Unifying Static and Dynamic Intermediate Languages for Accelerator Generators

Caleb Kim*
Cornell University
USA

Pai Li*
Cornell University
USA

Anshuman Mohan
Cornell University
USA

Andrew Butt
Cornell University
USA

Adrian Sampson
Cornell University
USA

Rachit Nigam
Cornell University
USA

Abstract

Compilers for accelerator design languages (ADLs) translate high-level languages into application-specific hardware. ADL compilers rely on a hardware *control interface* to compose hardware units. There are two choices: *static* control, which relies on cycle-level timing; or *dynamic* control, which uses explicit signalling to avoid depending on timing details. Static control is efficient but brittle; dynamic control incurs hardware costs to support compositional reasoning.

Piezo is an ADL compiler that unifies static and dynamic control in a single intermediate language (IL). Its key insight is that the IL's static fragment is a *refinement* of its dynamic fragment: static code admits a subset of the run-time behaviors of the dynamic equivalent. Piezo can optimize code by combining facts from static and dynamic sub-modules, and it opportunistically converts code from dynamic to static control styles. We implement Piezo as an extension to an existing dynamic ADL compiler, Calyx. We use Piezo to implement an MLIR frontend, a systolic array generator, and a packet-scheduling hardware generator to demonstrate its optimizations and the static–dynamic interactions it enables.

1 Introduction

Accelerator design languages (ADLs) [8, 10, 14, 19, 30] raise the level of abstraction for hardware design. The idea is analogous to traditional software compilation: we want users to work not with gates, wires, and clock cycles, but with high-level or domain-specific concepts such as tensor operations [21], functional programs [10, 34], and recurrence equations [8]. Compilers then translate these high-level descriptions into efficient hardware designs. ADLs suffer cross-cutting compilation challenges, and the architecture community has responded with a range of compiler frameworks and intermediate languages [26, 37, 44, 46].

This paper identifies a central challenge for ADL compilers: the *control interface* for composing units of hardware. The choice of interface has wide-ranging implications on a compiler's expressive power, its ability to optimize programs, and the semantics of its intermediate language. There are two categories. *Dynamic* or *latency-insensitive* interfaces abstract away timing details and streamline compositional design, but they incur fundamental overheads [29]. *Static* or *latency-sensitive* interfaces are efficient, but they depend on the cycle-level timing of each module and therefore leak implementation details across module boundaries.

Intermediate languages (ILs) for ADLs use either dynamic interfaces [17, 32], static interfaces [26], or both [37, 46]. Static interfaces alone are insufficient because some computations, such as off-chip memory accesses, have fundamentally variable latencies. Infrastructures that support both interfaces typically *stratify* the IL into separate dynamic and static sub-languages [6, 46]. While stratified compilers can bring customized lowering and optimization strategies to bear on each sub-language, they entail duplicated implementation effort and miss out on cross-cutting optimizations that span the boundary between static and dynamic code. Stratification also infects the frontends targeting the IL: they must carefully separate code between the two worlds and manage their interaction.

We introduce Piezo, an IL and compiler for accelerator designs that freely mix static and dynamic interfaces. The key insight is that static IL constructs are all *refinements* of their dynamic counterparts: they admit a subset of the run-time behaviors. This unified approach lets transformations and optimizations work across both interface styles. Piezo also enables the incremental adoption of static interfaces: frontends can first establish correctness using compositional but slow dynamic code, and then opportunistically convert to efficient static interfaces. Refinement in Piezo guarantees that this transition is correct.

We implement Piezo as an extension to the dynamic-first Calyx infrastructure [32]. This paper shows how to compile Piezo's static extensions into pure Calyx. We lift Calyx's existing optimizations to support Piezo's static abstractions and implement new time-sensitive optimizations. Piezo can also automatically infer when some dynamic Calyx code has fixed latency, and promote it to static code.

We evaluate Piezo's new optimizations using a frontend that translates from high-level MLIR [23] dialects to Piezo. Time-sensitive optimizations improve execution times by

*Equally contributing authors.

(a) A dynamic IL wraps static components with dynamic wrappers.

(b) A *stratified* IL uses one compilation path for static "islands" and a separate one that incorporates dynamic components.

(c) Piezo represents static dynamic and static components in the same IL. The compiler can perform whole-program optimizations.

Figure 1. Hardware implementations of $(a+b)\times c \div d$. Green units have static interfaces; orange is dynamic.

2.5× on average over dynamic Calyx code. We also implement a packet-scheduling engine and study how Piezo optimizations, in concert with domain-specific human insight, are able to improve the performance of the generated hardware. As another domain-specific case study, we extend a systolic array generator to support fused dynamic operations to understand how Piezo can support interactions between fundamentally static and dynamic components.

2 Hardware Interfaces

Consider compiling the integer computation $(a + b) \times c \div d$ into hardware. Generating a hardware datapath entails orchestrating physical units—such as adders, multipliers, and dividers—over time. For this example, we use sequential, i.e., non-pipelined, hardware units. While many hardware units, such as adders and multipliers, have fixed latency, many do not: integer dividers, for instance, typically have data-dependent timing. Control logic for these two categories is fundamentally different. A variable-latency divider may expose 1-bit signal wires to start computation and to signal completion. A fixed-latency multiplier, however, needs no explicit completion signalling: clients can simply provide inputs and wait the requisite number of cycles. For this example, assume we have an adder with latency 1, a multiplier with latency 3, and a divider with variable latency.

