justified in a post Moore's law era, we have yet to see widespread adoption. Even though reconfigurable architectures, such as field programmable gate arrays (FPGAs), make it easy to deploy accelerators, the tooling and languages inhibit ubiquitous use. Hardware description languages (HDLs) operate at the level of gates, wires, and clock cycles; while this level of abstraction is useful for designing high-end processors, it is inappropriate for rapid design and deployment of computational accelerators.

To liberate hardware design from these low-level abstractions, researchers have proposed several compilers for high-level specification languages. The traditional approach is high-level synthesis (HLS): to compile legacy software languages such as C, C++, or OpenCL to HDLs [3, 14, 24, 44, 45]. However, such languages are a poor fit for generating hardware—they reflect pointer-based, sequential, von Neumann models of computation. The hardware they seek to generate is pervasively parallel, without a unified address space, and free from program counters.

The cavernous semantic gap between C++ and HDLs motivates a more domain-specific approach. A new wave of hardware languages and compilers focus on a specific application category [30, 40], on a specific architecture style [8], or on lifting hardware-level concerns into a restricted imperative language [18, 25]. These narrower languages sacrifice the familiarity and backwards compatibility of traditional HLS to simplify compilation, generate better hardware, and avoid the uncanny valley of inconsistent software-like semantics. They can focus on providing high-level abstractions that concisely capture the parallelism of the application domain.

DSL-to-hardware compilers, however, remain substantial feats of engineering. The compiler developer needs not only to conceive of a high-level architecture; they must also design a data path and a control path to implement the execution strategy and perform architectural optimizations [8, 18]. Each such compiler re-engineers a new intermediate language (IL) to encode the high-level semantics of the input language while exposing architectural information to perform optimizations. A shared IL, along with a compiler infrastructure that implements useful optimizations and analyses, will let compiler engineers design new hardware DSLs and quickly get competitive hardware designs.

We propose Calyx, a new intermediate language for compiling DSLs to hardware. Calyx combines a software-like imperative sublanguage, which explicitly represents the control flow of a design, with a structural language, which instantiates hardware modules and describes connections between them. Frontend compilers can



(a) Calyx program. Groups `incr_idx` and `cond` elided. (b) Initial architecture (groups marked). (c) Optimized architecture.

Figure 1: Calyx describes the reduction tree using its split representation. The execution schedule makes the control flow explicit while encapsulate connections between hardware modules. Done signals (Section 3.3) elided from group definitions.

specify architectural details using the structural sub-language and rely on the high-level control language to encode a DSL's semantics. The Calyx compiler optimizes these programs, generates control logic, and emits synthesizable RTL.

The contributions of this paper are:

- Calyx, an intermediate language for compiling DSLs to hardware that uses a split representation combining a high-level control flow language with a hardware-like structural language.
- An open-source pass-based compiler for analyzing, optimizing, and lowering Calyx programs to synthesizable RTL.
- The implementation of two compilers that target Calyx: (1) a PE-parametric systolic array generator that encodes the data movement and computation schedule using Calyx's control language, and (2) Dahlia [25], a general-purpose programming language for accelerator design which has a preexisting backend targeting high-level synthesis (HLS) toolchains.
- Three optimizations implemented within the Calyx compiler: resource sharing, live-range-based register-sharing, and a pass to infer cycle latencies.

2 OVERVIEW BY EXAMPLE

This section introduces Calyx by using it to implement a parallel *reduction tree*. A reduction tree applies an operator to many inputs to produce a single output. Figure 1b shows a small summation tree on four inputs. The operators within a tree level run in parallel to produce the inputs to the next level. Unlike hardware description languages (HDLs) or high-level synthesis (HLS), Calyx programs are meant to be generated by compiler frontends. We show that with Calyx's control language, compilers can encode the semantics of high-level languages while producing programs amenable to hardware optimization.

2.1 Reduction Tree in Calyx

Figure 1a shows a Calyx program fragment that implements a parallel reduction tree that computes $(m_0 + m_1) + (m_2 + m_3)$. The program uses *groups* to specify the data path (1). Groups encapsulate hardware connections that implement an action. For example, the group `add0` uses the hardware adder `a0` to compute the sum of the first two inputs and save the result in a register `r0`. The assignments used inside groups correspond to *non-blocking assignments* in RTL languages—updates to the left hand side of an assignment are immediately propagated to the right hand side. In this way, each group encapsulates a data flow graph.

To compute the reduction, we need to schedule the execution of the layers. We want to execute the layers sequentially and to run the adders inside a tree layer in parallel. The Calyx program specifies the reduction tree's schedule using a separate *control* language (2). The control language uses group names to activate hardware connections. Unlike groups, control statements have no direct hardware analog—instead, they resemble a small imperative program with explicit parallelism. The schedule iterates over the memories using a `while` statement and sequences the execution of the layers using the `seq` operator. The `par` operator specifies that the adders in the first layer will be executed in parallel. Finally, the loop body uses the group `incr_idx` to increment the index into the memories.

Figure 1b shows the high-level architecture generated from the Calyx program and marks the connections that correspond to the groups. The figure elides the control circuitry generated to implement the schedule.

2.2 Optimizing Accelerator Designs

High-level specifications of accelerators encode a treasure trove of control flow information that is lost when lowering to a register-transfer level (RTL) language. Compilers for such programming models need a stable intermediate language (IL) to capture and use such information. However, RTL is ill-suited for this task.

RTL languages do not distinguish between control flow and data flow because they implement both using the same structural constructs. For example, in order to sequence two operations, an RTL program must implement a state machine to track the current state. Such a state machine is implemented using registers and adders which are indistinguishable from registers and adders used to implement the program's data flow. This conflation means that a compiler cannot automatically extract and transform the control flow of an arbitrary RTL program.

Consider an optimization that reuses existing circuitry to perform temporally disjoint computations. For example, our reduction tree uses adders `a0` and `a2` in two different stages and never overlaps their execution. Therefore, it would be safe to transform the program to share a single adder for both the stages. Implementing this optimization in RTL, however, is difficult because the structural implementation of a state machine obscures the program's control flow. To determine that the two adders run at different times, an analysis would need to reverse-engineer the execution schedule from the state machine implementation. Furthermore, transforming an RTL program would require pervasive changes. Figure 1c shows the optimized architecture. The transformation requires carefully rewiring the input and output signals for `a0` through multiplexers.

In contrast, a Calyx program makes the control flow explicit and enables straightforward transformation. Given the execution schedule of our Calyx program, it is clear that the groups `add0` and `add2` do not execute simultaneously since they are scheduled using the `seq` operator. Figure 1a ③ shows the only change required to implement this optimization. The Calyx program simply renames the uses of `a2` in group `add2` with `a0` and the compiler correctly generates the additional multiplexers and control signals to share the adder.

2.3 Structure and Control

Calyx is neither a software IL nor a hardware IL. Software ILs, such as LLVM [22], focus on providing a uniform representation of the control flow and data flow of a program. They do not explicitly represent structural facts, such as the mapping of logical adds onto physical adders. On the other hand, hardware ILs focus on a purely structural representation with explicit use of gates, wires, and clocks while conflating data flow with control signals. By marrying structure and control, Calyx provides access to both structural and control flow facts to enable a new class of optimizations that cannot be captured by either style of ILs.

3 THE CALYX INTERMEDIATE LANGUAGE

The Calyx infrastructure's focal point is its program representation. The Calyx IL aims to represent domain-specific accelerator designs throughout the entire lifetime of a hardware generation pipeline: generation from a language frontend, optimization and lowering, and implementation in a hardware description language. This section describes the Calyx IL; the following sections show how to generate, lower and optimize the IL.

