Understanding Accelerator Compilers via Performance Profiling

AYAKA YORIHIRO✉, Cornell University, USA
GRIFFIN BERLSTEIN, Cornell University, USA
PEDRO PONTES GARCÍA, Cornell University, USA
KEVIN LAEUFER, Cornell University, USA
ADRIAN SAMPSON, Cornell University, USA

Accelerator design languages (ADLs), high-level languages that compile to hardware units, help domain experts quickly design efficient application-specific hardware. ADL compilers optimize datapaths and convert software-like control flow constructs into control paths. Such compilers are necessarily complex and often unpredictable: they must bridge the wide semantic gap between high-level semantics and cycle-level schedules, and they typically rely on advanced heuristics to optimize circuits. The resulting performance can be difficult to control, requiring guesswork to find and resolve performance problems in the generated hardware. We conjecture that ADL compilers will never be perfect: some performance unpredictability is endemic to the problem they solve.

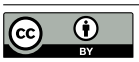
In lieu of compiler perfection, we argue for *compiler understanding tools* that give ADL programmers insight into how the compiler's decisions affect performance. We introduce Petal, a cycle-level profiler for ADLs that compile to the Calyx intermediate language (IL). Petal instruments the Calyx code with probes and then analyzes the trace from a register-transfer-level simulation. It then maps the events in the trace back to high-level control constructs in the Calyx code to determine when each construct was active. Petal processes that information into a trace of *call trees*, each representing active events in a specific cycle and their relationships. Lastly, Petal uses metadata generated by the ADL compiler to construct an ADL-level profile. Using case studies, we demonstrate that Petal's cycle-level profiles can identify performance problems in existing accelerator designs. We show that these insights can also guide developers toward optimizations that the compiler was unable to perform automatically, including a reduction by 46.9% of total cycles for one application.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Software performance**; • **Hardware** → *Hardware description languages and compilation*; *High-level and register-transfer level synthesis.*

Additional Key Words and Phrases: Intermediate Language, Accelerator Design Language, Performance Profiling, Accelerator Compiler

ACM Reference Format:
Ayaka Yorihiro, Griffin Berlstein, Pedro Pontes García, Kevin Laeufer, and Adrian Sampson. 2026. Understanding Accelerator Compilers via Performance Profiling. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 377 (October 2026), 27 pages. https://doi.org/10.1145/3839509

Authors' Contact Information: Ayaka Yorihiro, Cornell University, Ithaca, USA, ayaka@cs.cornell.edu; Griffin Berlstein, Cornell University, Ithaca, USA, griffin@cs.cornell.edu; Pedro Pontes García, Cornell University, Ithaca, USA, pp457@cornell.edu; Kevin Laeufer, Cornell University, Ithaca, USA, laeufer@cornell.edu; Adrian Sampson, Cornell University, Ithaca, USA, asampson@cs.cornell.edu.



ACM 2475-1421/2026/10-ART377
https://doi.org/10.1145/3839509

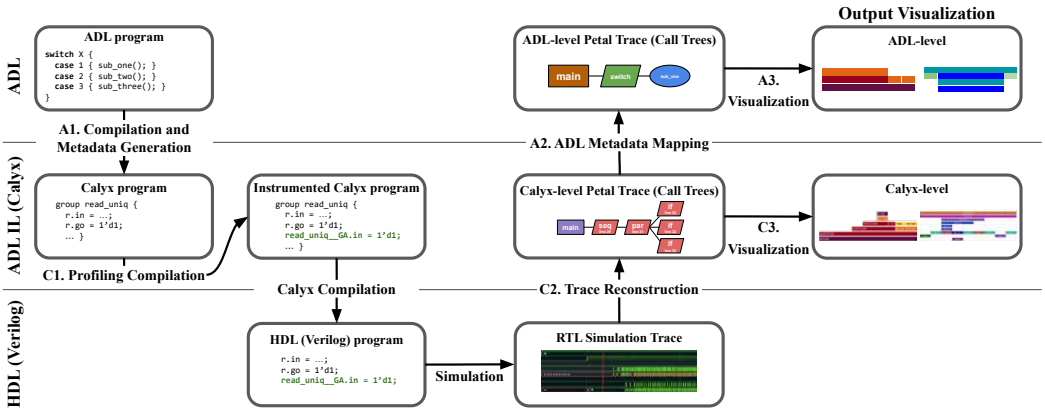


Fig. 1. Overview of Petal. Petal takes in either an ADL program or an ADL intermediate language (IL) program, and produces performance visualizations at the ADL and/or ADL IL level.

1 Introduction

Accelerator design languages (ADLs) are high-level languages for developing application-specific hardware accelerators. ADL compilers must bridge a large semantic gap: they typically start with an untimed, resource-unaware source program and introduce cycle-level schedules, physical resource management, and explicit control structures such as finite-state machine (FSM) logic.

This control-logic synthesis is fundamentally complex. ADL compilers therefore typically rely on heuristics that lead to unpredictable performance [29]. Even if the heuristics work well in most cases, in the rare cases where they produce poor results, they leave accelerator designers with little recourse. The high-level ADL source code abstracts over the implementation details for the control logic, so the only option is to inspect or simulate the generated hardware. A waveform viewer [9, 34] can help exhaustively visualize the generated control paths' behavior on every cycle, but these low-level signals may have little connection to the original ADL program, making it especially challenging to understand *when* a high-level action in the source program occurred or *how long* it took to run. They can also obscure any control overheads that the compiler itself introduces, which can manifest in wasted time when the high-level program makes no progress.

This paper argues that, because of the fundamental complexity of hardware control logic generation, ADL compilers will never achieve perfect results for every program. Instead, we must furnish developers with tools that help them understand how the compiler behaves. In particular, *trace-based profiling* for ADL programs could help illuminate how high-level constructs map onto cycle-level execution time. By revealing how the compiler chose to orchestrate the events from the source program, a profiler could help developers debug compiler-induced sources of inefficiency.

We present Petal, a multi-level cycle-counts profiler for ADLs. Petal recreates a cycle-by-cycle trace of program execution during register-transfer-level (RTL) simulation in terms of user-defined program blocks and control events. Figure 1 shows an overview of Petal. Petal operates at two levels of abstraction: the ADL level and the intermediate language (IL) level. To collect IL-level profiles, Petal instruments the code with *probes* that do not affect the program's cycle-level behavior. After obtaining a trace from RTL simulation, Petal uses the probes' values to construct a *call tree* for every cycle that identifies the active events and their relationships. Events include user-defined computation blocks and compiler-generated control constructs. Parallelism is inherent in hardware, so call *trees* can better represent execution state than traditional call *stacks*. Then, to map the profile

to the ADL level, Petal uses metadata generated by the ADL compiler. Petal uses the reconstructed event trace to produce visualizations such as flame graphs and timelines at both levels of abstraction.

Each level of abstraction that Petal produces visualizations for has a different purpose. At the IL level, Petal helps differentiate control overhead from user-defined computation. At the ADL level, Petal shows the duration of source code statements and blocks. Together, this makes Petal useful to three audiences: (1) compiler engineers for the Calyx IL seeking to generate more performant hardware, (2) compiler engineers for ADLs seeking to generate more efficient Calyx code, and (3) ADL users who want to generate hardware without much background in hardware design. Our implementation of Petal uses the Calyx IL [30] and Dahlia [29], a loop-based imperative ADL.

Our central contribution revolves around Petal's treatment of parallelism. Pervasive, fine-grained parallelism is key to the performance of ADL compilers; to the best of our knowledge, no prior work on hardware profiling has directly addressed the challenges that it presents. Prior work focuses on either associating each cycle with the statements that were active that cycle or on capturing profiles of coarse-grained program elements (functions or loops). Petal's call trees and probe-based instrumentation capture the activity of individual program elements under parallelism.

We demonstrate the usefulness of Petal's cycle-level profiles towards understanding the effects of existing optimization passes in the Calyx compiler (Section 9). By using Petal visualizations to compare optimized and unoptimized versions of a program, users can learn about scheduling changes and other transformations performed by a pass. We also find that Petal can help users identify performance problems in existing Calyx and Dahlia designs and direct their hand-optimization efforts to make the resulting accelerator more performant (Sections 10 and 11). We find that reducing control overhead can result in large performance gains, up to 46.9% in one application.

This paper's key contributions are:

- We leverage the concept of *call trees* as an extension of call stacks as an abstraction for representing parallelism-aware ADL profile data.
- We use probe-based instrumentation to distinguish concurrently executing parts of a program to correctly construct call trees from an RTL simulation trace.
- We implement the above ideas in Petal, a profiler for the Calyx intermediate language (IL).
- We extend Petal to perform ADL source-level profiling by using generated metadata to map Calyx-level performance information to ADL-level source constructs.
- We conduct two case studies that use Petal to understand the effects of two Calyx compiler optimization passes: static promotion and resource sharing.
- We describe two case studies that use Petal to find optimization opportunities by reducing control overhead, including a cycle-count reduction of 46.9% in one application.
- We perform one case study that uses Petal to find and improve cycle-level timing of user-defined computations in a Dahlia program.

2 Hardware Background and Motivation

This section motivates the problem Petal solves by describing details about ADLs and the hardware context. We (1) position ADLs with respect to other types of languages for hardware design (Section 2.1), (2) describe performance concerns in hardware (Section 2.2), and (3) discuss the semantic gap that exists within ADL programs and the generated hardware, and the performance problems induced by this semantic gap (Section 2.3).

2.1 Languages for Hardware Design

We will briefly contextualize ADLs within the landscape of languages for hardware design. Traditional hardware description languages (HDLs) such as Verilog [1], VHDL [2], and Bluespec [31] are

the dominant tools for real-world hardware design. They operate at the register-transfer level (RTL), where programmers directly control cycle-level timing and resource allocation. Hardware construction languages (HCLs) such as Chisel [7] and SpinalHDL [10] enhance HDLs with sophisticated metaprogramming layers, but they operate at the same abstraction level.

On the other hand, Accelerator Design Languages (ADLs) such as Dahlia [29], Allo [11], Spatial [23], and HeteroCL [25] operate at a higher level of abstraction that relieves the user from directly dictating cycle-level timing. A widespread form of ADL in industry is high-level synthesis (HLS) [6, 33], which is often based on C or C++. ADL programmers can express logical control flow, of which the cycle-level timing is determined by the compiler (described more in Section 2.3).