Dynamic compilation. Figure 1a shows how dynamic-first ILs, such Calyx [32] and Dynamatic [17], might compile our expression. All units expose explicitly signalled dynamic interfaces; each static module requires a *wrapper* that counts clock cycles up to the unit's latency and then signals completion. A purely dynamic compiler benefits from a uniform interface and compositional reasoning, because no module can depend on the timing of any other. However, these wrappers incur time and space overheads, and optimizations cannot exploit timing information (§5.2).

Static compilation. Static-first ILs, such as HIR [26], require fixed-latency operations. They can support dynamic operators like dividers by using an upper-bound latency. Upper bounds are pessimistic, however, and some hardware operations have unbounded latency: the latency for an arbiter that manages conflicting memory accesses, for instance, fundamentally depends on the address stream.

Stratified static–dynamic compilation. Figure 1b illustrates a hybrid approach, such as DASS [6], that combines static and dynamic compilation. The idea is to compile the two parts of the program separately: first using static interfaces for the fixed-latency fragment, $(a+b)\times c$, and then using dynamic interfaces to combine this fragment with the variable-latency divider. This combination allows latency-sensitive optimizations on the static fragment while still allowing dynamic scheduling where it is beneficial.

This *stratified* approach, however, needs separate ILs for the two styles of computation. The compiler cannot exploit information across the static–dynamic boundary. Furthermore, it complicates the job for frontends that emit these ILs: switching a single subcomputation from static to dynamic requires a global change in the way the program is encoded.

Unified static–dynamic compilation in Piezo. Figure 1c represents our approach with Piezo: a unified IL that expresses both static and dynamic interfaces in one program. Piezo extends Calyx [32], an existing dynamic-first IL, with static constructs that *refine* the semantics of its dynamic constructs. By mirroring the dynamic IL abstractions with static counterparts, Piezo enables compositional reasoning, incremental adoption, and whole-program optimization across the static–dynamic boundary.

3 The Piezo Intermediate Language

This section introduces Piezo, a *unified* IL for compiling hardware accelerators. Piezo extends Calyx [32], an existing dynamic IL. Calyx has a growing family of frontends, such as for Halide [13, 35] and MLIR dialects in CIRCT [44, 47], that can adopt Piezo's static interfaces to improve performance.

We introduce Piezo using the program in Figure 2 as a running example. Our extensions to Calyx are in red. Deletions when porting from Calyx to Piezo are commented out with red slashes. We describe the existing Calyx IL (§3.1), show that its original *hint-based* treatment of static interfaces is insufficient (§3.2), and then introduce Piezo's extensions.

3.1 The Calyx IL

The Calyx IL intermixes software-like *control operators* with hardware-like *structural resources* [32]. The former simplifies encoding of high-level language abstractions, while the latter enables optimizations that exploit control information to optimize the physical hardware implementation.

Components. Components define units of hardware with input and output *ports*. In Figure 2, `expr` has four 32-bit input

```
component expr(a:32,b:32,c:32,d:32)->(out:32) {
  cells {
    add = std_add(32);   // 32-bit adder
    mult = std_mult(32); // 32-bit multiplier
    div = std_div(32);   // 32-bit divider
  }
  wires {
    static<1> group do_add {
      add.left = %[0:1] ? a;
      add.right = %[0:1] ? b;
      // do_add[done] = add.done;
    }
    static<3> group do_mult {
      mult.left = %[0:3] ? add.out;
      mult.right = c; // implicit %[0:3] guard
      // do_mult[done] = mult.done;
    }
    group do_div {
      div.go = 1'd1;
      div.left = mult.out;
      div.right = d;
      do_div[done] = div.done;
    }
    out = div.out;
  }
  control {
    seq { static seq { do_add; do_mult; }
          do_div;
    }
  }
}
```

Figure 2. A Piezo component that computes $(a + b) \times c \div d$. Our extensions to the language are shown in red.

ports (`a` through `d`) and one output port (`out`). A component has three sections: `cells`, `wires`, and `control`.

Cells. The `cells` section instantiates subcomponents. Cells can be either other Calyx components or *external definitions* defined in a standard HDL. The component `expr` instantiates three cells from the standard library: `add`, `mult`, and `div`. Each is parameterized by a bitwidth.

Wires. Calyx uses *guarded assignments* to connect two ports when a logical condition, called the *guard*, is true. Consider:

```
add.left = c0 ? 10;
add.left = c1 ? 20;
add.right = 30;
```

Here, `add.left` has the value 10 or 20 depending on which guard, `c0` or `c1`, is true. Meanwhile, `add.right` *unconditionally* has the value 30. Calyx's well-formedness constraint requires that all guards for a given port be mutually exclusive: it is illegal for `c0` and `c1` to simultaneously be true.

Groups. Assignments can be organized into unordered sets called *groups*. A group can execute over an arbitrary number of cycles and therefore requires a 1-bit `done` condition to signal completion. In Figure 2, the assignments in `do_div` compute `mult.out` ÷ `d` by passing in inputs and asserting the divider's "start" signal, `div.go`. The group's `done` signal is connected to the divider's `done` port, which becomes 1 when the divider finishes.

Control. The control section is an imperative program that decides when to execute groups. Calyx supports sequential (`seq`), parallel (`par`), conditional (`if`), and iterative (`while`) composition. The `if` and `while` constructs use one-bit condition ports. An `invoke` operator is analogous to a function call: it executes the control program of a subcomponent fully and then returns control to the caller.