3.1 Components

Calyx programs consist of *components* which encapsulate hardware structures and define an execution schedule to orchestrate their behavior:

```
component name(inputs) -> (outputs) {
  cells { ... }
  wires { ... }
  control { ... }
}
```

The body includes hardware-like structural listings of *cells* and *wires* (Section 3.2) and software-like *control* code (Section 3.3). The input and output ports form the interface to the component and define their size in bits. For example, a component defining a 32-bit integer adder uses these ports:

```
component adder(lhs: 32, rhs: 32) -> (sum: 32)
```

Ports in Calyx are *untyped*—they can hold any value of a given width. Calyx leaves type-based reasoning to the language frontend.

3.2 Cells and Wires

Calyx programs explicitly instantiate components and define the connections between them in a way that closely resembles RTL languages. This low-level of detail gives frontends precise control over fine-grained architectural choices when needed and lets Calyx lower programs to synthesizable RTL.

The cells section instantiates components:

```
cells {
  a_reg = std_reg(32); // 32-bit register
  add = std_add(32);   // 32-bit adder
}
```

This example instantiates a register and an adder that operate on 32-bit values using the `std_reg` and `std_add` components. The wires section defines *assignments* between the ports of components:

```
wires {
  add.left = a_reg.out;
  add.right = a_reg.out;
}
```

These *assignments* connect the `out` port of the register to the two input ports of the adder. The connections are *non-blocking*: updates to `a_reg.out` are immediately visible to `add.left`. This closely resembles non-blocking assignments in RTL languages.

Wire assignments can specify more complex dataflow policies by using *guarded assignments*:

```
add.left = cmp.out ? a_reg.out;
add.left = !cmp.out ? b_reg.out;
```

These guarded assignments to the `left` port of the `add` component use the value of `cmp.out` to determine which assignment to activate. Guards can be built using ports and a language of simple boolean connectives.

Like its RTL counterparts, Calyx requires that each port have a *unique driver*—activating multiple assignments in the same cycle results in undefined behavior. This requirement also differentiates Calyx's guarded assignments from Bluespec's atomic rules [26]. While Bluespec resolves conflicting assignments by generating

scheduling logic to dynamically abort them, Calyx does not. Being an intermediate language, Calyx trades-off the convenient programming abstraction for predictable compilation.

Guarded assignments in Calyx correspond exactly to assignments in RTL languages. By themselves, they can encode arbitrary hardware designs, but are less amenable to analysis and transformation. The next section describes Calyx's two novel constructs that simplify the specification of a program's structural connections and its execution schedule.

3.3 Groups and Control

Calyx uses *groups* to encapsulate assignments. Inside a group, assignments must obey the same constraints as RTL—unique drivers for ports, no combinational loops, etc. However, multiple groups can use the same port:

```
group assign_one { x_reg.in = 1; }
group assign_two { x_reg.in = 2; }
```

Both groups unconditionally assign to the same port. However, since the groups encapsulate the assignments, they are not active by default and do not violate the unique driver requirement. In contrast, RTL languages require programmers to reason about all assignments to a port and weave in control signals to define a unique driver.

The *control program* determines when groups run:

```
control { seq { assign_one; assign_two } }
```

The control block uses the `seq` (sequence) statement to specify that `assign_one` executes first, followed by `assign_two`. Since the two groups execute at different times, the two assignments to the port `x_reg.in` do not conflict and Calyx can generate valid RTL.

While control statements like `seq` can pass the control flow of a program to a group, they have no way to know when to return—groups can encode arbitrary computations that don't have an obvious done condition. To signal when it has finished executing, a group use a `done` signal:

```
group assign_one {
  x_reg.in = 1;
  assign_one[done] = x_reg.done;
}
```

In the above group, we are writing a value to a stateful element `x_reg`, and must wait for the element to signal that the write was committed. The group uses the `x_reg.done` port to signal that the group's computations has finished.

Interface signals, such as a group's done signal, are used by Calyx to define a *calling convention* (Section 4.1). A control program passes control flow to a group by setting a group's `go` to 1 and the group returns control by setting its `done` signal to 1. Similarly, components use `go` and `done` interface signals to define a consistent calling convention. Calyx's interface is *latency-insensitive*; it does not not reason about the number of cycles needed to execute a computation. Section 4.4 shows how enriching Calyx programs with latency information enables efficient compilation.

3.4 Control Statements

Calyx provides several primitives to encode the schedule of components. We designed these primitives to capture high-level properties such as branching and looping, freeing frontends from the need to realize them in control circuitry.

enable. Naming a group inside a control statement passes control to the group.

par. List of control statements that execute once in parallel.

```
par { group_a; seq { group_b; group_c; }; group_d; }
```

seq. List of control statements executed in order.

```
seq { group_a; par { group_b; group_c; }; group_d; }
```

if. Conditionally executes one of the branches. Specifies a port to use as the 1-bit conditional value (*port_name*) and a group (*cond_group*) to compute the value on the port.

```
if port_name with cond_group {
  true_stmt
} else {
  false_stmt
}
```

while. The loop statement is similar to the conditional. It enables *cond_group* and uses *port_name* as the conditional value. When the value is high, it executes *body_stmt* and recomputes the conditional using *cond_group*.

```
while port_name with cond_group {
  body_stmt
}
```

3.5 Attributes

Calyx programs can use *attributes* to encode frontend and pass-specific information such as the latency of a group. Attributes are key-value pairs. For example, the following group defines an attribute "latency" and associates the value 1 to it.

```
group foo<"latency"=1> { ... }
```

3.6 Synopsis

Components are the building blocks of Calyx programs. Each component instantiates subcomponents (*cells*) and defines the connections between them (*wires*). The *control* program defines the execution schedule by enabling groups.

The design principle behind Calyx is thus: in general, frontends generate small groups to perform simple actions, such as incrementing a register or comparing values, and use the control flow program to schedule them. However, when frontends have domain-specific knowledge, they can instantiate complex architectures and encapsulate them using groups.

4 COMPILING CALYX TO HARDWARE

The Calyx compiler optimizes (Section 5) and lowers Calyx programs into synthesizable RTL. Compilation passes use *interface signals*, which define a calling convention, to realize a component's execution schedule. The result is a Calyx program with a flat list of guarded assignments and no control statements or groups. The compiler can then directly translate this flattened form into RTL. The primary compilation passes are:

- GoInsertion: Guards all assignments in a group with the group's `go` interface signal.

```
group one {
  x.in = 1;
  one[done] = x.done;
}
group two {
  x.in = 2;
  two[done] = x.done;
}


control {
  seq { one; two }
}
```

(a) Original program

```
group one {
  x.in = one[go] ? 1;
  one[done] = x.done;
}
group two {
  x.in = two[go] ? 2;
  two[done] = x.done;
}


control {
  seq { one; two }
}
```

(b) GoInsertion

```
group one { … } // Unchanged
group two { … }
group seq0 {
  // enable contained groups
  one[go] = fsm.out == 0 ? 1;
  two[go] = fsm.out == 1 ? 1;
  // FSM state updates
  fsm.in =
    fsm.out == 0 & one[done] ? 1;
  fsm.in =
    fsm.out == 1 & two[done] ? 2;
  seq0[done] = fsm.out == 2 ? 1;
}
control { seq0 }
```

(c) CompileControl

```
wires {
  x.in = fsm.out == 0 ? 1;
  x.in = fsm.out == 1 ? 2;

  fsm.in =
    fsm.out == 0 & x.done ? 1;
  fsm.in =
    fsm.out == 1 & x.done ? 1;

  // done condition for the component
  done = fsm.out == 2 ? 1;
}

control { /* empty */ }
```

(d) RemoveGroups

Figure 2: Calyx realizes the execution schedule by encoding it with structural components. After the CompileControl pass (c), the `fsm` register encodes the current state for the `seq` statement.

- CompileControl: Generates latency-insensitive finite state machines to structurally realize control operators.
- RemoveGroups: Inlines uses of interface signals and eliminates all groups.
- Lower: Translates control-free Calyx to RTL.
- Sensitive: Opportunistically compiles control statements into groups using latency-sensitive FSMs. Only affects groups with `"static"` attribute.