Calyx [30] is an intermediate language for ADLs that allows users to express both hardware structure and software-like control. Calyx provides a reusable compiler infrastructure for compiling ADLs to HDLs, and its use of control operators allows for software-style stepwise debugging [8].

2.2 Performance Concerns in Hardware Design

Hardware design comes with a set of key performance concerns: cycle counts, area, critical path delay, and power. *Area* means one of two things: (1) when targeting custom circuits (ASICs), the actual silicon silicon mm^2, or (2) when targeting FPGAs, the number of basic reconfigurable resources used by a design. In either case, area determines monetary cost. The *critical path delay* is the intra-cycle physical latency that limits the whole accelerator's clock speed. This dictates the maximum clock frequency because the clock period needs to be greater than the slowest logic path between two stateful elements in the design. We consider power as outside the scope of this paper.

Trade-offs. Hardware optimization is replete with trade-offs. Suppose a designer wants to parallelize two sequential computations involving a single hardware unit to reduce cycles. To do so, they would need a second copy of the hardware unit, increasing the *area* of the generated hardware. A designer might try to optimize for *cycles* by performing more computation in combinational logic. Such a program transformation may introduce a longer *critical path*, negatively affecting the clock period and slowing down the entire circuit. We include post-place-and-route area usage and clock frequency numbers whenever possible in our cycle-optimizing case studies in Sections 10 and 11.

Discussion. This work focuses on measuring clock cycles. Fortunately, area is straightforward to analyze with existing tools: most EDA toolchains can already attribute hardware resources to each component in a design's module hierarchy. Critical path delay, in contrast, is a global property that reflects the maximum latency across an entire circuit. It is therefore difficult to attribute to any specific source-code site. We see intriguing future work in developing tools to understand critical path problems, which would require an entirely different set of techniques.

2.3 ADL Compilers

ADL compilers are tasked with the complex job of converting higher-level code to semantically equivalent circuits and wires. While abstractions at the ADL level enable domain experts to design hardware, their performance behavior is challenging to understand. The constructs and operators of a high-level language bear little resemblance to the circuits and gates they compile to, rendering the source language's execution model a poor proxy for the observed circuit behavior. Furthermore, this semantic gap between source code and hardware necessitates many steps of transformation; how is an end user supposed to know if an observed performance behavior is inherent to their source code or the result of a inefficient compiler pass? Below, we discuss two major sources of performance problems induced by this semantic gap.

Problem 1: Variable Latencies. Suppose a user wants to estimate the latency of an ADL code statement. This may be difficult because hardware involves multiple, potentially unfamiliar, notions of time with each operator having many possible realizations each with different timing behaviors, some of which may not even be fixed!

Operators will have timing that is either *combinational* or measured in *cycles*. A combinational circuit element operates entirely through the propagation of values on wires, without state, and is necessarily sub-cycle in timing. In hardware languages, such propagation happens "instantly." However, combinational circuits are not free; the clock period of a circuit—and thus its speed—is directly determined by the amount of time it takes electricity to propagate along the *longest combinational path* ("critical path") in a circuit. Timing behavior for non-combinational elements is more easily understood, determined by the total number of cycles it takes to execute. Assuming an equivalent clock rate, components with smaller cycle counts execute faster.

For example, an adder can execute combinationally, while a multiplier would require a fixed number of cycles. Additionally, operators may have variable latencies, like an integer divider, or a multiplier which implements shortcuts for specific inputs such as zero or one. So, the specific number of cycles that an ADL statement spends is further obscured.

At the ADL level, Petal visualizes statements' latencies so programmers can identify expensive statements or blocks. At the IL level, it depicts the latencies of finer-grained operations, which can provide ADL users insight into the decisions made by the ADL compiler, such as the schedule it chose and where it has introduced control overhead.

Problem 2: Control register cycles. Hardware description languages describe computational elements and the wires connecting them. At this level, control flow operators do not exist. So, to impose an ordering between actions, the ADL compiler needs to produce and implement a schedule. The compiler realizes the schedule via finite state machine (FSM) logic: a register holds the current FSM state, while wires connected to operators are enabled or disabled as a function of the FSM state. Control logic costs additional cycles because updates to the FSM registers take a full cycle to write. Unless the ADL compiler exposes the full schedule, an ADL user has the means to understand neither the number of control registers required nor their overhead.

Petal visualizations at the IL level highlight cycles spent on updating control registers. Section 4 describes an example where Petal identifies a latency bottleneck from such cycles.

3 The Calyx Intermediate Language

We describe Calyx [30], the intermediate language (IL) for hardware generation that Petal targets. We will use Figure 2 as a running example.

Components. Components are encapsulations of units of hardware, similar to Verilog modules, and analogous to functions in software languages. A component has input and output *ports* to interface with other components. Each component definition has three sections: *cells* that instantiate computation elements, *wires* that connect ports, and *control* that imperatively describes the component's execution schedule. Programs begin execution at the `main` component (Figure 2, line 1).

Cells. The `cells` section instantiates other components. Each cell is an instance of either a user-defined component or a *primitive* from the Calyx standard library. In Figure 2, the `main` component instantiates six cells from the standard library (`mem`, `r`, `ans`, `eq_1`, `eq_2`, and `eq_3`) and three cells from user-defined components (`s1`, `s2`, `s3`).

Wires and assignments. The `wires` section describes assignments that connect between cell ports. Calyx assignments may contain conditional *guards* which determine when the assignment is active.

```
1  component main() -> () {
2    cells {
3      @external mem = comb_mem_d1(32, 1, 1);
4      r = std_reg(32);
5      ans = std_reg(32);
6      s1 = sub_one();
7      s2 = sub_two();
8      s3 = sub_three();
9      eq_1 = std_eq(32);
10     eq_2 = std_eq(32);
11     eq_3 = std_eq(32);
12   }
13   wires {
14     group read { // read mem into register r
15       mem.addr0 = 1'd0;
16       r.in = mem.read_data;
17       r.write_en = 1'd1;
18       read[done] = r.done;
19     }
20     group run_s1 {
21       s1.go = 1'd1;
22       run_s1[done] = s1.done;
23     }
24     ... // groups run_s2, run_s3, write
25     eq_1.left = r.out;
26     eq_1.right = 32'd1;
27     ... } // similar assignments for eq_2, eq_3
28   control {
29     seq {
30       read;
31       par {
32         if eq_1.out { run_s1; }
33         if eq_2.out { run_s2; }
34         if eq_3.out { run_s3; }
35       }
36       write; } } }
```

Fig. 2. A Calyx component implementing a `switch-case` routine, generated from Figure 3. The user-defined components `sub_one`, `sub_two`, and `sub_three` are omitted. Each `case` is checked in parallel.

Groups. A Calyx *group* is a named, unordered set of assignments that together express a logical operation. All assignments in the group are active simultaneously, and only when the group itself is active. For example, the `read` group defined on line 14 in Figure 2 writes the value in memory `mem` to the register `r`. Groups in Calyx use a go-done control interface. A group is *called* when its `go` signal is set by another group or control, and remains active as long as the `go` signal is high. In our example, the `read` group is called by the control block in line 30. A group terminates when its `done` condition is set; in our example, this occurs on line 18 to be when the register `r` finishes writing its new value. When this happens, the group's `done` signal becomes high for one cycle.

An assignment that does not belong in a group is a *continuous assignment* (e.g., Figure 2, line 25), and is always active.

Control. A component's *control* section is an imperative program that orchestrates the named groups. Our example uses an `if` block (e.g., Figure 2, line 32) to conditionally call the `run_s1` group

```
1 X = mem;
2 switch X {
3   case 1 { sub_one() }
4   case 2 { sub_two() }
5   case 3 { sub_three() } }
6 write_output();
```

Fig. 3. ADL pseudocode that uses `switch-case` to determine which subroutine to call.

if `eq_1.out` is true, a `par` block on line 31 to parallelize the case checking and execution, and sequential composition operator, `seq`, at line 29 to wrap the whole program.

The control section can also orchestrate a sub-component using a call-like operator `invoke`. For example, the `run_s1` group in our example only calls the sub-component `s1`. So, the below line would be semantically equivalent as line 32:

```
if eq1.out { invoke s1[]()(); }
```

An `invoke` statement may also name a combinational group that will be active during its execution.

Dynamic vs. Static Calyx. By default, Calyx groups are *dynamically timed* (have variable timing), meaning that the hardware needs to manage `go` and `done` ports. Calyx opportunistically adds *static* control when the cycle counts of groups are known, removing control overhead. In dynamic control, the schedule needs to wait for the `done` port to fire, then take a cycle to update the FSM (as described in Section 2.3). But static control allows for compacted schedules: the schedule can start the next computation after counting the group's specified number of cycles. The Calyx compiler contains optimization passes to infer latencies of dynamic groups and "promote" them to static whenever possible [21]. In Section 9.1, we study the effects of this static promotion using Petal.

Converting Control to Structure. After optimization, the CompileControl pass performs a bottom-up traversal which converts control blocks to structural *compilation groups* that manage state through registers [30]. There are two kinds of compilation groups: (1) `seq` groups, which encode sequential control (`seq`, `if`, and `while` statements), and `par` groups, which encode `par` statements.

A sequential control block is encoded into a *finite state machine* (FSM), where states contain group calls (including other compilation groups), condition checks (for `if` and `while` blocks), and a special done state. The compilation group manages the FSM state using an `fsm` register. Groups within the control block are called from the `seq` group with a guard that checks the `fsm` register's output value. When the groups terminate, the `fsm` register proceeds to the next state. The `seq` group terminates when the `fsm` reaches its final state.

A `par` group implements parallelism by calling all child groups (including other compilation groups) that are inside the original `par`, and finishing once all of the groups signal `done` once. Since child groups may finish at different points in time, each child group is assigned a 1-bit `pd` register for the `par` group to store the child's done status. A `pd` register is set to 1 when a child group is done. The `par` group's `done` condition is the conjunction of these `pd` registers.

4 Example: Performance Debugging with Petal

In Figure 3, ADL code uses a `switch-case` to determine which subroutine to call. Figure 2 shows the same program after compilation to Calyx. The `control` block on line 28 shows that the `switch-case` was implemented using parallel composition (lines 31–35). Each thread runs the comparison for a specific `case` and the corresponding subcomponent on success. Suppose that in this execution, `case` 1 passes. In total, the computation in the `main` component during our execution will consist of the groups `read`, `run_s1`, and `write`.