3.2 Latency Sensitivity in Calyx

As a fundamentally dynamic language, Calyx *provides no guarantees on inter-group timing* in its control programs: programs cannot rely on the relative execution schedule of any two groups. For example, any amount of time may pass between steps in a `seq` block; and different threads in a `par` block may start at different times, so no thread may rely on the timing of another [2]. The compiler exploits this semantic flexibility to optimize programs by adjusting this timing.

However, latency insensitivity is expensive [29]. To help mitigate this cost, Calyx comes with an optional attribute, `@static(`n`)`, that *hints* to the compiler that a group or component has a fixed latency of n cycles. These hints do not affect the program's semantics, so the compiler can disregard or erase them. However, this optional nature makes them challenging to support and reason about. Each compiler optimization pass must treat the hint *pessimistically*; there is no contract to maintain time-sensitive behavior, which has lead to several bugs [41, 43]. Erasable hints are also unsuitable for integrating with external hardware, such as a module that produces an answer exactly 4 cycles after reading an input.

This paper's thesis is that the distinction between static and dynamic control is too important—and too semantically meaningful—to be encoded as an optional hint. Instead, the IL's static constructs must be a *semantic refinement* (§3.5) of its dynamic equivalents: converting from dynamic to static restricts a program's timing behavior; the reverse is not allowed because it allows more possible behaviors.

3.3 Static Structural Abstractions

Piezo extends Calyx with new, time-sensitive structural abstractions: static components and static groups.

Static components. Piezo's static components are like Calyx's dynamic components, but they use a different "calling convention." Where dynamic components, such as `std_div`, use a `go` signal to start computation and a `done` signal to indicate completion, static components only use `go`. Compare the interface of a multiplier to that of a divider:

```
static<3> primitive std_mult[W](
  go: 1, left: W, right: W) -> (out: W);
primitive std_div[W](
```

```
  go: 1, left: W, right: W) -> (out: W, done: 1)
```

The `static<n>` qualifier indicates a latency of n cycles that is guaranteed to be preserved by the Piezo compiler.

Static groups and relative timing guards. Static groups in Piezo use *relative timing guards*, which allow assignments on specific clock cycles. This group computes $ans = 6 \times 7$:

```
1 static<4> group mult_and_store {
2   mult.left = %[0:3] ? 6;
3   mult.right = %[0:3] ? 7;
4   mult.go = %[0:3] ? 1;     // run the multiplier
5   ans.in = %3 ? mult.out;   // ans is a register
6   ans.write_en = %3 ? 1;  // assert write enable
7 }
```

Like `do_div` in Figure 2, the group sends operands into the `left` and `right` ports of an arithmetic unit. Here, however, relative timing guards encode a cycle-accurate schedule: a guard `%[i:j]` is true in the half-open interval from cycle i to cycle j of the group's execution. The assignments to ports `mult.left` and `mult.right` are active for the first 3 cycles. The guard `%3` is syntactic sugar for `%[3:4]`, so the write into the `ans` register occurs on cycle 3. The `static<4>` annotation tells us the group is done on cycle 4.

Piezo's relative timing guards resemble cycle-level schedules in some purely static languages [26, 31]. However, they count relative to the start of the *group*, not that of the *component*. This distinction is crucial since it lets Piezo use static groups in both static and dynamic contexts.

3.4 Static Control Operators

Piezo provides a static alternative to each dynamic control operator in Calyx. Unlike the dynamic versions, static operators guarantee specific cycle-level timing behavior.

The `static` qualifier marks static control operators. While dynamic commands may contain both static and dynamic children, static commands must only have static children. We write $|c|$ for the latency of a static command c.

Sequential composition. A `static seq` like this:

```
static seq {c1; c2; ...; cn;}
```

has a latency of $\sum_1^n |c_i|$ cycles. c_1 executes in the interval $[0, |c_1|)$ after the `seq`'s start, c_2 in $[|c_1|, |c_1| + |c_2|)$, and so on.

Parallel composition. A `static par` statement:

```
static par {c1; c2; ...; cn;}
```

has latency $\max_1^n |c_i|$. Command c_1 is active between $[0, |c_1|)$, program c_2 between $[0, |c_2|)$, and so on.

The parallel threads in a `static par` can depend on the "lockstep" execution of all other threads. Threads can therefore communicate, whereas conflicting parallel state accesses in Calyx are data races and therefore undefined behavior [2].

Conditional. Static conditionals use a 1-bit port p:

```
static if p { c1 } else { c2 }
```

Table 1. Interfaces between types of control.

Abbr.	Caller	Callee	Calling Convention
$D \rightarrow D$	Dynamic	Dynamic	Calyx [32]
$S \rightarrow S$	Static	Static	See §4.1
$D \rightarrow S$	Dynamic	Static	See §4.3
$S \rightarrow D$	Static	Dynamic	Not supported

The latency is the upper bound of the branches, $\max(|c_1|, |c_2|)$.

Iteration. There is no static equivalent to Calyx's unbounded `while` loops. Piezo instead adds both static and dynamic variants of fixed-bound `repeat` loops:

```
static repeat n { c }
```

The body executes n times, so the latency is $n \times |c|$.

Invocation. Piezo's `static invoke` corresponds to Calyx's function-call-like operation and requires the target component to be static. The latency is that of the invoked cell.

Group enable. A leaf statement can refer to a `static group` (e.g., `do_add` in Figure 2). The latency is that of the group.

3.5 Unification Through Semantic Refinement

Piezo's static constructs are all semantic *refinements* [9] of their dynamic counterparts in Calyx. The semantics of dynamic code admit many concrete execution schedules, such as arbitrary delays between group executions. Each static construct instead selects one *specific* cycle-level schedule from among those possibilities.