Figure 2 illustrates the main steps. This section describes the complete compilation process.

4.1 Calling Convention

To realize a Calyx program's execution schedule, the compiler needs a mechanism to pass control flow in purely structural programs. We use a pair of *interface signals* to define this interface: when a group sets another group's go signal high, control is passed to that group and it can enable assignments within it; when a group sets its own done signal high, it passes control back. This interface resembles traditional latency-insensitive hardware design [4].

Most passes treat interface signals like any other 1-bit port. The main compilation passes treat them specially—using them to wire up the control signals. The final compilation pass eliminates interface signals by inlining them.

4.2 Compilation Workflow

We describe the compilation pipeline by compiling the example Calyx program in Figure 2a.

Inserting go interface signals. Calyx's semantics requires that assignments within a group are only enabled when the group executes. To enforce this requirement, the GoInsertion pass inserts the group's go signal into the guards of the contained assignments. Figure 2b shows the resulting program: `one[go]` guards assignments in group `one` while `two[go]` guards assignments in group `two`. When all groups are eventually removed, these guards will ensure that the correct set of assignments are active at a given time.

Compiling control using interface signals. The next step in the compilation process is realizing the control program using a structural implementation. Compilation relies on two important properties of Calyx: (1) groups can encode arbitrary computations, and (2) all groups are treated uniformly, regardless of the computation they perform—a group that increments a register is compiled the same way as a group that runs a systolic array.

The CompileControl pass performs a bottom-up traversal of the control program and does the following: (1) for each control statement, such as `seq` or `while`, instantiate a new group, called the *compilation group*, to contain all the structure needed to realize the control statement, (2) implement the schedule by setting the constituent groups' go and done signals, and (3) replace the statement in the control program with the corresponding *compilation group*. After this pass, every component's control program is reduced to a single group enable.

Figure 2c shows these transformations. The pass defines a new group `seq0` to encapsulate the structure required to realize the `seq` statement as well as a new register `fsm` to track the current state. Next, the pass enables the groups contained in the `seq` by writing to their go interface signals and updates the FSM state when the groups set their done signal high. The done condition for `seq0` is when the FSM reaches its final state. Finally, the pass replaces the `seq` control statement with the group `seq0`.

Inlining interface signals. The RemoveGroups pass inlines all uses of interface signals and removes all groups. It performs three transformations:

(1) Add new go and done *ports* to each component definition and wire them up to the single group enable in the control program.
(2) Collect all writes to a group's go and done signals and inline them into all uses of the signals. If there are multiple writes to a signal, replace the corresponding reads with a disjunction of the written expressions. This step eliminates all interface signals from the component.
(3) Remove all groups. Since all assignments are guarded by expressions that encode the schedule, it is safe to remove the groups and place them in the top-level wires section.

Figure 2d shows the resulting program that contains no groups, interface signals, or control statements.

Code generation. Each component now contains a flat list of guarded assignments. The Lower pass generates SystemVerilog programs

by mapping each component to a module, generating wires for all the ports, and threading a clock signal through the design.

4.3 Compiling Control Statements

The CompileControl pass performs a bottom-up traversal of the control program, encodes the control flow of each control statement using structural components, and replaces its use with corresponding compilation group. This example illustrates the timeline of bottom-up elimination of control statements:

```
control { par {
  seq { one; two; }
  seq { foo; bar; }
}}
```

```
control { par {
  seq0;
  seq1;
}}
```

```
control {
  par0;
}
```

We sketch the CompileControl pass's strategies for implementing each control statement in Calyx.

par. A `par` control block enables all groups inside it and finishes executing when all groups have signaled `done` once. Since groups may finish executing at different times, the pass generates a 1-bit register to save each child group's done signal. The go signal for each child group is set to high when the value in this register is 0. The done signal for the compilation group is 1 when all the 1-bit registers output 1.

if. Calyx's semantics dictate that an `if` statement executes a group `cond` before reading the value from a port and deciding which branch to execute. `cond` is supposed to update the value on the port. The pass generates two 1-bit registers: `cc` which tracks if `cond` has been executed, and `cs` to store the value of the port generated after executing `cond` to ensure that the value of the port is available through the execution of the branches. The compilation group enables either branch using the value in `cs` and finishes executing when the branch's done signal is high.

while. The loop compilation strategy resembles the one for `if`. The group runs the condition group, saves the value from the condition port to a register, and uses it to either enable the group in the body. The compilation group finishes executing when the value of the conditional port is 0.

Resetting compilation groups. Compilation groups reset their internal state to operate correctly within loops. The pass generates assignments that reset the value of internal state elements when a compilation group sets its done signal high.

4.4 Latency-Sensitive Compilation

The default compilation pass, CompileControl, generates latency-insensitive finite-state machines (FSMs) when realizing a component's schedule. Such latency-insensitive designs allow the execution schedule to uniformly reason about multi-cycle components and groups. The cost of this approach, however, is the additional hardware and additional execution cycles required to coordinate with the interface signals. Frontend compilers can often provide latency information that the compiler can exploit to build smaller and faster hardware.

We implemented a pass that can opportunistically generate latency-sensitive FSMs when latency information is available. This pass is best-effort: it only attempts to generate such FSMs when latency information is available and gracefully falls back to CompileControl. The encapsulation property of groups enables these kinds of best-effort passes—the compilation pipeline does not have to reason about what is inside a group to compile it.

The key benefits to this approach are: (1) frontends can quickly build a functioning end-to-end flow and incrementally add latency information to generated programs, and (2) latency-sensitive compilation is *just* an optimization—it can be disabled, debugged, and interacted with separately from the compilation pipeline. To the best of our knowledge, Calyx's ability to fluidly mix latency-sensitive and latency-insensitive compilation is unique. Prior systems intertwine latency information through the compilation process, so either everything is statically timed [44] or nothing is [16].

Section 6.2 shows how a frontend can generate latency information, Section 7.2 demonstrates that the pass speeds up designs by 1.43× without an area penalty, and Section 5.3 demonstrates how latency information can be automatically inferred in certain cases.

Compiling seq. The latency-sensitive compilation pass, Sensitive, traverses the control program bottom-up and opportunistically compiles control statements when all of the nested groups specify their latency using the `static` attribute (Section 3.5):

```
group one<"static"=1> { ... }
group two<"static"=2> { ... }
control { seq { one; two } }
```

It generates an FSM with a self-incrementing counter and enables each group for the specified number of cycles, and ignores the `done` signal from the groups:

```
group static_seq0<"static"=3> {
  one[go] = fsm.out >= 0 && fsm.out < 1 ? 1;
  two[go] = fsm.out >= 1 && fsm.out < 3 ? 1;
  static_seq0[done] = fsm.out == 3 ? 1;
  // Increment the FSM.
  fsm.in = fsm < 3 ? fsm.out + 1;
  static_seq0[done] = fsm.out == 3;
}
```

When compiling `seq`, `par`, or `if` statements, the pass uses the latency information of the contained groups to generate a `static` attribute for generated compilation group.

The pass demonstrates how Calyx enables development of small, modular passes that interact with the broader infrastructure. It is feasible because the IL has a well-defined semantics that lets passes reason independently about the preservation of program semantics.

5 OPTIMIZING CALYX PROGRAMS

We describe the design and implementation of three optimizations to demonstrate Calyx's ability to support control-flow-sensitive optimizations.

5.1 Resource Sharing

Resource sharing is an optimization that reuses existing circuits to perform temporally disjoint computations. For example, if an accelerator needs to perform two add operations that are never executed in parallel, it can map them to the same physical adder. Calyx is uniquely suited to implement such optimizations which require both control flow facts (if two computations run in parallel) and structural facts (which physical adder performs an add).

```
group let_r0 { r0.in = 0 }
group let_r1 { r1.in = 0 }
group incr_r0 {
  a0.l = r0.out; a0.r = 1;
  r0.in = a0.out; }
group incr_r1 {
  a1.l = r1.out; a1.r = 1;
  r1.in = a1.out; }
```

(a) Defined groups. `r0` and `r1` are registers; `a0` and `a1` are adders.

```
seq {
  par {
    let_r0;
    let_r1;
  }
  incr_r0;
  incr_r1;
}
```

(b) Schedule with resource sharing opportunities.