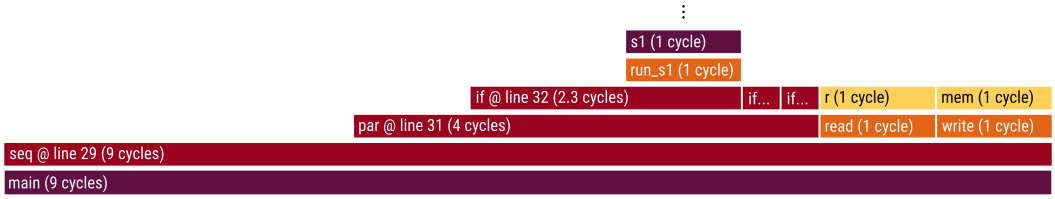


Fig. 4. Flame graph from running Petal on the program in Figure 2. Elements that occur in parallel are normalized relative to the total length, so that summing up all threads result in the total length of the `par` block. Elements within the `s1` cell are omitted for space.

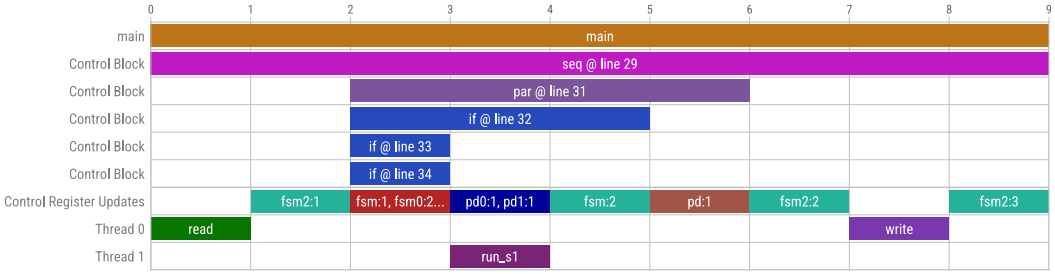


Fig. 5. A timeline view from running Petal on the program in Figure 2. Elements within the `s1` cell are omitted.

How many cycles should this execution take? Supposing each group takes one cycle, we may expect a total of three cycles. However, through simulation we find that the execution actually takes *nine*.

We diagnose this discrepancy using Petal. Petal produces a flame graph [17] (Figure 4) that summarizes the execution cycles spent on cells, groups, and control blocks. The flame graph shows the total number of cycles that each "call stack" was active. Starting from the bottommost `main` component, the y axis indicates a call stack. Each box on the graph is either a cell, group, or control statement along the stack, and its *width* along the x axis is the total number of cycles that the particular stack was active for. Cells, groups, and control statements that are on the same level of the stack are ordered *alphabetically* on the x axis, not based on the passage of time. Viewing this flame graph, we (the ADL users) can see that `read`, `run_s1`, and `write` each take one cycle as expected. But we additionally see that individual control statements (in red) take up a total of six cycles. The gaps that exist between control statements and groups are *control overhead*—cycles spent *only* on control and not on user-defined computation. This suggests that some part of our control program introduces unexpected overhead.

To get a closer look at the control overhead, we turn to Petal's timeline view (Figure 5). From the top, the timeline view shows the whole `main` component (row "main"), each control block (marked "Control Block"), updates to registers that manage control (marked "Control Register Updates"), and two threads of Calyx groups.

These control register updates are the source of our mysterious overhead. While a user previously may have assumed that `par` blocks do not incur any overhead, we see that the compiler's structural implementation of the inner `if` on line 32 adds two control update cycles (cycle 2 and cycle 4), and the `par` results in adding another control update cycle (cycle 5). None of cycles 2, 4, and 5 contain any user-defined computation; they exclusively manage control flow. This `par` overhead seems unnecessary for implementing a `switch-case` since the cases are mutually exclusive.

```
if eq_1.out { run_s1; }
else {
  if eq_2.out { run_s2; }
  else {
    if eq_3.out { run_s3; } } }
```

Fig. 6. The control of a Calyx implementation of `switch-case` that uses nested `if-else` blocks. This control leads to a three-cycle reduction by eliminating the control overhead of managing `par`s.

Perhaps parallelism is not the most cycle-efficient way to implement a `switch-case` because of the control overhead. We attempt a different implementation of `switch-case` in Figure 6, where we use a nested series of conditionals. Running this new version of the Calyx program, we find that it takes six cycles for all three `case`s. This change eliminates the three-cycle control overhead created by the `par` and the `if`s within it. So, we updated the ADL compiler to replace the `par` implementation of `switch-case` with the nested `if` alternative.

Petal provides precise cycle attribution for user-defined computation and control overhead. The gaps in the flame graph show a high-level view of control overhead, and the timeline gives a precise walkthrough of *when* computation and control updates occur. We find in Section 10.2.3 that this optimization of compiling ADL `switch-case`s into Calyx nested `if`s can reduce cycle counts by 17.7% in a larger program. A better understanding of the compiler's behavior, through Petal, led directly to an optimization.

5 Trace-Based Profiling for Accelerator Designs

A profiler needs to provide actionable feedback to the developer by mapping raw performance data to the structure of the source program. When profiling software, this often corresponds to sampling the call stack and presenting execution time in terms of which functions were active. However, a simple call stack is not enough to represent the execution of high-level programs that compile to hardware. Unlike software targeting CPUs, the hardware languages we seek to profile are inherently parallel. So, we need to be able to represent this fine-grained parallelism. We thus use the concept of call trees as an extension of call stacks and highlight the challenges that need to be overcome to collect detailed call tree information.

We start from a cycle-by-cycle RTL simulation trace of all signals in the circuit. From this we construct a *Petal trace*, which we define as a sequence of call trees, i.e., one call tree for every cycle. A *call tree* is a directed graph in which nodes are instances of user-defined component cells (which we will call "cells" for brevity), groups, primitives, or control blocks. An edge exists from node `n1` to node `n2` if `n1` called `n2`, and branches correspond to parallelism. Call trees have special rules:

- The root of any call tree is the `main` cell.
- A (non-`main`) cell node's parent can only be a group node.
- A control block node's parent must be a cell node or another control block node.
- A primitive node's parent must be a group node.
- Primitive nodes are always leaves.

Figure 7 shows call trees of cycles 0–3 from executing the example code in Figure 2. Rectangles represent cells from user-defined components, parallelograms represent control blocks, and ovals represent groups. In Cycle 2, we observe that the `par` on line 31 contains three branches—one for each active thread. However, in Cycle 3, the `if` statements on lines 33 and 34 terminate, leaving the `par` with one active child thread.

We use Petal traces to produce our two main visualizations. Flame graphs are generated by traversing the trace and identifying the number of cycles that a unique call tree path (a call stack)

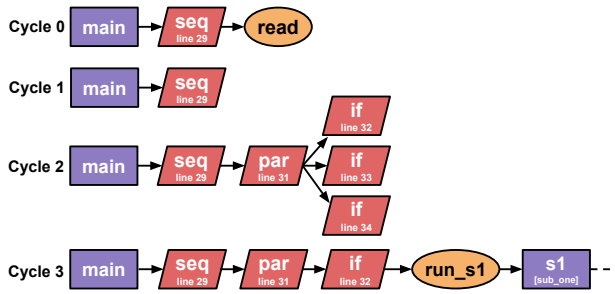


Fig. 7. Call trees from cycles 0–3 of the example program in Figure 2. Descendants of `s1 [sub_one]` are omitted.

appears in. Timelines are generated by traversing the trace to identify each node's start and end time. Paths between a cell node and group nodes within its corresponding component are used to organize the call hierarchy.

Software profilers can sample a call stack by examining the state of memory at any given point in the program. In hardware, this corresponds to observing the control signals that describe which components are active. But FSM states and go/done port values in the trace do not contain enough information to reproduce accurate cycle counts of Calyx constructs (Section 7). We add information to the *RTL simulation trace* using instrumentation and analyze the source code to add information about the *program structure*. These two additional sources are enough to "fill in the gaps" that cannot be obtained just by looking at the state of the control logic.

6 Profiling Mechanism

Figure 1 shows Petal's overall workflow. In this section and Section 7 we will focus on the phases that let us produce profiles at the Calyx-level. There are three key phases (C1, C2, C3 from Figure 1). The *Profiling Compilation* phase (Section 6.1) instruments a Calyx program to add profiling probes. Profiling probes track group activity during simulation without affecting timing. The Calyx compiler produces Verilog code, which we simulate with a standard RTL simulator to produce a signal trace. The *Trace Reconstruction* phase (Section 6.2) constructs the Petal trace from the RTL trace and control information obtained from the compiler (Section 5). The *Visualization* phase (Section 6.3) produces flame graphs, the timeline view, and statistics tables from the Petal trace.

6.1 Profiling Compilation

In the profiling compilation phase, Petal uses a special compiler flow to instrument the program with profiling probes and record metadata about the structure of the program.

Instrumentation. We add an Instrumentation compiler pass that instruments each group in the program with *profiling probes* that track when the group is active and when the group calls another group/cell/primitive. Profiling probes are implemented as wires, which are combinational; hence they do not affect the program's cycle-level timing.

There are four types of profiling probes:

(1) *Group active* (`GA`): High when a group is active. Contains the group name and the component name.
(2) *Group calls group* (`CG`): High when a group directly calls another group. Contains the parent group name, the child group name, and the component name.

```
1 group myMult {
2   ... // mult primitive multiplies two numbers
3   mult.go = !mult.done ? 1'd1;
4   r.in = mult.out;
5   r.go = mult.done ? 1'd1; // r can go after mult
6   myMult__main__GA.in = 1'd1; // group myMult is active
7   mult__myMult__main__CP.in = !mult.done ? 1'd1; // myMult calls primitive mult
8   r__myMult__main__CP.in = mult.done ? 1'd1; // myMult calls primitive r
9   myMult[done] = r.done; } // group is done after r
```

Fig. 8. A group containing multiplication instrumented with probes. Bold lines contain probe assignments.

(3) *Group calls cell* (CC): High when a group calls a non-primitive cell by via the go port. Contains the cell name, the group name, and the component name.
(4) *Group calls primitive* (CP): High when a group calls a primitive cell. Contains the primitive cell name, the group name, and the component name.

We will refer to CG, CC, CP probes collectively as *call probes*, as they encode parent-child relationships. GA probes are active during the entire execution of the group. Call probes are only active when the call assignment is active.