Refinement enables *incremental adoption*: a frontend can first generate purely dynamic code, establish correctness using the original Calyx semantics based on partial ordering between group executions, and then add `static` qualifiers. We can establish correctness for the `static` code by the same argument as the original code, since it admits a subset of the original's cycle-level executions. This implication also means that Piezo may automatically infer `static` qualifiers for some code (§5.2).

Semantic refinement also enhances optimization (§5.3). Piezo can enrich existing Calyx passes with timing information to expose more optimization opportunities in static code. New optimizations can also combine information across static and dynamic code. This kind of optimization would be challenging in a stratified compiler like DASS [6] with separate ILs and lowering paths for static and dynamic code.

4 Compilation

Figure 3 shows the compilation flow for Piezo. After optimizations (§5), we translate Piezo constructs to pure Calyx.

The Piezo compiler relies on control interfaces for static code, dynamic code, and invocations that cross the static–dynamic boundary. For example, in a control statement like

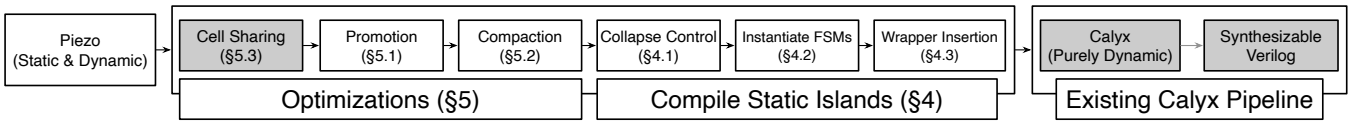


Figure 3. Piezo compilation flow. The extended Piezo syntax is optimized (§5) and compiled (§4) to pure Calyx abstractions.

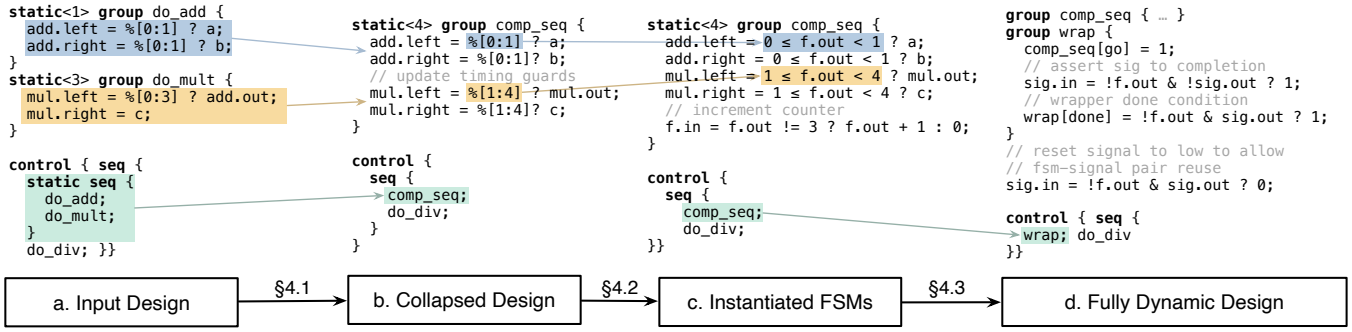


Figure 4. The stages for implementing Piezo's static control operators.

`seq { a; b; }`, both the parent (the `seq`) and the children (`a` and `b`) could use either static or dynamic control.

Table 1 lists the four possible cases, denoted $I_p \longrightarrow I_c$ where the parent and child interfaces I are static (S) or dynamic (D). The all-dynamic case, $D \longrightarrow D$, is the Calyx baseline. The all-static case, $S \longrightarrow S$, works by counting cycles (§4.1). For $D \longrightarrow S$, the compiler adds a *dynamic wrapper* around the static child (§4.3). Piezo disallows the $S \longrightarrow D$ case with a compile-time error: if the child takes an unknown amount of time, it is impossible to give the parent a static latency bound. Given the prohibition against $S \longrightarrow D$ composition, we can think of any Piezo program as a dynamic control program with interspersed *static islands* [5, 7].

Compilation starts by *collapsing* static islands into static groups (§4.1) and then generating FSM logic to implement relative timing guards (§4.2). Finally, it *wraps* static islands for use in their dynamic context (§4.3).

4.1 Collapsing Control

Figures 4a–b illustrate how Piezo *collapses* static control statements into static groups. The new group contains all the assignments from the old groups used in the statement (`do_add` and `do_mult` in the example), with their timing guards updated to implement the statement's timing.

We collapse each static island in a *bottom-up* order: to compile any statement, we first collapse all its children. Before collapsing, we preprocess assignments to add timing guards where they are missing: for example, the assignment `mul.right = c` in Figure 4a is normalized to `mul.right = %[0:3] ? c`. We combine these timing guards with any existing guards using the conjunctive operator `&`.