Figure 3: Resource sharing example. Since `incr_r0` and `incr_r1` do not run in parallel, they can share their adders.

Calyx implements a group-level resource sharing optimization: if two groups are guaranteed to never execute in parallel, they can share components. This pass does not attempt to share stateful components because state is visible across groups. Frontends use the `"share"` attribute (Section 3.5) to denote that a component is safe to share.

```
component adder<"share"=1> { ... }
```

The pass uses the execution schedule of a component to calculate which groups may run in parallel and relies on the encapsulation property of Calyx groups to implement sharing. It proceeds in three steps:

Building a conflict graph. A conflict graph summarizes potential conflicts—nodes denote groups and edges denote that the groups *may* run in parallel. The pass traverses the control program and adds edges between all children of a `par` block. For example, in Figure 3b, the groups `let_r0` and `let_r1` conflict with each other while `incr_r0` and `incr_r1` do not. If the children of the `par` block are themselves control programs, the pass adds edges between the groups contained within each child.

Greedy coloring. The pass performs a greedy coloring over the conflict graph to allocate shareable components to each group. If two groups have an edge between them, they cannot have the same components. The result of this step is a mapping from the names of old components to new components. For example, in Figure 3a, `incr_r1` gets the mapping: $a1 \mapsto a0$.

Group rewriting. In the final step, the pass applies local rewrites to groups based on the mapping. The simplicity of this step comes from the encapsulation property of groups—a rewriter does not have to reason about uses of a component outside the group.

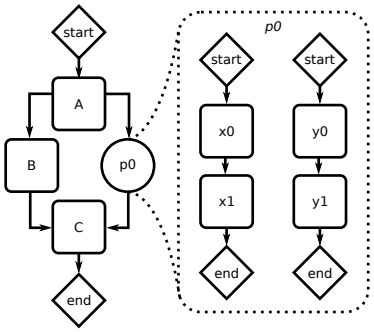
Resource sharing demonstrates Calyx's flexibility in analysis and transformation—passes can recover control flow information from the schedule and use groups to perform local reasoning.

5.2 Register Sharing via Live-Range Analysis

Group-local reasoning is insufficient for sharing stateful elements such as registers; writes to a register in one group are visible in other groups. To enable register sharing, we implement a *live-range analysis* that, for each register, determines the last group in the execution schedule to read from it. Since the register is guaranteed to never be used afterwards, subsequent groups can reuse the register. Live-range analysis is common in software compilers but is infeasible in RTL languages since the control flow of the program is not explicit.

```
seq {
  A;
  if cond.out with G {
    B;
  } else {
    par {
      seq { x0; x1; }
      seq { y0; y1; }}}
  C;
}
```

(a) Calyx program.



(b) A visual representation of a pCFG.

Figure 4: A Calyx program along with the corresponding parallel control flow graph (pCFG).

The live-range analysis has to contend with two problems: (1) coping with the `par` blocks in the control program, and (2) inferring which groups read and write to registers.

Parallel control flow graphs. We handle `par` blocks using parallel control flow graphs (pCFGs) based on the work of Srinivasan and Wolfe [38]. Most control operators in Calyx map directly to a traditional CFG. However, `par` statements need special handling since, unlike an `if` statement which executes one of its two branches, a `par` statement executes *all* its children. While writes to a register in a conditional branch *may* be visible after the `if` statement, writes within children of `par` blocks are *always* visible after the `par` block.

Parallel CFGs introduce a new kind of node—called a *p-node*—to handle `par` blocks ($p0$ in Figure 4b). A p-node represents an entire `par` block and recursively contains a set of pCFGs representing its children. In Figure 4b the p-node has two children.

Calculating read and write sets. Calyx implements a conservative analysis pass to determine the registers that groups and p-nodes read from and write to. Both groups and p-nodes can, in general, contain complex logic, so the pass must conservatively over-approximate these sets. The read set is the set of registers a group or p-node *may* read from and the write set is the set of registers they *must* write to. The data-flow analysis uses this information to determine the range each register is alive.

Computing liveness. The pass uses a standard data-flow formulation to compute the live ranges. The only aspect that needs special handling is the children of p-nodes. For these, we set the live sets at the end of each child to be the set of live registers coming out of the p-node.

Sharing registers. The pass uses the liveness information to build a conflict graph where nodes are registers and edges denote overlapping live ranges. The pass performs greedy coloring over this graph using registers as colors and rewrites groups in a similar manner to resource sharing.

5.3 Inferring Latencies

The final optimization pass in the Calyx compiler attempts to conservatively infer the latencies of groups and components. This enables the downstream Sensitive pass (Section 4.4) to lower Calyx programs using more efficient, latency-sensitive finite state machines. Consider the following group:

```
component foo<"static"=1> { ... }
group incr {
  f.in = add.out; // f is an instance of foo.
  f.go = 1'd1;
  incr[done] = f.done;
}
```

The Calyx program specifies that the latency of the `foo` component is 1 using the `"static"` attribute. Given this information, this pass infers that latency of `incr` to be 1 as well. It follows a simple rule: if a group's done signal is equal to a components go signal, and if the component's go signal is set to 1 within the group, the latency of the group is inferred to the same as the component. Such uses of components occur in groups that simply activate one component and end their execution.

This pass is conservative and only works for simple groups. Given Calyx's design principle—that most of the time frontends generate simple groups—such passes can be extremely powerful. Furthermore, such passes can be incrementally improved by adding new rules that enables the pass to infer latencies for more groups and transparently speed up programs. Section 7.1 shows that this pass transparently improves the performance of frontend code.

6 CASE STUDIES

We built two compilers that target Calyx for our case studies. The first generates systolic arrays [19] for linear algebra computations. The second compiles Dahlia [25], an imperative programming language that uses a substructural type system to enable predictable hardware design. Our goal in both case studies is to demonstrate how Calyx makes it possible to quickly bring up good compiler implementations for specialized languages. We do not aim to beat existing commercial HLS compilers which represent decades of engineering effort.

6.1 Systolic Array Generator

Systolic arrays [19] are a class of architectures that exploit data reuse. They power the recent wave of state-of-the-art linear algebra accelerators for machine learning [10, 17]. Figure 5 shows an example systolic array. In every time step, data moves from left-to-right and top-to-bottom, while the processing elements (PEs) in the grid perform computations on the data streams. Systolic arrays can maintain a high throughput because data is reused between PEs.

However, generating a custom systolic array implementation is challenging: producing RTL directly requires generating complex custom control hardware, and systolic arrays' unique parallelism pattern can be challenging to express in HLS C++ [5, 21]. We implement a systolic array generator using Calyx in only 239 LOC of Python and approximately 40 person-hours of effort. The generator can produce arrays with arbitrary dimensions and arbitrary PEs which are implemented as Calyx components themselves.