For example, Figure 8 shows a group myMult in component main instrumented with probe assignments (in bold). The group performs multiplication using the multiplier primitive mult, and then stores the result into the register primitive r. The instrumentation pass sets a GA probe high through the scope of the group (line 6). mult's go is high as long as its done port is not set high (line 3), which its corresponding CP probe reflects (line 7). Similarly, r is called when mult is done (line 8), which the CP probe also reflects (line 8).

For each component, the Instrumentation pass iterates through all groups. Within a group, the pass first creates a GA probe assignment to high. Then, it scans all assignments within the group; if there is an assignment to a go port of a cell, group, or primitive, the pass creates a corresponding call probe (CC, CG, CP). The right-hand side of a call probe assignment copies the right-hand side of the corresponding go port. Each profiling probe is given a special annotation to prevent compiler optimizations from removing it. If the group is a combinational group associated with an invoke in control, the Instrumentation pass adds a CC probe to track that the cell being invoked is active.

Profiling probes provide accurate cycle counts and parent-child relationships between non-control block Calyx constructs. We know that the main cell is active through the entire execution. Because of call tree rules (Section 5), groups are the only nodes that can call cells or primitives. Thus, CC and CP probes suffice to capture any other cell or primitive activation. Groups can be called from other groups or directly from the cell's control. Groups that are called from other groups are tracked with CG probes; any groups without a corresponding CG probe (i.e., the set difference between groups with active GA probes and those that are children denoted by CG probes) are called from control blocks.

Control structure metadata. To produce a faithful representation, Petal must recreate the structure of the original Calyx program's control prior to optimization. To do so, we need to overcome two challenges. First, we must distinguish between two calls of the same group at two different points in control. Second, obtaining active durations and parent-child relationships for control blocks requires more care as they cannot be directly instrumented, and they ultimately get transformed into compilation groups by the CompileControl pass (Section 3). We can use a compilation group's go/done ports to find its duration, but we need to identify the original control block. Optimization passes prior to the CompileControl pass may also transform control blocks.

To address both challenges, we add a compilation pass EncodeControl which runs before Instrumentation. For the first challenge, EncodeControl creates a uniquely named group for each group call from control. EncodeControl also creates a uniquely named combinational group to associate with every cell `invoke` in control. Then, for the second challenge, the pass assigns a unique identifier to each control block. We use Calyx's generic facility for attaching arbitrary metadata to control blocks to preserve this identifier. We modify optimization passes to preserve this metadata through transformations. Lastly, we modify the CompileControl pass to use the identifier to emit a map between the created compilation group and its original control block.

6.2 Trace Reconstruction

Petal uses a standard RTL simulator, such as Verilator [39], to run the instrumented program and record a trace containing all profiler probe signals. In the trace reconstruction phase, we read profiler probe signals from the RTL simulation trace to reconstruct the *Petal trace* consisting of call trees, which we use to produce visualizations (Section 6.3).

Building the Petal trace. To reconstruct the Petal trace, Petal must create a call tree for every cycle of the program by parsing the RTL simulation trace. We start by enumerating all of the possible nodes (cells, control blocks, groups, and primitives). The objective is to add edges that are active within the cycle (obtained from active profiling probes and compilation groups), which would result in a tree rooted at the vertex for the `main` cell.

First, Petal uses call probe signals to add edges between cell, group, and primitive nodes. For example, if the `CP` probe (`mult__myMult__main__CP`) on line 7 of Figure 8 is active, we would add an edge from the `myMult` group node to the `mult` primitive node. If there are any groups that are active (found from `GA` probe signals) but do not have an incoming edge after adding in all edges from call probe signals, they must be a root group of the cell. We add edges for these groups from their corresponding cells.

Next, Petal adds edges connecting active control block nodes into the tree. For each active cell, we construct a *tree fragment* containing all the active control blocks within the cell—derived from active compilation groups. We begin construction by enumerating the cell's control blocks and selecting those that are active. Next, we connect edges between active control blocks based on the control program hierarchy. Finally, we insert the new fragment into the larger tree.

6.3 Visualization

In the *visualization* phase, Petal uses Petal traces to produce three kinds of visualizations: (1) flame graphs, (2) timeline views, and (3) statistics tables.

The flame graph [17] gives a summary containing total cycles that were spent on each call tree node. Wide boxes take up a lot of cycles and are likely bottlenecks. Boxes in the flame graph are colored based on the call tree node type: purple is cells, red is control blocks, orange is groups, and yellow is primitives. *Gaps* on the flame graph between the widths of an entry box and the aggregate width of its children boxes often signify a large control overhead.

To further gain an understanding of the "flow" of the program, users can refer to timeline views. Timeline views are visualized using the Perfetto UI [32] visualization tool. Each cell gets its own track, which contains child tracks for control blocks, control register updates, and regular groups. Groups are assigned a thread ID based on which (if any) `par` arm they appear under. Pinning control blocks and control register updates while traversing the timeline can help users identify root causes of gaps found in the flame graph. To add control register updates to the timeline view, Petal identifies control registers in the RTL simulation trace and tracks their signal changes.

Statistics tables help quantify bottlenecks. *Cell tables* quantify the duration of each cell activation and the control overhead within them. *Group tables* display the number of times groups were active, and the minimum, maximum, and average duration of the group across all calls (Section 10.1).

7 Implementation

We implement a prototype of the core machinery of Petal in 4,931 lines of Rust. *Trace Reconstruction* uses the `Wellen` [34] library to parse and process VCD traces, and *Visualization* uses Perfetto [32] and Gregg's original SVG-based flame graph visualization tool [17].

Petal profiling performance. We briefly discuss the performance information of Petal with respect to the five programs used in case studies (Sections 9–11). All numbers are reported from runs on a server with 512 GB of RAM, dual-socket Xeon Gold 6230 processors for a total of 20 cores at 2.10 GHz, running Ubuntu 20.04 LTS. Most programs had 204–306 profiling probes inserted, except the feed-forward neural network (FFNN) program described in Section 10.1 which had 2830 probes. In simulation, there are two sources of overhead: (1) generating an RTL trace, and (2) instrumenting profiling probes in the original program. RTL tracing during simulation adds 1.2 to 3.1× overhead (compared to running a non-RTL tracing simulation) through the five programs. Compared to RTL tracing the original program, RTL tracing the instrumented program adds a negligible overhead for all five programs. For all non-FFNN programs, simulating the instrumented program with RTL tracing took less than 5 seconds, and the FFNN program took slightly over 1 minute. In the FFNN program, *trace reconstruction* took 36% less time than simulating the original program. For all non-FFNN programs, trace reconstruction had a 2-10x overhead compared to simulating the original program. However, trace reconstruction took less than 15 seconds in all cases. The entire end-to-end profiling process took less than three minutes in all programs.

Alternative profiling techniques. Petal uses instrumentation to trace executions. At first, we attempted to avoid instrumentation by directly interpreting FSM control register values using a map between states in an FSM and the groups that they represented. However, this approach failed to distinguish control-overhead cycles from actual group execution cycles. Next, we explored using Calyx's `go`/`done` control signals to track active groups (Section 3). While this approach accurately measured groups' actual cycle counts, not all user-defined groups persist through optimization passes. Also, there is no straightforward way to use `go`/`done` signals to track parent-child relationships between groups that call other groups. Instrumentation addresses both of these problems. Petal's probes represent opaque effects from the compiler's perspective, so optimizations must preserve them along with other program behavior.

Instrumentation. We needed to ensure that probes did not affect the program's cycle-level timing. One challenge involved a corner case where a probe interfered with the live-range analysis of the resource sharing optimization through redundant use of an otherwise shareable resource. Here, a `CP` probe assignment of primitive cell `p` contained the negation of `p`'s done port (e.g., line 7 in Figure 8 uses `!mult.done` as the guard). During compilation, a pass divided the original group such that the probe assignment only ran during the execution of the primitive (e.g., the group in Figure 8 was split into a group that *only* runs the multiplier `mult` and a group that only runs the register `r`). The `p.done` in the probe guard was thus redundant, but it interfered with an existing resource sharing optimization pass. So, we added a pass that would replace `p.done` in this pattern with a constant false signal. We also prevented optimization passes from removing probes and their assignments by implementing a `@protected` attribute that prevents passes from removing otherwise-dead cells and assignments. Petal's probes are robust to all intra-component Calyx optimizations.

Generality of approach. Our profiling probes mechanism was fairly simple to implement and did not necessitate large modifications to the Calyx compiler. Thus, we believe it is applicable to other ADL compilers. This is in contrast to a solely metadata-based approach, which would require carefully modifying each compiler pass to propagate the location information. The minimum requirement for applying our approach is a compiler intermediate representation, which is unfortunately not available in most commercial HLS tools. Any IR that exposes both control (e.g., FSMs) and structure (like Calyx's RTL-like groups) should suffice.

8 Source-Level Profiling

Petal's core machinery (phases C1, C2, and C3 in Figure 1) raises performance information to the Calyx level. We can then combine this Calyx-level information with metadata generated by the ADL compiler to show performance information at the ADL source language level. This section describes Petal's mechanisms for profiling at the ADL level, which are divided into three phases (A1, A2, A3 in Figure 1). The *Compilation and Metadata Generation* (Section 8.1) phase extends the compiled Calyx with metadata on how to translate Calyx-level cycle timings to the ADL level. The *ADL Metadata Mapping* (Section 8.2) phase creates an ADL-level Petal trace using the generated metadata and the Calyx level Petal trace. Lastly, the *Visualization* phase (Section 8.3) creates flame graphs and timeline views from the ADL-level Petal trace. In this section, we demonstrate Petal's source level profiling applied to Dahlia [29], a loop-based imperative ADL.

8.1 Compilation and Metadata Generation

To generate visualizations at the ADL level, the ADL compiler needs to emit only two kinds of metadata when generating Calyx: position metadata and program hierarchy metadata.

Position metadata. Position metadata associates Calyx groups, cells, and control block nodes with the line(s) in the ADL program that generated them. Previously, Berlstein et al. [8] added source-position metadata attributes to Calyx for limited group enables in control, which mapped them to a string representation of the source line. ADL compilers could be modified to emit Calyx with these attributes to enable source-level debugging. While this string representation suffices for source-level debugging where breakpoints only need one control-program location per line, source-level profiling requires mapping every single Calyx control point to an ADL statement to correctly attribute the execution latency of each line. So, we further extend source-position metadata attributes to *all* group enables, but also for all program points in `control` and all group declarations in the `wires` section.