Parallel composition. With all timing guards explicit and the children already collapsed, compiling `static par` is simple: we merge the assignments from the children into a single static group. The new group's latency is the maximum latency among the children. For example, we compile:

```
static<1> group A { r1.in = 1; r1.write_en = 1; }
static<2> group B { r2.in = 4; r2.write_en = 1; }
control { static par { A; B; } }
```

into:

```
static<2> group comp_par {
  r1.in = %[0:1] ? 1; r1.write_en = %[0:1] ? 1;
  r2.in = %[0:2] ? 4; r2.write_en = %[0:2] ? 1;
}
control { comp_par; }
```

Sequential composition. To compile `static seq`, we can merge assignments from child groups. We rewrite each timing guard `%[a:b]` into `%[cur+a:cur+b]` where *cur* is the cumulative latency of all previous siblings. The new group's latency is the sum of latencies of the children. For example:

```
control { static seq { A; B; } }
```

compiles (where `A` and `B` are as above) into:

```
static<3> comp_seq {
  r1.in = %[0:1] ? 1; r1.write_en = %[0:1] ? 1;
  r2.in = %[1:3] ? 4; r2.write_en = %[1:3] ? 1;
}
control { comp_seq; }
```

Conditional. Semantically, `static if` only checks its condition port once: it must ignore any changes to the port while either branch executes. To honor this while compiling `static if cond { A } else { B }`, we stash `cond`'s value in a special register on the first cycle, and leave the register's value unchanged thereafter. We generate logic to

select between `A` and `B` using `cond` directly during the first cycle, and the special register for the remaining cycles.

Iteration. To implement `static repeat` n `{ }` g `}`, the collapsed body group g must run n times. Activating a static group in Piezo entails asserting its `go` signal for the group's entire latency. We can therefore compile the loop into a group that asserts g's `go` signal for $n \times |g|$ cycles:

```
static<n × |g|> repeat_group { g[go] = 1; }
```

In this case, the body group g remains alongside the new `repeat_group`. The body group's FSM (see §4.2) is responsible for resetting itself every $|g|$ cycles.

4.2 FSM Instantiation

Figures 4b–c illustrate the next compilation step: eliminating static timing guards (§3.3). For a static group with latency n, this pass generates a finite state machine (FSM) counter that counts from 0 to $n-1$; it automatically resets back to 0 immediately after hitting $n-1$. We translate each timing guard `%[j:k]` into the guard $j \leq f < k$ where f is the counter.

Resetting the counter from $n-1$ to 0 lets static groups re-execute immediately after finishing. Compiled `repeat` and `while` loops, for example, can chain invocations of static bodies without wasting a cycle between each iteration.

While FSM instantiation would work the same on the original program, it is more efficient to run it after collapsing control. Generating fewer static groups yields fewer FSM registers and incrementers.

4.3 Wrapper Insertion

Figures 4c–d illustrate the final compilation step: converting each collapsed, timing-guard-free static group (4c) into a dynamic group (4d).

We generate a *dynamic wrapper* group for every static group that has a dynamic parent. Like any dynamic group, the wrapper exposes two 1-bit signals, `go` and `done`. When activated with `go`, the wrapper in turn actives the `go` signal of the static group. To generate the `done` signal, the wrapper uses a 1-bit signal `sig` to detect if a static island's FSM has run once. When the FSM is 0 *and* `sig` is high, we know that the FSM has *reset* back to 0: the wrapper asserts `done`.

Special case:* `while` *with static body. The wrapper strategy works in the general case, but when the dynamic parent is a `while` loop, the compiled code "wastes" one cycle per iteration to check the loop condition. This strategy incurs a relative overhead of $1/b$ when the body takes b cycles, which is bad for short bodies and large trip counts. This special case is common because it lets programs build long-running computations from compact hardware operations, so we handle it differently to eliminate the overhead.

To compile `while` c `{` g `}` where g is static, we generate a wrapper for the entire `while` loop instead of a wrapper for g alone. Each time the FSM returns to the initial state, the wrapper concurrently checks the condition port and asserts `done` if the condition is false. This is another application of refinement in Piezo: Calyx's `while` operator admits multiple possible cycle-level timing behaviors, and we generate a specific one to meet our objectives.

5 Optimizations

We design a pass to opportunistically convert dynamic code to static code and new time-sensitive static optimizations.

5.1 Static Inference and Promotion

Calyx code written as dynamic often does not need to be dynamic: its latency is deterministic. *Promoting* such code to use static interfaces can save time and resources for dynamic signalling—but it is not always profitable. We therefore split the process into two steps: *inference*, which detects when dynamic groups and control have a static latency, and *promotion*, which converts dynamic code to static code when it appears profitable. Inference records information without affecting semantics, while promotion refines the program's semantics. We infer freely but promote cautiously.

Inferring static latencies. We use an existing Calyx pass called `infer-static-timing` pass to infer latencies for both groups and control programs. It infers a group's latency by analyzing its uses of its `go` and `done`. Suppose we have:

```
group g {
  reg.in = 10;  // reg is a register (latency 1)
  reg.write_en = 1;
  g[done] = reg.done;  }
```

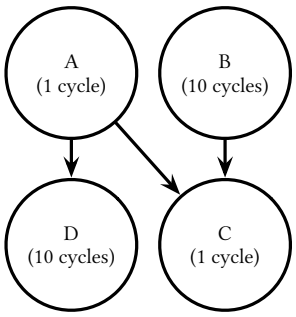
The pass observes that (1) `reg.write_en` is asserted unconditionally, (2) the group's `done` flag is tied to `reg.done`, and (3) the register component definition declares a latency of 1. Calyx therefore attaches a `@static(1)` annotation to `g`: the group will take exactly one cycle to run.

For control operators, e.g., `seq`, inference works bottom-up. If all of a `seq`'s children have `@static` annotations, the `seq` gets a `@static(`n`)` annotation where n is the sum of the latencies of its children. Despite this inference, Calyx's original time-sensitive FSM generation pass cannot compile static control islands; instead, the entire component needs to be static [42]. Piezo lifts this restriction.