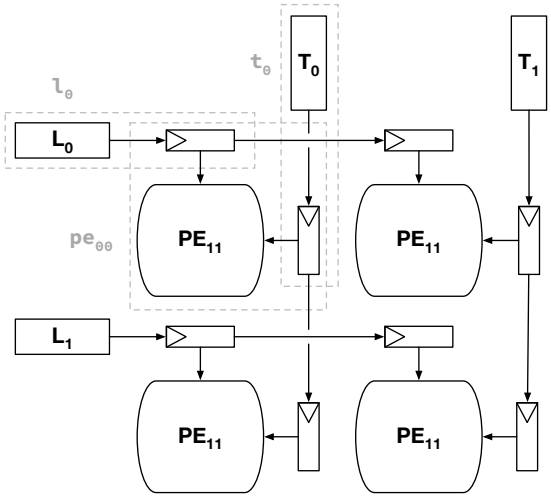


Figure 5: Architecture for a 2×2 by 2×2 matrix-multiply systolic array. Highlighted boxes show some of the groups used by the control.

```
seq {
  par { t0; l0; } // Move data from memories
  par { pe_00; }  // Run the first PE
  // Move data from memories and from registers
  par { t0; t1; l0; l1; pe_00_down; pe_00_right; }
  // Execute first PE and PEs on diagonals
  par { pe_00; pe_01; pe_10; }
  // Next step...
  par { t1; l1; down_00; down_01; right_00; right_10; }
  par { pe_01; pe_10; pe_11; }
  par { down_01; right_10; } par { pe_11; }
}
```

Figure 6: Control generated for 2x2×2x2 matrix-multiply. Execution interleaves data movement and PE execution.

Input. The systolic array generator takes the dimensions of the matrix block and a Calyx component that implements the PE. For a matrix multiply accelerator, for example, the PE consists of a multiply–accumulate (MAC) unit. It generates a systolic array that matches the dimensions of matrix block.

Architecture. Figure 5 shows the desired architecture for a 2×2 systolic array. The design consists of several groups highlighted in the figure. The groups that surround a PE implement *data movement:* the groups on the edges move the data from the input memories to registers, and the ones in the middle move the data along the fabric. Finally, the compute groups perform the computation in the PE and write their results to an internal register.

Generating Calyx. To target Calyx, the systolic array generator needs to (1) instantiate PEs, (2) create the relevant groups, and (3) define the control for the systolic array. The compiler performs (1) and (2) using templates. For each PE, the compiler also instantiates the surrounding input registers and connects them to registers in the previous PE. Finally, it defines groups to move the data and perform the computation.

The next step is generating the control. Figure 6 shows the control statements generated for a 2×2 systolic array. At each time step, the compiler enables all the data movement groups to move the data to input registers of each PE, and then in sequence, enable all the PEs with valid inputs. This schedule implements the classic systolic data flow that implements matrix multiplication which shifts the input data by one step in each dimension. The generated control accounts for invalid data and selectively enables data movement and compute groups when the input data streams start and end.

Inferring latencies. The systolic array generator does not generate any `"static"` annotations. However, the Calyx compiler is able to completely infer the latency (Section 5.3) of a generated systolic array when the processing element provides its latency. This means that the generator, by virtue of using the Calyx compiler, automatically supports both latency-sensitive and latency-insensitive systolic arrays.

Debugging with Calyx. In an initial version, the generator prematurely enabled data movement groups causing the systolic array to compute the wrong result. While debugging the kernel, it was easy to spot this mistake in the control program. This demonstrates a key quality-of-life improvement when using the Calyx infrastructure to build accelerator generators—control logic bugs can be caught by investigating the execution schedule.

6.2 The Dahlia Compiler

Dahlia [25] is a recently proposed general-purpose language for designing accelerators that resembles traditional C-based HLS. It differs from traditional HLS by adding a substructural type system that constrains the language to rule out programs that lead to inefficient hardware. The original Dahlia compiler generates C++ with annotations for the commercial Vivado HLS [44] toolchain.

In this case study, we build a new compiler for the Dahlia language that generates hardware using Calyx, eliminating the dependence on a monolithic, closed-source HLS backend and allowing greater control over the generated architecture. The goal is not to outperform the Vivado HLS backend; instead, we aim to show that Calyx makes it possible to exploit Dahlia's unique semantics to build a compiler that is far simpler than a full-fledged C-to-RTL toolchain.

Lowered Dahlia. Dahlia is a simple imperative language extended with high-level convenience features such as memory partitioning, loop unrolling, and logical array indexing. We elide the details of the first step of compilation that unrolls loops and compiles accesses to partitioned memories. We refer interested readers to our implementation.

Our explanation focuses on compiling Dahlia programs that use a small set of constructs: variables, unpartitioned memories, `while` loops, conditionals, and Dahlia's two novel composition operators: *unordered* composition (`;`) and *ordered* composition (`---`).

In Dahlia, memories and variables have an associated type and can be updated with assignment syntax:

```
let x: ubit<32> = 1; x := 2;
let arr: ubit<32>[10]; arr[1] := 3;
```

Dahlia's *unordered* composition operator allows backends to parallelize computations while preserving data flow:

```
x = 1; y = 2  // can occur in parallel
```

In contrast, Dahlia's *ordered* composition operator requires backend to execute statements in a sequence:

```
x = 1
---
x = 2
```

Ordered composition does not reason about explicit clock cycles. Instead, it imposes a partial order over the execution of program statements by reasoning about *logical timesteps*. Lowered Dahlia also supports standard imperative `while` loops and `if` conditionals.

Generating Calyx. The Calyx backend for Dahlia is a bottom-up pass that compiles each expression by instantiating groups and scheduling them using the control language.

For example, for this Dahlia program:

```
let x = 0
---
if (x > 10) { x = 1 } else { x = 2 }
```

The Calyx backend generates a group for each statement:

```
group init_x { x.in = 0; init_x[done] = x.done; }
group one { x.in = 1; one[done] = x.done; }
group two { x.in = 2; one[done] = x.done; }
group cond { gt.left = x.out; gt.right = 10; cond[done] = 1; }
```

And schedules them using the following control program:

```
seq {
  init_x;
  if gt.out with cond { one } else { two }
}
```

The Calyx backend has a one-to-one mapping for the language constructs in lowered Dahlia and the Calyx control language: memory and variable assignments generate groups representing the update, ordered composition becomes `seq`, unordered composition becomes `par`, loops and conditionals map to `while` and `if`.

Interfacing with black-box RTL. Dahlia's HLS backend uses a vendor-provided header to implement custom math functions such as square root. The HLS compiler connects definitions within such headers to black-box RTL code. In order to interact with black-box RTL components, Calyx programs can provide external definitions:

```
extern "sqrt.sv" {
  component sqrt(left: 32, right: 32, go: 1) -> (
    out: 32, done: 1
  );
}
```

External definitions do not provide an implementation; instead the Calyx compiler links in the corresponding RTL program, in this case `sqrt.sv`, during code generation. External components can be used like any other component:

```
group foo {
  sqrt0.left = 10; sqrt0.right = 20;
  sqrt0.go = !sqrt0.done ? 1;
  foo[done] = sqrt0.done
}
```

Latency annotations. Most operations in a Dahlia program have a precise latency—register writes take one cycle, multiplies take four cycles, etc. The Calyx backend uses this information to annotate the latency of each group with the `"static"` attribute. Some operations,

such as the RTL primitive to calculate the square-root, take a data-dependent number of cycles, so groups using them omit latency information. Since the Calyx compiler gracefully handles mixed latency-sensitive and latency-insensitive groups, we do not need to change anything else.

6.3 Summary

In our experience, a Calyx-based compiler requires three ingredients: (1) the abstract architecture for the domain, (2) a mapping from source constructs to Calyx constructs, and (3) a strategy to generate *groups* and *control*. For Dahlia, the architecture corresponded directly with the control language; for systolic arrays, we used a templated design with a latency-insensitive interface. In both compilers, we used groups and control to modularize and compose data flow graphs, which is not possible when generating RTL directly.

7 EVALUATION

We evaluate Calyx by generating accelerators using the frontends in the previous section and answering three questions:

- Can we build a simple compiler that generates performant specialized architectures?
- Can we use Calyx to generate reasonable hardware in a general-purpose, HLS-like domain?
- What is the effect of control-flow-sensitive optimizations implemented in the Calyx compiler?