We exemplify in Dahlia the broader principle of source-level metadata mapping which applies to any ADL. Figure 9a contains an example Dahlia program that triples the elements in an array. Compiling this program yields the Calyx program in Figure 9b and the `sourceinfo` metadata table in Figure 9c. The `@pos` tags attached to Calyx groups and control can be cross-referenced with the `POSITIONS` and `FILES` entries in the `sourceinfo` metadata table to determine what specific file and line of Dahlia code generated the Calyx construct. The Calyx groups `let1`, `let2`, and `upd0` and their control enables all have position 0, this means that their combined latency is the latency of the Dahlia statement on line 6. `let1` reads `A[k]` and writes it to an intermediate register, `let2` connects the register's output value to the multiplier, and `upd0` writes the result to memory.

Program hierarchy metadata. Using the position metadata, Petal will deduce that line 5 will take three cycles and line 6 will take 14 cycles. However, this feedback neglects the fact that both lines are parts of a `for` loop. Providing additional latency information for conditional and loop *blocks* enables users to reason at those higher abstraction levels. To do so, we provide Petal a map that describes the line/block hierarchy. In our example, this map would say that line 6 is a descendant

```
decl A: ubit<32>[2];
decl B: ubit<32>[2];

// Triple values in array x and write it into y
for (let k: ubit<4> = 0..2) {
  B[k] := A[k] * 3;
}
```

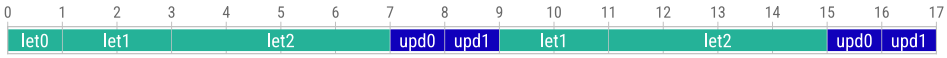
(a) Dahlia code to triple the elements of an array (`triple.fuse`).

```
wires { ...
  group let1<"promotable"=2, "pos"={0}> { ... }
  group let2<"promotable"=4, "pos"={0}> { ... }
  group upd0<"promotable"=1, "pos"={0}> { ... }
}
control {
  @pos{1} seq {
    @pos{1} let0;
    @pos{1} repeat 2 {
      @pos{1} seq {
        @pos{0} let1;
        @pos{0} let2;
        @pos{0} upd0;
        @pos{1} upd1;
      } } } }
```

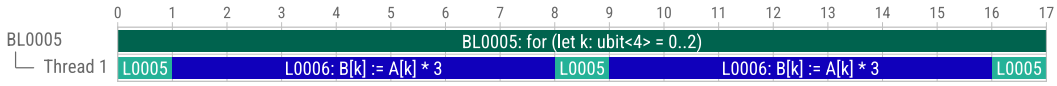
(b) Snippet of compiled Calyx.

```
sourceinfo #{
  FILES
  0: triple.fuse // fileID 0 corresponds
       to triple.fuse
  POSITIONS
  0: 0 6 // position tag 0 corresponds
       to line 6 in the file listed
       under fileID 0
  ...
}#
```

(c) Snippet of metadata produced during compilation to Calyx.



(d) Calyx timeline view.



(e) Dahlia timeline view.

Fig. 9. Example Dahlia code, metadata, and visualizations.

of line 5. Then, output visualizations will show that the *block* at line 5 is active whenever lines 5 and 6 are active.

8.2 ADL Metadata Mapping

Taking the position metadata, the program hierarchy metadata, and the Calyx-level Petal trace, Petal constructs an ADL-level Petal trace. For every cycle, Petal produces an ADL call tree by associating nodes in the Calyx call tree with the position metadata. An ADL call tree can contain a hierarchy of ADL-level control constructs consisting of more than just individual statements/lines and functions. Visualizing multiple levels of containment can help ADL users easily identify performance bottlenecks. For example, by adding Dahlia `for` loops into the Dahlia call tree, users would be able to see the whole duration of a loop, deduce the duration of an iteration, and consider the duration of each statement within the loop body. Petal uses the program hierarchy metadata (Section 8.1) to map the Calyx call tree onto a tree containing ADL-specified containment relationships.

Let us return to our example in Figure 9a. Petal constructs the Dahlia call tree for a cycle where the Calyx group `upd0` is active (e.g., cycle 7) as follows. First, Petal uses the position metadata to find that line 6 is active. Then, it uses the program hierarchy metadata to identify that line 6 is

contained in the block starting from line 5. From this information, Petal constructs a Dahlia-level call tree that consists of a single path: `main` → **`for`** block (line 5) → `B[k] := A[k] * 3;` (line 6).

Discussion of our ADL-level trace collection approach. Our approach leverages compilers' common approach towards lowering ADL statements to Calyx. ADL-to-Calyx compilers tend to create very fine-grained groups. So, instead of several ADL statements being lowered to a single Calyx group, there would be many functionally equivalent Calyx groups that each correspond to a unique ADL statement. The Calyx compiler may then perform optimizations on the program. Petal's instrumentation-based approach for constructing the Calyx-level call tree ensures that its probes remain accurate through compiler transformations.

8.3 Visualizations

Petal generates flame graphs and timeline views from the ADL-level Petal trace. Figure 9d shows the Calyx timeline view generated from an execution of the Dahlia program in Figure 9a, and Figure 9e shows the Dahlia timeline view. The Dahlia-level timeline shows the duration of the topmost block (the `for` loop on line 5) at the top, and activations of specific statements contained within that block (incrementing the loop counter `k` and the contents of line 6) are shown below it. With this Dahlia-level view, Dahlia users can use Petal to find optimization opportunities in their own source code, and they can perform optimizations without having to understand Calyx or having to manually map Calyx-level cycle information to the Dahlia level.

9 Case Studies: Understanding Compiler Choices

ADL compilers are equipped with optimization passes to improve the resulting hardware's performance. However, these passes rely on heuristics and have to optimize multiple performance metrics (cycle counts, area, and frequency). So, they may not yield optimal performance. Previously, there was no straightforward way for users to understand the effects of transformations made by optimization passes. Petal visualizations aid users by showing precise group execution times and control overheads, which we demonstrate through case studies of two compiler passes: Section 9.1 studies how the *static promotion* pass [21] optimizes cycle counts, and Section 9.2 studies how the *resource sharing* pass [30] inhibits cycle count optimizations in favor of optimizing resources.

9.1 Static Promotion

The *static promotion* optimization pass [21] converts originally dynamic groups into static groups to avoid control register update cycles. We use the Dahlia implementation of the `2mm` Polybench [26] benchmark program compiled to Calyx. To isolate the effects of static promotion, we disabled a compiler pass that splits a group using two primitives into two groups.

Specifically, we focus on the simplified code snippet shown in Listing 1. The `read_A_idx` group reads an index of memory `A` and stores the value into register `reg_a`. Similarly, the `read_B_idx` group reads an index of memory `B` and stores the value into register `reg_b`. The `mult_A_B` group performs `reg_a` × `reg_b`, storing the result into a register `res`. `write` writes the value of `res` into memory `B`. Lastly, `i_next` initializes a value for the next steps of computation, which is a loop.

Figure 10 shows a timeline of this code snippet *without* static promotion, which takes 14 cycles. In the top track, we see updates to the FSM that manage control in the snippet. The order of events is exactly as the program states, with an FSM register update after every group's execution. We find that these FSM register updates result in a five-cycle control overhead.

From a high-level understanding of the static promotion pass, a user may expect that running the pass for this snippet will straightforwardly lead to a nine-cycle execution. They may expect the resulting timeline to look like Figure 11, where groups occur immediately after another, and

the five-cycle control overhead is eliminated. However, when the program is run through static promotion, the snippet only took *six* cycles—a much larger optimization than expected!

How did we get a more optimized execution? We notice on the actual timeline (Figure 12) that there is no control register update track at the top, since static promotion was able to convert the whole snippet into static code. We also observe that the compiler inferred that there were no dependencies between groups `read_A_idx`, `read_B_idx`, and `i_next`, and scheduled them to run in parallel. By using Petal, the user can now understand scheduling effects of the static promotion compiler optimization, including parallelism between groups that they did not intentionally include.

```
seq {
  read_A_idx; // store value in reg_a
  read_B_idx; // store value in reg_b
  mult_A_B; // multiply reg_a and reg_b
  write; // write to memory
  i_next; // initializes value for loop
  ... } // loop
```

Listing 1. `control` block snippet that multiplies indices from two memories.

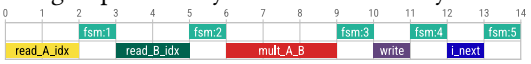


Fig. 10. Timeline of Listing 1 *without* static promotion (14 cycles).

Fig. 11. *Expected* timeline of Listing 1 with static promotion (9 cycles).

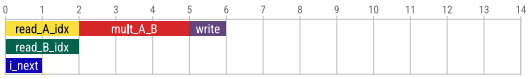


Fig. 12. *Actual* timeline of Listing 1 with static promotion (6 cycles).

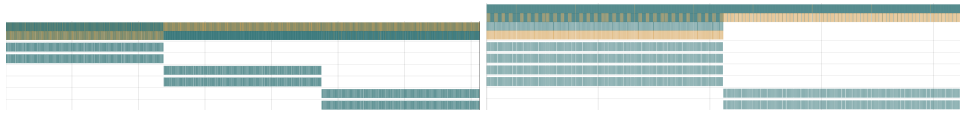
9.2 Resource Sharing

Resource sharing is an area resource optimization that reuses existing circuits to perform disjoint computations. For example, if two registers were used sequentially, the compiler could use one register to run the functionality of both. FPGAs have limited resources and the cost of building ASICs depends on the necessary resources. So, saving resources is important. However, by using a single resource, resource sharing may *disable* other cycle count optimizations.

Similar to Section 9.1, we study a Calyx implementation of the `3mm` benchmark from Polybench [26]. The `3mm` program performs matrix multiplication between matrix pairs and then multiplies the results together, yielding $(A \times B) \times (C \times D)$. We run the program under the standard Calyx optimization passes with resource sharing enabled, which takes 14,259 cycles. However, with resource sharing disabled, we get 8,481 cycles. Resource sharing *adds* a 1.7× cycle count overhead!