Promoting code from dynamic to static. We can promote groups and control based on inferred `@static` annotations. For example, after inferring the `@static(1)` annotation for the group `g`, we can promote it to:

```
static<1> group g { reg.in=10; reg.write_en=1; }
```

While static control has lower control overhead and enables downstream optimizations, it incurs two major costs. We introduce *promotion heuristics* to balance each of these costs.



(a) Dependency graph.

```
wires {
  // Dummy delay groups
  static<1> group
      del_1 {}
  static<10> group
      del_10 {}
}
static par {
  A; B;
  static seq {
      del_10; C; };
  static seq {
      del_1; D; };
}
```

(b) Compacted schedule.

Figure 5. Schedule compaction uses data dependencies to generate an *as-soon-as-possible* schedule.

First, each static island requires one wrapper interface and one counter register. This cost is constant for each island, while the benefit of simpler static control scales with the code size of the island. Therefore, the compiler introduces a *threshold* parameter that only promotes static islands above a certain code size, in terms of the number of groups and conditional ports.

The second cost affects long-running static islands, which can require large FSM registers and associated comparators. Some islands reach hundreds of millions of cycles (see §6.1), so two smaller islands can sometimes be cheaper than one large island. The compiler accepts a parameter that gives an *upper bound* on the number of cycles that a potential static island can run for, and does not promote any island that would run longer than this upper bound.

We empirically calibrate these parameters' default settings using experience with real programs; see §6.1.

5.2 Schedule Compaction

Piezo features a new *schedule compaction* optimization to maximize parallelism while respecting data dependencies. Schedule compaction is only feasible in a unified compiler. In a dynamic IL, the compiler lacks latency information altogether. In a static IL, the compiler has latency information but is barred from rescheduling code, which could violate timing properties that the program relies on. Traditional C-based high-level synthesis (HLS) compilers accomplish similar scheduling optimizations, but by translating between two vastly different representations: from untimed C to a fully static HDL. A unified IL, in contrast, can perform this optimization within a single abstraction by exploiting the interaction between static and dynamic code.

Compaction occurs during the transition from dynamic to static code, after `@static` inference and as a supplement to standard promotion. Consider the following `seq`:

```
@static(22) seq { A; B; C; D; }
```

where Figure 5a shows the groups' latencies and data dependencies. If we only perform promotion, it will take 1 + 10 + 1 + 10 = 22 cycles.

Piezo's schedule compaction pass reschedules the group executions to start as soon as their dependencies have finished. Specifically, `A` and `B` start at cycle 0 because they have no dependencies; `C` and `D` start on cycle 10 and 1 respectively: the first cycle after their dependencies have finished. This compacted schedule takes only 11 cycles.

The optimization extracts a data dependency graph for the children of the `seq` and topologically sorts it to produce an as-soon-as-possible schedule. Next, it reconstructs a control program to implement this schedule. It emits a `static par` with one group per thread. To delay a group's start, it uses an empty delay group, as shown in Figure 5b. Since all `del_n` groups are removed during the collapsing step of compilation (§4.1), they incur no overhead.

5.3 Cell Sharing

Calyx has a register sharing pass [32] to reduce resource usage. It uses Calyx's control flow to compute registers' live ranges and remaps them to the same instance when the ranges do not overlap. Piezo's variant is a generalized *cell sharing* pass that works with arbitrary components instead of just registers.

Piezo first extends this pass to work with mixed static–dynamic designs and then enhances it to opportunistically exploit static timing information. In addition to working uniformly on both static and dynamic code, Piezo's cell sharing optimization can share cells across the static–dynamic boundary: static and dynamic parts of the design can use the same cell. This is not possible in stratified ILs [6, 46] that use separate optimization pipelines for the two interface styles.

Piezo's cell sharing pass also improves over sharing in Calyx when it can exploit cycle-level timing in static code. The original Calyx optimization must over-approximate live ranges because of Calyx's loose timing semantics. For example, `par` provides no guarantees about the cycle-level timing of its threads (§3.4), so the compiler must conservatively assume that *all* live ranges in one thread may overlap with the live ranges in a different thread. This prevents Calyx from sharing cells between sibling `par` threads. Piezo's enhanced cell sharing optimization exploits timing guarantees (§3.4) to compute precise, cycle-level live ranges. These live ranges are soundly comparable across `par` threads and enables sharing between them. This enhancement is an example of a *latency-sensitive* optimization from Figure 3.

6 Effects of Piezo Optimizations

We compare Piezo's performance to Calyx when compiling linear algebra kernels and a packet scheduling engine.

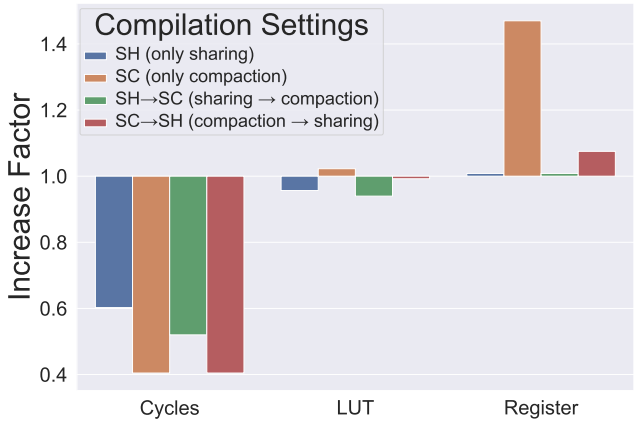


Figure 6. Geometric means across the 19 benchmarks for cycle counts, LUT usage, and register usage. The bars on the graph are normalized to the **B**aseline, i.e., non-promoted, implementation. Because the geometric means for worst slack (not shown here) among the five configurations are all within 5% of each other, cycle counts are a fairly accurate measure of runtime.