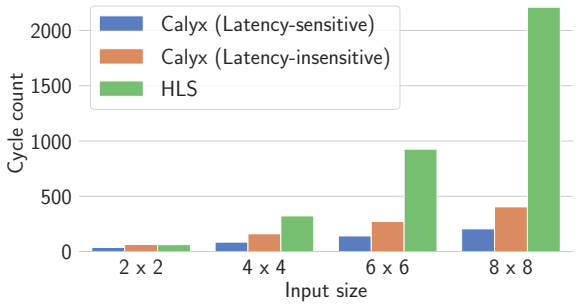
We compare Calyx-generated accelerators to Vivado HLS, a commercial HLS tool that represents decades of engineering effort. Our aim is not to beat HLS at its own game but instead achieve the same performance regime with much lower effort.

7.1 Systolic Arrays

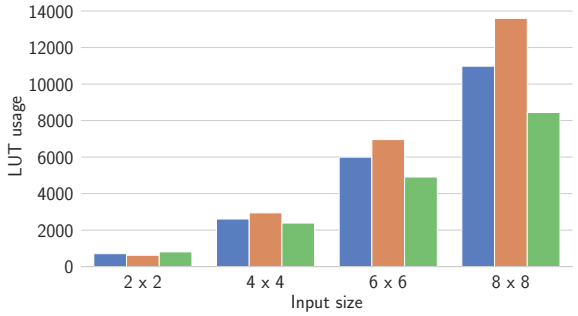
To the best of our knowledge, Vivado HLS does not automatically infer systolic arrays from loop nests. Instead, programmers need to rewrite their program to coerce the compiler into generating precisely the hardware they want. Calyx advocates for a more domain-specific approach—instead of relying on black-box compilers to infer hardware structures, design new DSLs that automatically synthesize them. We study the performance characteristics of the Calyx-based systolic array generator (Section 6.1).

Evaluation setup. We generate hardware designs for matrix multiplication kernels ranging from 2×2 to 8×8. For each configuration, we generate a systolic array using the Calyx-based generator and implement a straightforward matrix-multiply kernel in Vivado HLS that fully unrolls the outer two loops. For the Calyx designs, we collect the number of cycles by simulating the design in Verilator [39] (v4.108) and get resource estimates by synthesizing designs with Vivado [44], targeting Zynq UltraScale+ XCZU3EG FPGA at a 7ns clock period. For the HLS designs, we report the latency and resource estimates from the HLS report. We compare the cycle counts (Figure 7a) and the LUT usage (Figure 7b) of the designs. We report the characteristics of systolic arrays compiled with the Sensitive pass (Latency-sensitive) and those without (Latency-insensitive).

Comparison against HLS. Compared to HLS-based designs, Calyx-based systolic arrays are faster by a geometric mean of 4.6× and



(a) Absolute cycle counts.



(b) Absolute LUT usage.

Figure 7: Resource and cycle count comparison matrix multiply implementation HLS and as systolic arrays.

take 1.11× more LUTs. For the largest input size, the systolic array is 10.78× faster than the HLS implementation while using 1.3× more LUTs.

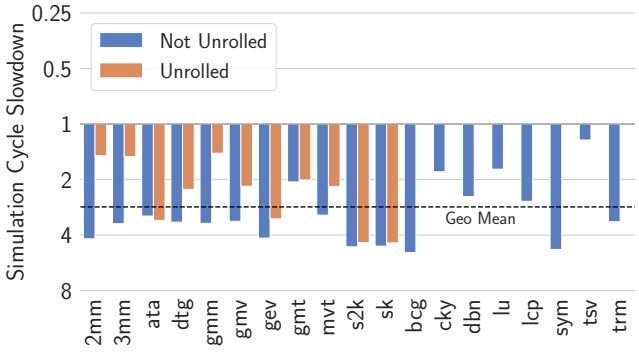
Latency-sensitive compilation. The systolic array generator does not generate any `"static"` annotations used by the Sensitive pass. It instead relies on the Calyx compiler to infer these attributes (Section 5.3). On average, Sensitive makes designs 1.9× faster and 1.1× smaller.

Discussion. Our systolic array case study demonstrates how a language designer can quickly experiment with architectural designs that are harder to express in traditional HLS tools. Without extensive engineering effort, the specialized approach can outperform a general-purpose HLS compiler.

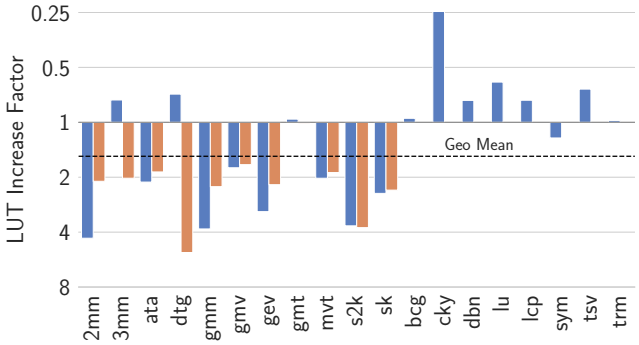
7.2 Dahlia

We built the Dahlia-to-Calyx compiler in 2011 LOC of Scala. This includes extensions to the Dahlia compiler that add passes to lower Dahlia specific constructs as well as the backend to generate Calyx from lowered Dahlia.

Evaluation setup. We compare the Calyx-generated RTL against the original Dahlia compiler [25], which emits annotated C++ and relies on Vivado HLS to generate hardware designs. We implement all 19 kernels from the linear algebra category of the PolyBench [23] benchmark suite and, for the 11 benchmarks Dahlia's type system allows it, unroll the loops to unlock parallelism. We use the same setup as in Section 7.1 to gather numbers.



(a) Cycle slowdown of Calyx designs compared to Vivado HLS. Designs below the *y*-axis are slower.



(b) LUT increase of Calyx designs over Vivado HLS. Designs below the *y*-axis are larger.

Figure 8: Resource and cycle count comparison for Dahlia-generated Calyx designs and HLS designs for PolyBench benchmarks. Missing unrolled bars indicate that the benchmark was not unrollable in Dahlia.

We also evaluate the effects of the latency-sensitive compilation (Section 4.4). We run each benchmark with the Sensitive pass enabled and disabled, following the same synthesis and measurement workflow above.

Comparison against HLS. We collected cycles counts (Figure 8a) and LUT usage (Figure 8b) for each benchmark with all optimizations turned on and normalized them to the corresponding Vivado HLS implementation. For the unrolled designs, we normalize against the corresponding unrolled HLS designs. Since DSP and BRAM usage is almost identical for all benchmarks, we elide them.

On average, the Calyx generated designs are 3.1× slower than the designs generated by Vivado HLS and use 1.2× more LUTs. For the unrolled designs, Calyx comes closer to HLS execution time, being 2.3× slower while taking 2.2× more LUTs. Vivado HLS is a heavily optimized toolchain that incorporates state-of-the-art optimizations and is designed to perform well on the kinds of nested loop nests we evaluated.

Latency-sensitive compilation. Figure 9c shows the effect of the Sensitive pass (Section 4.4) on the Dahlia-to-Calyx compiler. Enabling the optimization reduces execution time on average by 1.43× without significantly changing the resource usage.

Discussion. Despite its simplicity, the Dahlia frontend for Calyx can already generate designs that are within a few factors of the performance of a heavily optimized, commercial HLS toolchain. Part of the reason is that Dahlia is a far simpler language than C++, which makes a narrowly focused compiler tractable to build. This is the use case for Calyx—rapidly designing compilers for specialized languages and achieving good performance quickly.

We see adding traditional HLS-focused optimizations to Calyx, such as SDC scheduling [6], as the main avenue to close the gap with Vivado HLS.

7.3 Effects of Optimization

To demonstrate Calyx's ability to express control-flow based optimizations, we wrote a resource sharing pass (Section 5.1) and a register sharing pass (Section 5.2). We perform an ablation study to characterize their effects on the final designs.

Figure 9a reports the resource utilization of PolyBench benchmarks in three configurations: (1) resource sharing enabled, (2) register sharing enabled, and (3) both resource sharing and register sharing turned on. We normalize the resource counts against baselines with both passes disabled.