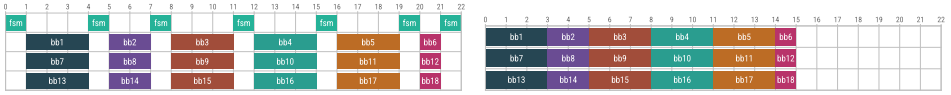
Running Petal on both versions of the program produces the timeline views in Figure 13. We can observe that the resource-sharing-disabled version of the program contains more parallelism. Additionally, the resource-sharing-enabled version of the program has three "sections" corresponding to the three matrix multiplication blocks, whereas the resource-sharing-disabled version has only two. A detailed investigation shows that in the resource-sharing-disabled version, the two initial multiplications (`A` × `B` and `C` × `D`) are run in parallel. As a result, we can conclude that resource sharing shared cells between the two initial multiplications which forced them to happen in sequence when they could be done in parallel.

At this point, it is up to the user to determine which of area or cycle counts they would like to save more, or for them to derive a more optimized version of the program that saves both. However, Petal helps the user understand attribution of cycle counts under this trade-off.



(a) Resource sharing is *enabled* (14,259 cycles). (b) Resource sharing is *disabled* (8,481 cycles).

Fig. 13. Full-program timeline views of the full `3mm` program execution, with and without resource sharing.



(a) Original program: Total 22 cycles. (b) Optimized with `static`: Total 15 cycles.

Fig. 14. Timeline views of two versions of a snippet of the FFNN program.

10 Case Studies: Optimizing Calyx User Programs

Petal helps users identify and fix performance bottlenecks, especially ones that do not directly come from user-defined computation. In the process of synthesizing control logic, ADL compilers may introduce control overhead in unexpected and "invisible" ways to the user. Petal demystifies this nebulous cost. We demonstrate this through two case studies: Section 10.1 studies optimizations we made on a feed-forward neural network accelerator; Section 10.2 walks through optimization strategies applied to a 1.4-million-cycle packet-scheduling hardware design.

Performance debugging workflow with Petal. Each of Petal's visualizations play a role in the performance debugging process. The flame graph is useful for identifying high-level discrepancies between expected and actual bottlenecks, timeline views give a detailed trace, and statistics tables quantify targets for optimization. We envision an ADL programmer using Petal by first identifying candidate performance bottlenecks through graphical (flame graphs) or numeric (cell statistics) summaries. With this insight, they can proceed to the fine-grained timeline view to understand the source of the bottleneck and devise a solution.

10.1 Feed-Forward Neural Network

Setup. We evaluate a Calyx implementation of a feed-forward neural network (FFNN) model, produced by compiling a PyTorch model via Allo [11] to Calyx. The FFNN model has an input of 64 features. It contains a fully connected layer of size 64×48, a ReLU activation, and a second fully connected layer of size 48×4. The initial implementation takes 409,715 cycles. Similarly as Section 9.1, we disabled a compiler pass that splits a group into two because it obscures the effects of the particular optimization we will be making.

Identifying the problem. Using Petal's cell overview statistics, we find that only 62% of these cycles are user-defined computation cycles, which presents to us an opportunity for optimization. Since the `main` component only exists as a wrapper, we know that `forward` is the component to optimize. To take a closer look at a potential control bottleneck, we refer to the timeline view; Figure 14a shows a close-up snippet of the timeline. In this snippet, we can see that there are multiple parallel sequences running in lockstep (for example, both `bb_1` and `bb_7` run for three cycles), with an FSM update cycle in between (one FSM for each `par` track, but all updating simultaneously). This sequence occurs in a loop in the program execution, and we confirm that the cycles where FSM updates happen do not have any other activity, marking it as control overhead.

Table 1. Group statistics obtained from the initial FFNN program. `bb_7-12`, and `bb_13-18` follow.

Group	Min	Max	Avg	Times Active	Total
bb_1	3	3	3	9,216	27,648
bb_2	2	2	2	9,216	18,432
bb_3	3	3	3	9,216	27,648
bb_4	3	3	3	9,216	27,648
bb_5	3	3	3	9,216	27,648
bb_6	1	1	1	9,216	9,216

The timeline revealed to us that the control program of our snippet uses dynamic scheduling. This control overhead is necessary because we and the compiler are not certain that groups in the snippet will have a fixed latency. However, if we can deduce that groups will always have a fixed latency, we can eliminate the control overhead by switching from dynamic to static code.

Before we refer to code to figure out whether we can convert groups to static, we can obtain evidence from Petal's group statistics table (Section 6.3). Table 1 shows group statistics for groups `bb_0` to `bb_6`. The "Min", "Max", and "Avg" columns show the minimum, maximum, and average number of cycles for which the group was active. The "Times Active" column shows the number of times the group was active, and the "Total" cycle shows the total number of cycles for which the group was active. Since the "Min", "Max", and "Avg" columns all have the same value for each row, we gain greater confidence that these groups can be converted to static, and we can eliminate the control register update cycles in between them as a result.

Solution. After confirming that each group will only be active for the stated "Avg" number of cycles, we annotate all of the groups of interest as static code. The optimized version of the program contained 287,603 cycles, yielding a 30% cycle count cut. Referring to the timeline view for this optimized execution (Figure 14b), we find that this optimization had the expected effect of removing the control overhead we previously found in Figure 14a.

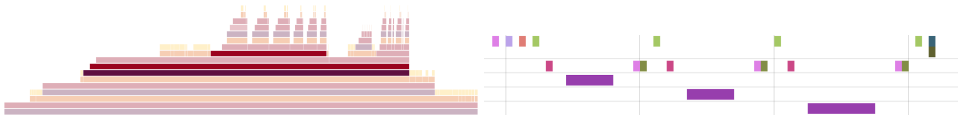
10.2 Packet Scheduling Queues

Setup. In our second case study, we profile a PIFO tree packet scheduler created by the Python-Calyx packet scheduler generator [21]. The PIFO tree decides allocations of bandwidth between multiple network flows. Specifically, the program we profile implements a *strict* policy with six input flows managed by FIFO queues, containing a total of 20,000 commands (between push and pop). The program contains a `pifo` component that manages the six FIFOs to perform push and pop commands. The program originally took 1,489,329 cycles.

In this case study, we identify multiple causes of control overhead. Section 10.2.1 discusses how we found the parallel `switch-case` inefficiency discussed in Section 4; Section 10.2.2 demonstrates how we used Petal to identify a compiler performance bug in Calyx `while` loops conditions; Section 10.2.3 summarizes how our control overhead optimization techniques led to a 698,129 cycle optimized version (an overall 46.9% cycle reduction) of the packet scheduling program.

10.2.1 `switch-case` Implementation.

Identifying the problem. Petal produces the flame graph on Figure 15a. From the lengths of the red boxes (representing control blocks), we observe that control blocks take up a significant amount of time in the execution. Furthermore, the `myqueue` cell (the bottom-most, purple block) which instantiates the `pifo` component takes up a significant portion of the total cycles spent. It also devotes a large percentage of its cycles to control register updates, as indicated by the large



(a) Flame graph of the packet scheduling program.

(b) Timeline of groups active during one call of the myqueue cell.

Fig. 15. Flame graph and timeline view snippets of the packet scheduling program.

```
1 while cond.out with comb {
2   seq {
3     ... // body
4   }
5 }
```

(a) while loop using with combinational group.

```
1 cond_group; // write cond result to
      cond_reg
2 while cond_reg.out {
3   seq {
4     ... // body
5     cond_group; // write cond result to
          cond_reg
6   }
7 }
```

(b) Compiler implementation of the while block.

Fig. 16. Two versions of while loops in Calyx.

horizontal gaps. The two control node blocks in bold are par blocks, which suggests that the pifo component involves nested parallelism. We decide to attempt optimizing the pifo component.

For a closer look, we refer to the timeline view in Figure 15b, which shows us the groups that were active in one call of myqueue. The shape of the timeline shows that, despite the program's nested parallelism, the groups actually run *sequentially* (except for the compiler-optimized parallelism on the first row). Scheduling parallelism incurs overhead via pd register updates, so there may be an optimization opportunity in the nested par blocks. By inspecting the Python-Calyx packet scheduler source code, we find that these par blocks come from the implementation of switch-case in the Python-Calyx generator. We discover that the switch-case implementation in the Python-Calyx generator adds unnecessary control register update cycles, as described in Section 4.

Solution. As described in Section 4, we changed the implementation of switch-case in the Python-Calyx generator to use nested if statements instead of parallelism. The resulting program took 1,224,589 cycles because switch-case was used in multiple cells. Thus, a small change inspired by Petal made an overall 17.7% cut on cycle counts.

10.2.2 While Loop Conditions. We describe a code pattern (while loop conditions) exhibiting unnecessary control overhead that occurred in multiple contexts in the packet scheduling program. We present a simplified example of this pattern.

while loop semantics in Calyx allow for the with keyword with a combinational group that computes the condition. Figure 16a shows a while loop using a combinational group comb which contains assignments to the cond cell. The with keyword and combinational group is implemented by the compiler as Figure 16b. Here, the cond_group group (non-combinational) computes the condition and writes to the register cond_reg. cond_group is called both before the while loop, and in the last step of the body.

```
1 seq {
2   init; // initialize counter
3   while lt.out with cond {
        // counter == bound?
4     par {
5       seq { read; update;
              write; }
6       incr; // increment
              counter
7     } } }
```

(a) Original `while` using a combinational group.

```
1 seq {
2   init; // initialize counter
3   cond_group; // write cond
        result to cond_reg
4   while cond_reg.out {
5     seq {
6       par {
7         seq { read; update;
                write; }
8         incr; // increment
                counter
9       }
10      cond_group; } } }
```

(b) `while` without combinational group.

```
1 seq {
2   init; // initialize counter
3   cond_group; // write cond
        result to cond_reg
4   while cond_reg.out {
5     par {
6       seq { read; update;
              write; }
7       seq { incr; cond_group;
              } // increment and
                check counter
8     } } }
```

(c) Parallel optimization of incrementing and checking the group.

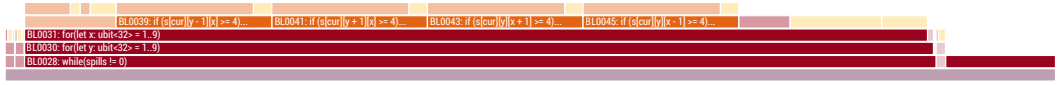
Fig. 17. Example `control` programs of `while` with conditions pattern and optimizations.

Setup. The Calyx `control` block in Figure 17a updates a computed number in memory a specified number of times. The `while` block on line 3 drives the combinational group `cond`, which updates the `lt` cell's values based on whether we have reached the loop bound.