6.1 Linear Algebra Kernels

CIRCT [44] is an MLIR [23] subproject for designing open-source hardware flows. Calyx is a core dialect within CIRCT and can be generated from C++ or PyTorch programs. We lower the Polybench benchmarks [24], written in C++, to Calyx using the CIRCT flow, automatically promote them to use Piezo's new abstractions (§5.1), and report the cycle counts and resource usage on an FPGA.

Of the 30 Polybench benchmarks, the CIRCT flow fails to compile 11 due to various limitations in the frontend dialects. The 19 compilable benchmarks are chiefly dense loop nests, so large parts of them can be scheduled statically. However, there is some dynamic behavior, including dynamically-timed integer division and "triangular" nested loops (i.e., the inner loop bound depends on the outer loop's index).

Configurations. To generate Piezo designs from Calyx, we first perform static promotion (§5.1). Then, we compile each design with different configurations of the schedule compaction (**SC**, §5.2) and cell sharing (**SH**, §5.3) passes.

1. **B**aseline: The standard Calyx compiler, without Piezo abstractions or promotion.
2. **SH**: Static promotion, then cell sharing.
3. **SC**: Static promotion, then schedule compaction.
4. **SH→SC**: Static promotion, sharing, then compaction.
5. **SC→SH**: Static promotion, compaction, then sharing.

Experimental setup. We use Verilator v5.006 [45] to obtain cycle counts. Our synthesis flow uses Vivado 2020.2 and targets the Xilinx Alveo U250 board with a period of 7 ns. We report post place-and-route resource estimates for lookup tables (LUTs, FPGAs' primary logic resource) and registers.

The geometric mean of the worst timing slack across the 19 benchmarks varies within 5% between the five configurations. We therefore believe that 7 ns is an appropriate clock period for these designs, so measuring cycle counts suffices to reflect actual running time.

We also ran experiments to explore promotion parameters (§5.1). While they unsurprisingly yield nonuniform trade-offs between area and latency, we select default parameters that provide a good balance across benchmarks: 1 for the static island size and 4096 for the cycle-count limit.

6.1.1 Schedule Compaction and Cell Sharing.

Figure 6 compares the configurations normalized to baseline (**B**). We report the geometric means for cycle counts, LUTs, and register usage across the 19 benchmarks.

Cycle counts. Static promotion has significant impact on cycle count. Running **SH**, without compaction, can isolate the impact of promotion: a 1.67× geomean speedup over **B**. Schedule compaction (**SC**) improves the cycle count further, yielding a 2.5× geomean speedup over the baseline designs.

LUT usage. **SH** saves LUTs (0.96×) while **SC** increases them (1.02×), although both are quite similar to **B**. **SH** benefits from Piezo's static control interface, while **SC** incurs logic overhead to implement its parallelized schedules.

Register usage. Sharing hardware resources (**SH**) performs essentially the same (1.01×) as to **B**. This is because **B** already has a sharing optimization, and there were not many opportunities to apply the Piezo extensions to exploit time sensitive sharing (§5.3). **SC** incurs a register cost (1.47×) to implement its schedules.

6.1.2 Phase Ordering.

Schedule compaction and cell sharing are partially in conflict: the former adds parallelism, while the latter exploits *non-parallel* code to share resources. They embody a fundamental trade-off between performance and area. We measure their interaction in either order:

Cycle counts. **SC→SH** performs identically to **SC** alone (2.5× speedup). The opposite ordering, **SH→SC**, is slightly slower (1.92×), but still faster than **SH** (1.67×). Sharing impedes some, but not all, opportunities for compaction.

LUT usage. **SH→SC** saves LUTs slightly (0.94× of **B**) while **SC→SH** performs similarly to **B** (0.99×). However, the effects across benchmarks are nonuniform, and the combinations of optimizations can sometimes outperform **SH** alone.

Register usage. Running sharing first (**SH→SC**) achieves similar register reduction to **SH** alone (1.01× of **B**). The reverse ordering (**SC→SH**) is slightly worse (1.07×) but still significantly better than **SC** alone (1.47×). Running **SC** first only opportunistically adds parallelism; the designs still have some fundamental sequential behavior that allows sharing.

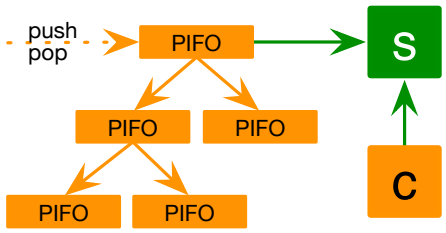


Figure 7. A PIFO tree, a statistics component (s), and a controller (c). Green is static; orange is dynamic.

6.2 Packet Schedulers

We use a second, more domain-specific case study to understand Piezo optimizations in more detail. In software-defined networking (SDN) [11], *programmable packet scheduling* offers flexible policies for allocating bandwidth and ordering packet delivery. *PIFO trees* [28, 39] are a flexible mechanism for line-rate packet scheduling. The packet buffer of a switch consists of a compositional hierarchy of priority queues (PIFOs), each of which implements a policy for scheduling the data held by its children.