While both optimization passes find opportunities to share hardware components, there is not a uniform drop in the LUT usage. On average, the resource sharing pass increases LUT usage by 3% and the register sharing pass increases LUT usage by 11%. Sharing hardware components causes additional multiplexers to be instantiated which makes the resource usage worse in some cases. We plan to implement a heuristic cost model to decide which components are worth sharing (Section 9).

Figure 9b shows the effects of the register sharing pass on the number of registers used in the designs. On average, the pass reduces register usage by 12% and finds register sharing opportunities in every benchmark. Registers, compared to multiplexers, are more expensive in ASIC processes which represents another opportunity for heuristics in a future version of the Calyx compiler.

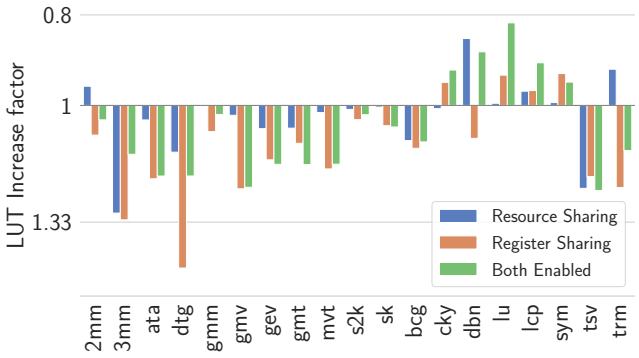
7.4 Compilation Statistics

For the largest PolyBench design (`gemver`) Calyx takes 0.06 seconds to generate RTL, compared to 26.1 seconds for the Vivado HLS compiler. The largest Calyx design is the 8×8 systolic array which contains 241 cells, 224 groups, and 1,744 control statements. The Calyx compiler generates 8,906 LOC of SystemVerilog in 0.7 seconds for this design.

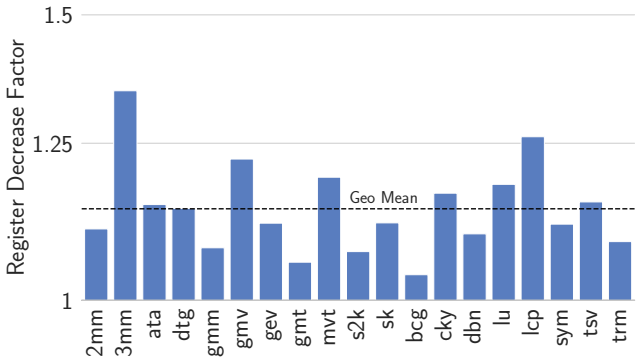
8 RELATED WORK

Intermediate representations (IRs) for hardware generation have been a topic of detailed study. Calyx differs from past work because it is not tied to a specific hardware generation methodology as in IRs for HLS compilers [3, 45], it represents programs at a higher level of abstraction than IRs for RTL design [7, 15], and it provides precise control over scheduling logic generation unlike Bluespec [26].

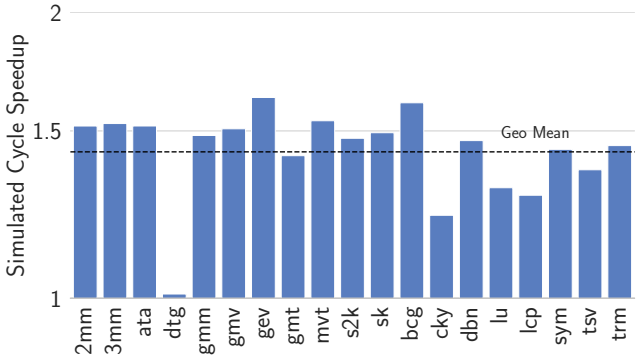
Bluespec. Bluespec [26] is an HDL that uses guarded atomic actions to enable compositional hardware design. The Bluespec compiler detects conflicts between such actions, generates a parallel execution schedule, and dynamically aborts rules on conflicts. Calyx



(a) LUT increase from resource sharing and register sharing.



(b) Register decrease from the register sharing optimization.



(c) Speedup from using latency-sensitive compilation.

Figure 9: Effects of optimization passes. All graphs use logarithmic scales.

requires no implicit dynamic scheduling; it provides explicit control over the execution schedule using its control language.

Halide. Halide [31] is an image processing DSL that pioneered the separation of algorithmic specifications from the implementation schedule to facilitate performance tuning, and follow-on work has shown how to compile Halide-like languages to hardware [13, 20, 30]. Halide schedules represent optimization strategies, such as loop tiling, that do not affect the algorithm's semantics. Calyx's concept of a schedule is different: it orchestrates and orders the invocation of hardware components and as such determines the program's semantics. Calyx's schedules are appropriate for expressing *implementations* of optimizations like loop tiling performed by high-level DSL compilers.

Software IRs. Some hardware generators repurpose software IRs such as LLVM [3, 22, 34, 45], GCC's internal IR [28], and SUIF [2]. Calyx is different from these approaches since it does not limit frontend compilers to sequential, C-like semantics. It can represent both hardware resources and fine-grained parallelism that these representations lack.

IRs for HLS. Several HLS compilers include IRs that extend their sequential input languages with representations of parallelism. μIR [35] uses a task-parallel representation, SPARK [12] targets speculation and parallelism optimizations, CIRRF [11] provides primitives for pipelining, and Wu et al. [43] propose a hierarchical CDFG representation. Calyx differs from these IRs by providing lower-level control primitives to explicitly represent hardware resources and avoiding ties to a traditional HLS setting.

Another category of HLS IRs uses finite state machines (FSMs) to model programs' execution schedules at the cycle level [9, 32, 37]. While such FSM representations are reminiscent of Calyx's control language, these IRs impose restrictions on the timing behavior of the operations inside the FSMs. Calyx imposes no such restrictions and can compose arbitrary RTL programs while providing an interface to generate optimized latency-sensitive designs when possible.

Languages with hardware parallelism. Language extensions and DSLs aim to combat the expressivity problems of HLS. They extend C with CSP-like parallelism [1], exploit software-oriented parallel interfaces in C# [36], or start with SystemC instead of plain C [24, 27]. Spatial [18] provides primitives to generate hardware from *parallel patterns* [29]. HeteroCL [20] is a Python-based DSL for optimizing programs above the HLS level of abstraction. These languages are higher level than Calyx and are not appropriate as general IRs because they are tied to specific models of parallelism. Calyx can serve as a backend for them.

IRs for HDLs. Modern HDL toolchains have IRs for transforming hardware designs [7, 15, 33, 41, 42]. These IRs work at the RTL level of abstraction and are appropriate for representing a finished hardware implementation. For generating and optimizing accelerators from DSLs, however, they have the same abstraction gap problem as any other RTL language. These IRs are potential compilation targets for Calyx.

9 FUTURE WORK

Calyx provides a useful foundation for exploring the design of higher-level DSLs, compiler optimizations, and target-specific hardware design. We plan to build upon it to explore these ideas.

First-class pipelining. Pipelines are a crucial building block for high-performance hardware designs. Calyx program encode pipelines using `while` loops and `par` blocks. However, in keeping with Calyx's philosophy of explicit control flow, we plan to design a first-class operator that will enable frontends to explicitly instantiate pipelines. An explicit representation will enable the compiler to implement pipeline-specific optimizations such as automatic buffer insertion. Higher-level control operators, such as pipelining, can

be compiled into more primitive control operators, which lets the Calyx IL and compiler incrementally add support for new operators.

Target-specific optimization. Calyx's optimization passes do not currently use cost models and other heuristics. We plan to extend the Calyx compiler to support target-specific heuristics that enable users to make different trade-offs for different targets. For example, multiplexers are cheap in ASICs but expensive in FPGAs while registers are the opposite. Such differences should affect how aggressively optimization passes that share registers are applied.