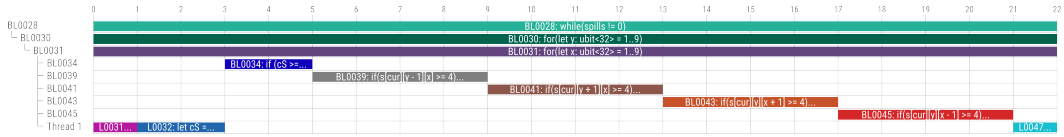
Identifying the problem. When we run this program through Petal (on inputs where the `while` loop runs for three iterations before terminating), we obtain the timeline on Figure 18. First, we note that every call of `cond` takes one cycle. This might be insightful for users who expected `cond` to be purely combinational. Additionally, the timeline shows that there are *three* FSM register updates (boxes on the top row): (1) on cycle 2 between `init` and `cond`, (2) between `cond` and the beginning of the `while` loop, and (3) after the end of the `while` loop. (2) and (3) are necessary register update cycles because `while` loops cannot be made static due to their fundamental dynamic nature. However, (1) seems redundant since both `init` and `cond` take one cycle each.

Solution. We manually transformed the code (Figure 17b) to not use a combinational component in an attempt to remove (1), which resulted in a one-cycle cutoff. Additionally, we notice that we could leverage the parallelism at play by performing the counter check in the same thread as the counter increment. Figure 17c shows the result, reducing three more cycles as expected.

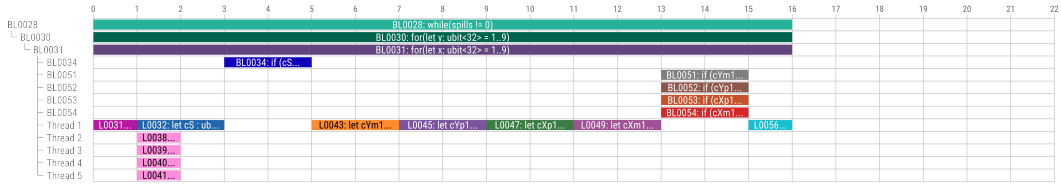
10.2.3 Summary. We apply the three control overhead reduction strategies previously explored—converting code to static (Section 10.1); fixing `switch-case` implementation (Sections 4 and 10.2.1); transforming `while` loop conditions (Section 10.2.2)— to the packet scheduling program. Table 2 shows the effects of each optimization strategy. By employing hand-optimization strategies inspired by profiler feedback, we reduced a total 46.9% of cycles. Additionally, we used Vivado [40] and targeted the Zynq UltraScale+ XCZU3EG board to obtain post-place-and-route resource and critical path estimates for both original and optimized versions of the packet scheduling program. We found that the two versions of the program had a similar critical path and can meet a frequency of 143 MHz, and area *decreased*, where the total number of LUTs went from 1,251 to 873. We conclude that our final design optimizes both cycle counts and area, and has a comparable critical path.



(a) Flame graph of original sandpile program.



(b) Timeline view of one iteration of the inner for loop in the original program (22 cycles).



(c) Timeline view of one iteration of the inner for loop after performing the optimization (16 cycles).

Fig. 21. Visualizations of the Dahlia sandpile program by Petal.

Fig. 18. Original `while` program (17 cycles).

Fig. 19. Manually transformed `while` program (16 cycles).

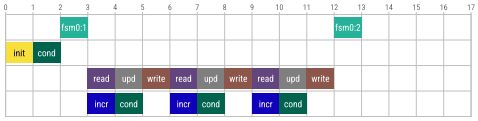


Fig. 20. `par` optimized `while` program (13 cycles).

Table 2. Various optimization strategies and the cycle reductions they made on the packet scheduling program.

Strategy	Cycles reduced	% Cycles reduced
`switch-case`	264,740	17.7%
`while`	91,026	6.1 %
`static`	332,928	22.4 %
Other	9,435	0.6 %
Total	698,129	46.9%

11 Case Studies: Optimizing ADL programs

Petal can help ADL users identify performance bottlenecks in their ADL code, akin to software profilers. We demonstrate this through a case study optimizing a Dahlia program.

Setup. In our case study, we profile a Dahlia program that computes the Abelian sandpile model [27]. The sandpile model is a self-organized critical system where grains of sand accumulate on a grid until some threshold is reached; when this threshold is reached on a vertex on the grid, the grains of sand "spill" into neighboring vertices. The initial implementation takes 45,536 cycles.

```
32 let cS = s[cur][y][x];
33 let nS : ubit<32>;
34 if (cS >= 4) {
35   nS := cS - 4;
36   sp += (1 as ubit<32>);
37 } else { nS := cS; }
38 --- // sequentialization operator
39 if (s[cur][y - 1][x] >= 4) { nS += 1; }
40 ---
41 if (s[cur][y + 1][x] >= 4) { nS += 1; }
42 ---
43 if (s[cur][y][x + 1] >= 4) { nS += 1; }
44 ---
45 if (s[cur][y][x - 1] >= 4) { nS += 1; }
46 ---
47 s[n][y][x] := nS;
```

(a) Original version.

```
let cS : ubit<32> = s[cur][y][x];                          32
let nS : ubit<32>;                                         33
if (cS >= 4) {                                             34
  nS := cS - 4;                                            35
  sp += (1 as ubit<32>);                                   36
} else { nS := cS; }                                       37
let nS1 : ubit<32> = 0; // temp var to store               38
     updates to s
let nS2 : ubit<32> = 0;                                    39
let nS3 : ubit<32> = 0;                                    40
let nS4 : ubit<32> = 0;                                    41
---                                                        42
let cYm1 = s[cur][y - 1][x];     // Sequential             43
     reads required
---                                                        44
let cYp1 = s[cur][y + 1][x];                               45
---                                                        46
let cXp1 = s[cur][y][x + 1];                               47
---                                                        48
let cXm1 = s[cur][y][x - 1];                               49
---                                                        50
if (cYm1 >= 4) { nS1 += 1; }                               51
if (cYp1 >= 4) { nS2 += 1; }                               52
if (cXp1 >= 4) { nS3 += 1; }                               53
if (cXm1 >= 4) { nS4 += 1; }                               54
---                                                        55
s[n][y][x] := nS + nS1 + nS2 + nS3 + nS4;                  56
```

(b) Cycle-optimized version that parallelizes condition checking and writing to intermediate variables.

Fig. 22. Code in innermost `for` loop of the Abelian sandpile model Dahlia program.

Identifying the problem. Petal initially produces the flame graph on Figure 21a. We observe that the nested loops (the `while` on line 28, the `for` on line 30, and the `for` on line 31) clearly occupy a large proportion of the flame graph. We also observe that there are four `if`s within the innermost loop which also occupy a relatively large number of cycles. With this in mind, we go to the timeline view and zoom into one iteration of the innermost `for` loop. An iteration takes 22 cycles; the code is shown in Figure 22a, and the timeline view of one iteration is shown in Figure 21b.

The timeline view shows that `if` blocks are run sequentially and each block takes four cycles. We find from inspecting the source code that the condition and bodies of the `if`s are not dependent on each other. So, we decide to parallelize the code in order to optimize for cycle counts.

Solution. Each of the `if` blocks have a similar routine: (1) read an index from the shared array `s`, (2) check whether the value is above 4, (3) update the shared variable `nS`. Ideally we could have parallelized all four `if` blocks, but Dahlia's semantics prohibits parallel array accesses. So, the reads from `s` (step 1) need to be sequentialized. However, we can parallelize steps 2 and 3 by creating new temporary variables to stand in for `nS`, which would be added to `s[n][y][x]` in line 47.

This challenge provides some insight into the complexity of ADL compilers, and how optimizing ADLs differ from optimizing software. While the Dahlia compiler intentionally restricts simultaneous accesses to the same array (`s` in this example) for the sake of predictability of the resulting hardware, this restriction isn't immediately evident from the source code itself and can invite confusion for the new Dahlia user. Additionally, our optimization strategy involves creating temporary variables, which is also a strategy that would not make much sense in software debugging. Because of the inherent parallelism in hardware, introducing a new variable often manifests in a parallel use of a new register, which won't affect the cycle-count latency.

Figure 22b shows the new body of the inner `for` loop. It creates two new intermediate variables for each `if` block. Lines 38–41 initializes intermediate variables which will store the amount to add

to `s[n][y][x]` (e.g. `nS1`). Lines 43–49 set intermediate variables to store the result of the read in step 1 (e.g. `cYm1`). The new timeline view of one iteration is shown in Figure 21c. An iteration now takes 16 cycles, since we were able to parallelize the last two cycles from each original `if` block.

The optimized version of the program contained 33,632 cycles, yielding a 26.1% cycle reduction from the original. From this case study, we can see that Petal can also be used at the ADL source language level to reduce inefficiencies introduced through the designer's code. As one may expect from the nature of this optimization, we found that the area increased: the total number of LUTs went from 780 to 990. We also found that while the maximum frequency decreased from 257.7MHz to 217.5MHz, but the end-to-end latency improved from 176,679 seconds to 154,370 seconds. We conclude that our final design trades better end-to-end latency for a slightly worse area.

12 Related Work

Software Profiling. Traditional software profiling and profiling-based performance diagnosis [16, 18, 20] help software developers identify performance bottlenecks. There is also work on profilers for specific software domains, such as high-performance computing [3, 4, 19, 24, 36]. Kremlin [15] identifies regions of serial programs that could most benefit from parallelization by using a hierarchical critical path analysis. This paper's focus on languages that compile to hardware comes with new challenges: the need to capture fine-grained parallelism in the form of call trees (Section 5) and the added complexity of reconstructing the program's execution state on each cycle.

Profiling High Level Synthesis. Preexisting work on profiling for higher-level accelerator generator languages focuses on traditional C++-based high-level synthesis (HLS) compilers. Techniques for profiling HLS programs use *simulation* or *in-FPGA execution*. In either approach, the challenge lies in identifying which source-level statements are active in each cycle. Petal takes the former approach; it uses traces from RTL simulation.