Burden of synthesizability. Several factors affect the ability of a design to meet a specific clock period: the fan-out and fan-in factors of modules, the size of the control FSM, and placement of registers in long combinational paths. Currently, Calyx requires frontends to account for these problems and generate programs that, for example, break up long combinational paths. In the future, we plan to implement passes that can analyze programs for such problems and transform them to make them synthesizable. Compiler developers can then use these passes and shift the burden of synthesizability onto the Calyx compiler.

10 CONCLUSION

The world of specialized hardware accelerator generators needs more shared infrastructure. A common representation of control and structure can enable interoperability between languages while amplifying the impact of cross-cutting optimizations, analyses, transformations, and tools.

ACKNOWLEDGMENTS

We thank Theodore Bauer and Kenneth Fang for their contributions to the implementation of the Calyx compiler. Drew Zagieboylo and Zhiru Zhang provided feedback on the design of Calyx and early drafts of the paper. Luis Vega provided invaluable help in understanding synthesis toolchains debugging RTL code generation. We also thank the anonymous reviewers and our shepherd, Sophia Shao, for their detailed feedback.

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This is also partially supported by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). We also gratefully acknowledge support from SambaNova Systems and software donations from Xilinx. Support included NSF awards #1845952 and #1723715.

A THE CALYX ARTIFACT

A.1 Abstract

Our artifact packages an environment that can be used to reproduce the figures in the paper and perform similar evaluations. It is available at the following link:

https://zenodo.org/record/4432747

It includes the following:

- `futil`: The Calyx compiler.
- `fud`: Driver for the `futil` compiler and hardware tools.
- Linear algebra PolyBench written in Dahlia.

Note on proprietary tools. We use Xilinx's Vivado and Vivado HLS tools to synthesize hardware designs and to generate HLS estimates. While trail version of these tools can be installed using Xilinx's HL WebPACK installer, their licenses for these tool disallow redistribution. Our `README.md` details installation steps for these tools.

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

- **Program:** Polybench Benchmark Suite [23]. (All benchmarks used in the evaluation are included with the artifact.)
- **Binary:** All binaries included except Vivado and Vivado HLS.
- **Run-time environment:** Rust source code can be compiled anywhere: macOS, Windows, and Linux will all work. Our evaluation scripts assume a Unix environment with the following installed:
  - `GNU Parallel 20161222`
  - `verilator v4.038`
  - `python3, pip3` and the python packages: `numpy, pandas, seaborn, matplotlib, jupyterlab`
  - `jq 1.5.1`
  - `vcdump 0.1.2`
  - `vivado v2019.2, vivado_hls v2019.2`
  - `futil, fud` from commit `dccd6f`.
  - `dahlia` from commit `978ffa`.

  Our packaged virtual machine has these tools installed.
- **Metrics:** LUT usage and simulated cycle counts.
- **Output:** The figures reported in the paper.
- **Experiments:** We provide scripts for running the experiments and use Jupyter notebook for making the figures.
- **How much disk space required (approximately)?:** 65 GB.
- **Time needed to prepare workflow?:** 4–8 hours.
- **Time needed to complete experiments?:** 4–8 hours.

A.3 Description and Installation

A.3.1 How to Access. The artifact is provided in two forms:

- A virtual image with all dependencies installed.
- Code repositories hosted on GitHub.

The instructions to download both the virtual image and the code repositories can be accessed here:

https://github.com/cucapra/calyx-evaluation

To install the proprietary tools and run the scripts, please follow the instructions in the `README.md` file at the root of the code repository.

A.4 Evaluation and Expected Results

The evaluation process aims to accomplish two goals:

- Reproduce the graphs in the paper (Figures 5 and 6).
- Demonstrate the robustness of our tooling and infrastructure.

The `README.md` file at the root of the code repository walks through the steps to reproduce the graphs from the paper, use the compiler to generate RTL code, and build on the infrastructure as a library.

Note on Figure 7a. Our original submission contained a bug in one of the plotting scripts that was caught and fixed during artifact evaluation process. Complete details are in the `README.md` instructions.

A.5 Methodology

Submission, reviewing, and badging methodology.

REFERENCES

[1] Ali E. Abdallah and John Hawkins. 2003. Formal Behavioural Synthesis of Handel-C Parallel Hardware Implementations from Functional Specifications. In *Hawaii International Conference on System Sciences (HICSS)*.

[2] C Scott Ananian. 1998. Silicon C: A Hardware Backend for SUIF. https://flex.cscott.net/SiliconC/.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[4] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2001).

[5] J. Cong and J. Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[6] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*.

[7] Ross Daly, Lenny Truong, and Pat Hanrahan. 2018. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Second Workshop on Open-Source EDA Technology (WOSET)*.

[8] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[9] Nikil D Dutt, Tedd Hadley, and Daniel D Gajski. 1991. An intermediate representation for behavioral synthesis. In *Design Automation Conference (DAC)*.

[10] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *International Symposium on Computer Architecture (ISCA)*.

[11] Zhi Guo, Betul Buyukkurt, John Cortes, Abhishek Mitra, and Walild Najjar. 2008. A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming* (2008).

[12] S Gupta, Renu Gupta, Nikil Dutt, and Alex Nicolau. 2004. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*.

[13] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics*.

[14] Intel. 2021. *Intel High Level Synthesis Compiler*. Retrieved January 16, 2021 from https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html

[15] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[16] Lana Josipoviundefined, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*.

[18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE computer* (1982).

[20] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[21] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey. 2020. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

[23] Louis-Noel Pouchet. 2021. *PolyBench/C: The Polyhedral Benchmark Suite*. Retrieved January 16, 2021 from http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[24] Mentor Graphics. 2021. *Catapult High-Level Synthesis*. Retrieved January 16, 2021 from https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[25] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[26] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Conference on Formal Methods and Models for Co-Design (MEMOCODE)*.

[27] Preeti Ranjan Panda. 2001. SystemC: A modeling platform supporting multiple design abstractions. In *International Symposium on Systems Synthesis*.

[28] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[29] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*.

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[32] Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. 2007. AHIR: A hardware intermediate representation for hardware generation from high-level programs. In *International Conference on VLSI Design (VLSID)*.

[33] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[34] Shang HLS Authors. 2021. *The Shang High-Level Synthesis Framework*. Retrieved January 16, 2021 from https://web.archive.org/web/20180610233052/https://github.com/etherzhhb/Shang

[35] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μIR: An Intermediate Representation for Transforming and Optimizing the Microarchitecture of Application Accelerators. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[36] Satnam Singh and David J. Greaves. 2008. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines (FCCM)*.

[37] Rohit Sinha and Hiren D Patel. 2012. synASM: A high-level synthesis framework with support for parallel and timed constructs. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[38] H. Srinivasan and M. Wolfe. 1992. Analyzing programs with explicit parallelism. In *Languages and Compilers for Parallel Computing*.

[39] Veripool. 2021. Verilator. https://www.veripool.org/wiki/verilator.

[40] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Symposium on SDN Research (SOSR)*.

[41] Sheng-Hong Wang, Akash Sridhar, and Jose Renau. 2019. LNAST: A language neutral intermediate representation for hardware description languages. In *Second Workshop on Open-Source EDA Technology (WOSET)*.

[42] Claire Wolf. 2021. *Yosys Manual*. Retrieved January 16, 2021 from http://www.clifford.at/yosys/files/yosys_manual.pdf

[43] Qiang Wu, Yunfeng Wang, Jinian Bian, Weimin Wu, and Hongxi Xue. 2002. A hierarchical CDFG as intermediate representation for hardware/software codesign. In *International Conference on Communications, Circuits and Systems (ICCCAS)*.

[44] Xilinx Inc. 2021. *Vivado Design Suite User Guide: High-Level Synthesis. UG902 (v2017.2) June 7, 2017*. Retrieved January 16, 2021 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf

[45] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*. 99–112.