HLS_Profiler [35] combines information given by AMD Vivado HLS (now Vitis) synthesis and simulation reports to form a mapping from clock cycles to lines of C code that were active. It parses Vitis's Design Analysis Report to deduce, for each value in an FSM register, a mapping that relates RTL assignments, their predicates, and the line of source code that generated them. Then, it uses RTL traces to check predicate values to identify which assignments were active for each clock cycle. While HLS_Profiler's approach avoids instrumenting the code, it instead relies on internal details of the compiler, tying the approach to a specific version of the HLS toolchain. HLS_Profiler also does not fully address parallelism. The authors acknowledge that parallelism introduced by the compiler would make the total latency of all of the active source lines greater than the program latency. Petal's instrumentation-based approach does not rely on reading generated reports and therefore remains more agnostic to compiler details. Additionally, Petal's call trees lets it overcome the challenges with parallelism HLS_Profiler struggled with. When multiple program blocks are active in a single cycle, Petal deduces the number of active parallel threads from the call tree's paths, and normalizes the duration of each path when producing the flame graph. Another crucial difference is that Petal yields more precise profiles when source-level constructs map unintuitively onto hardware schedules, whereas HLS_Profiler aims to generate a profile that is faithful to the control flow of the original program. For example, if the generated hardware performs computation of both the `then` and `else` branches of a conditional ahead of evaluating the guard, HLS_Profiler will choose to *only* include the computation of the branch that was ultimately taken in the profile. Petal will instead choose to include both computations to reflect the scheduling choices that the compiler took and their impact on a concrete hardware execution.

A key challenge profiling using *in-FPGA execution* is adding efficient hardware to record, store, and transmit profiling data. HLScope [12, 13] obtains cycle latencies of source-level functions to

identify the causes of stalls. To do so, it (1) instruments the beginning and end of each source-level function to update a function-specific signal; and (2) adds a monitor that records functions durations to DRAM for offline analysis. RealProbe [22] collects cycle-latency profiles for source functions and loops. It uses LLVM metadata to maintain a C-to-RTL mapping. RealProbe externalizes existing control signals (analogous to `go`/`done` signals in Calyx) for functions and loops. It uses a global cycle counter and individual counters for each control signal to determine when C program blocks occur, and writes results to DRAM. Neither tool constructs a call stack or a representation of the calling context; therefore, they do not need to contend with parallelism in the execution schedule. Additionally, because they prioritize high-performance FPGA execution, both profilers capture only coarse-grained latencies for functions/modules and possibly loops. As a simulation-based profiler, Petal instead collects exhaustive data about each schedule step that can reveal statement- or expression-level performance problems. Similarly, whereas RealProbe omits tracking loop iterations after the fourth cycle to minimize overhead, Petal captures all loop iterations. Because Petal's instrumentation does not rely on existing control signals, it remains robust to compiler optimizations that can obscure FSM-based profiling, and it can faithfully reconstruct parent-child relationships in a call tree (Section 7).

Profilers that focus on commercial C/C++ HLS compilers restrict the input languages they can support and rely on specific versions of closed-source commercial infrastructure. On an engineering level, Petal targets the open-source Calyx IL and supports any ADL that compiles to Calyx. Additionally, by targeting an intermediate language, we can capture finer-grained performance information than lines, loops, and functions. This lets us present the performance impacts of choices that the ADL user *and* the ADL compiler made. The Calyx-level profiling information can inform ADL users how to better structure their code for the compiler without need for the ADL user to perform extra guesswork. Further, HLS_Profiler and HLScope only present their results in table form, and RealProbe presents a waveform visualization. Petal produces a flame graph, timeline view, and summary statistics that more closely resemble a software profiler.

Performance Visualizations. Waveform visualization tools [9, 34] help users navigate RTL simulation traces, but they only reflect RTL-level signals which are difficult to trace back to the ADL level. Commercial HLS tools [5, 6, 28] commonly provide schedule viewers that show how the source code is scheduled. However, schedule viewers use pre-simulation performance estimates instead of post-simulation RTL traces, which may result in inaccuracies. Commercial HLS tools also provide post-simulation visualizations in the form of RTL-level waveforms. Vitis HLS [6] also provides a Timeline Trace Viewer that shows latency information at the source function and loop level. Academic HLS tool Dynamatic [14] provides a post-simulation visualization that shows the synthesized dataflow circuit, which can be animated to show dataflow activity during simulation. While Dynamatic's visualization can help users understand circuit behavior, it does not provide direct feedback on which part of a program is taking the most time.

Cider [8] is a debugger and interpreter for Calyx that aids ADL users in finding correctness bugs in their code. Petal can complement Cider by helping ADL users identify performance bugs.

13 Conclusion

ADL compilers exhibit a necessary unpredictability because of the complexity of converting software idioms to hardware structure. By creating Petal and using it to find optimization opportunities, we advocate for the necessity of compiler understanding tools. Such tools can help ADL users navigate choices made by compilers, work around cases where the compiler makes sub-optimal scheduling choices, and make ADLs even more accessible and realistic for user bases.

Data-Availability Statement

The source code for Petal can be found on GitHub [38], and documentation is available online [37]. A pre-built virtual machine image with the source code of Petal, the code that we used in our case studies, and detailed instructions on how to run Petal is available on Zenodo [41].

Acknowledgments

Akash Dhiraj, Ryan Fu, and Jiahan Xie were early users of Petal, and their use identified bugs and inspired additional features of Petal. In particular, Akash contributed the queues program in Section 10.2 and Jiahan contributed the FFNN program in Section 10.1. Additionally, Nimalan's Dahlia cookbook repository [27] from which we sourced the Abelian sandpile program in Section 11 was very helpful. We thank the anonymous reviewers for their detailed feedback, as well as Benjamin Carleton and Spencer Van Koevering for generously providing feedback on drafts of the paper.

This research was supported by NSF awards #2426764 and #2118709.

References

[1] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006). doi:10.1109/IEEESTD.2006.99495

[2] 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan 2009). doi:10.1109/IEEESTD.2009.4772740

[3] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* (2010). doi:10.1002/cpe.1553

[4] Anthony Agelastos, Benjamin Allan, Jim Brandt, Ann Gentile, Sophia Lefantzi, Steve Monk, Jeff Ogden, Mahesh Rajan, and Joel Stevenson. 2016. Continuous whole-system monitoring toward rapid understanding of production HPC applications and systems. *Parallel Computing* (2016). doi:10.1016/j.parco.2016.05.009

[5] Altera. 2023. Altera High Level Synthesis compiler. www.altera.com/products/development-tools/quartus-prime/hls-compiler.

[6] AMD. 2022. Vitis HLS. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC)*. doi:10.1145/2228360.2228584

[8] Griffin Berlstein, Rachit Nigam, Christophe Gyurgyik, and Adrian Sampson. 2023. Stepwise Debugging for Hardware Accelerators. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3575693.3575717

[9] Anthony Bybell. 2021. GTKWave. http://gtkwave.sourceforge.net.

[10] Charles Papon et al. 2016. *SpinalHDL*. https://github.com/SpinalHDL

[11] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3656401

[12] Young-kyu Choi and Jason Cong. 2017. HLScope: High-level performance debugging for FPGA designs. In *Field-Programmable Custom Computing Machines (FCCM)*. IEEE. doi:10.1109/FCCM.2017.44

[13] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. doi:10.1109/ICCAD.2017.8203844

[14] Dynamatic Team. 2025. Dynamatic. https://github.com/EPFL-LAP/dynamatic

[15] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1993316.1993553

[16] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. (1982). doi:10.1145/989393.989401

[17] Brendan Gregg. 2011. *Flame Graphs*. https://www.brendangregg.com/flamegraphs.html

[18] Matthias Hauswirth, Amer Diwan, Peter F Sweeney, and Michael C Mozer. 2005. Automating vertical profiling. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/1094811.1094834

[19] Jeffrey K Hollingsworth, R Bruce Irvin, and Barton P Miller. 1991. The integration of application and system based metrics in a parallel program performance tool. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)* (1991). doi:10.1145/109625.109645

[20] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/2048066.2048081

[21] Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and Rachit Nigam. 2024. Unifying Static and Dynamic Intermediate Languages for Accelerator Generators. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. doi:10.1145/3689790

[22] Jiho Kim and Cong Hao. 2025. RealProbe: An Automated and Lightweight Performance Profiler for In-FPGA Execution of High-Level Synthesis Designs. In *Field-Programmable Custom Computing Machines (FCCM)*. doi:10.1109/FCCM62733.2025.00025

[23] 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)*. doi:10.1145/3192366.3192379

[24] Pouya Kousha, Bharath Ramesh, Kaushik Kandadi Suresh, Ching-Hsiang Chu, Arpan Jain, Nick Sarkauskas, Hari Subramoni, and Dhabaleswar K Panda. 2019. Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance GPU clusters. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. doi:10.1109/HiPC.2019.00022

[25] 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)*. doi:10.1145/3289602.3293910

[26] 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/

[27] mark1626. 2026. Dahlia Cookbook. https://github.com/Mark1626/dahlia-cookbook/

[28] Microchip. 2026. SmartHLS Compiler. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler#Documentation.

[29] 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)*. doi:10.1145/3385412.3385974

[30] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3445814.3446712

[31] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. doi:10.1109/MEMCOD.2004.1459818

[32] Perfetto. 2026. *Perfetto UI.* https://ui.perfetto.dev/

[33] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Y. Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*. doi:10.1145/2678373.2665678

[34] Frans Skarman, Lucas Klemmer, Daniel Große, Oscar Gustafsson, and Kevin Laeufer. 2025. Surfer—An Extensible Waveform Viewer. In *International Conference on Computer-Aided Verification (CAV)*. doi:10.1007/978-3-031-98685-7_19

[35] Nupur Sumeet, Deeksha Deeksha, and Manoj Nambiar. 2022. HLS_Profiler: Non-intrusive profiling tool for HLS based applications. In *International Conference on Performance Engineering*. doi:10.1145/3489525.3511684

[36] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: A framework for visualizing detailed GPU execution. In *Computer Graphics Forum*. doi:10.1111/cgf.14303

[37] The Petal Authors. 2025. *Petal Documentation.* https://docs.calyxir.org/running-calyx/profiler.html

[38] The Petal Authors. 2025. *Petal Source Code.* https://github.com/calyxir/calyx

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

[40] Xilinx Inc. 2021. *Vivado Design Suite User Guide: Synthesis. UG901 (v2017.2) June 7, 2017.* Retrieved November 19, 2021 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug901-vivado-synthesis.pdf

[41] Ayaka Yorihiro, Griffin Berlstein, Pedro Pontes García, Kevin Laeufer, and Adrian Sampson. 2026. Reproduction Package for "Understanding Accelerator Compilers via Performance Profiling". doi:10.5281/zenodo.21515679

Received 2026-03-17; accepted 2026-06-10