We implement a new Piezo-based generator for PIFO tree packet schedulers as shown in Figure 7. We *push* incoming packets into the PIFO tree by inserting into a leaf node and adding priority metadata to each parent. To *pop* the highest-priority packet for forwarding, we query the tree to identify it and update the metadata. The tree also maintains telemetric data—counts of classes of packets—by reporting to a separate statistics component (s) at each push. An SDN controller (c) might exploit these statistics to implement adaptive scheduling policies.

Our implementation generates the PIFO tree itself, which is fundamentally dynamic because of the data-dependent behavior of queues, and a simple static statistics unit. While it is not the focus of this case study, we also include a simple dynamic controller to consume the statistics.

Implementation. We implement a flexible Piezo PIFO tree generator in 600 lines of Python. The generator can produce binary PIFO trees of varying heights, arrangements, and capacities. It can also implement different scheduling policies by deciding how packets get assigned to leaves and metadata to internal nodes. For our experiment, we generate a tree with 5 PIFOs and overall capacity 10. We set up the scheduling parameters to implement a hierarchical round-robin scheduling policy.

We use the generator to synthesize four hardware configurations: plain Calyx, Calyx promoted to Piezo, explicitly annotated Piezo, and annotated, promoted Piezo. The second configuration is the result of automatically promoting the first (see §5.1). The third includes manually inserted `static<>` annotations that encode domain-specific insight into the generated hardware's timing. The fourth configuration is the result of automatically promoting the third. The generated design is 1,100 lines of Piezo IL.

Results. We generate a workload of 10,000 packets with randomly interspersed but balanced push and pop events. We measure the LUT count, register count, and cycles per push (C/push) for each design. The best values are in bold.

Configuration	LUTs	Registers	C/push
Calyx	1547	393	143.25
Promoted to Piezo	1610	391	139.25
Annotated Piezo	1556	385	140.25
Annotated, Promoted Piezo	**1544**	**381**	**137.25**

The resource usage of the three designs is similar: the PIFO tree (which is always dynamic) is the dominant component in all three designs, and the statistics component (which is dynamic in Calyx and static in Piezo) is small.

The promoted Piezo implementation improves on the original Calyx implementation's C/push measure because the promotion pass exploits small opportunities for static promotion in all components, including components that are understood to be dynamic. This comes at the cost of resource usage: promotion to Piezo also triggers schedule compaction (§5.2), and compaction costs LUTs. The manually annotated Piezo implementation also improves on the baseline's C/push measure—domain knowledge lets the human guide the compiler—but *without* suffering a LUT cost. The annotated, promoted Piezo implementation performs the best of all. Its C/push measure is better than the manually annotated Piezo implementation because, as before, the promotion pass exploits opportunities for promotion that are not clear to a human. Critically, its LUT count does *not* suffer compared to annotated Piezo: compaction only runs on promoted static islands, not user-defined static components.

7 Systolic Arrays

Systolic arrays [20] are a class of architecture commonly used in machine learning [12, 18] built from interconnected processing elements (PEs). PEs perform simple computations and communicate with other PEs in a simple, regular manner. We redesign an existing systolic array generator that targets Calyx to use Piezo abstractions and demonstrate how it enables efficient composition and incremental adoption.

7.1 Systolic Arrays in Piezo

Calyx has an existing systolic array generator that produces hardware to multiple fixed-size matrices. The interface of the generated systolic array accepts rows and columns of input matrices A and B in parallel using an output-stationary dataflow. Each PE performs a multiply-accumulate operation and forwards its operands.

This case study addresses three main limitations:

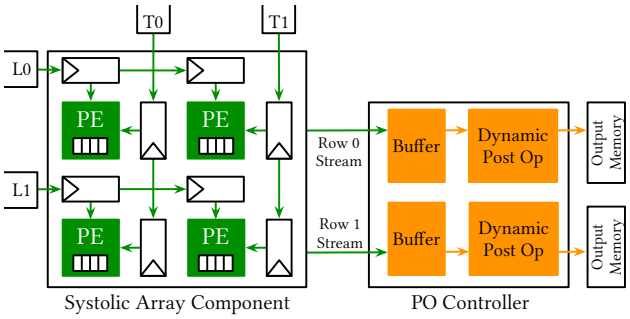


Figure 8. Our 2 × 2 systolic array with a *dynamic* post op. The buffers in the post op controller are not necessary for static post ops. Green is static and orange is dynamic. Input memories (`L0`, `L1`, `T0`, `T1`) may have non-fixed length.

- Calyx's dynamic interfaces between PEs make it challenging to pipeline computations, hindering performance. Piezo enables efficient pipeline execution using static interfaces.
- A purely static implementation can only efficiently support fixed-sized matrices. Piezo's unified approach makes it possible to support flexible matrix sizes while maintaining efficient pipelined execution.
- Systolic arrays often have fused *post operations* that apply elementwise functions to the product matrix. We show how Piezo's mixed interfaces support various post post ops and optimize the composed design across the static–dynamic boundary.

Pipelining processing elements. Calyx's systolic array generator decouples the logic for the PE from the systolic array itself to modularize code generation. This means that the systolic array must communicate with its PEs through dynamic interfaces. Because there are no timing guarantees, the generator does not pipeline the PEs and instead uses sequential multipliers. Extending the generator to a dynamically pipelined design would add unnecessary overhead; we would need queues to buffer values between PEs.

Instead, Piezo abstractions let the systolic array communicate with its PEs using efficient static interfaces that facilitate pipelining. Besides removing the overhead from dynamic interfaces, this also simplifies the logic of the systolic array fabric, which is in charge of data movement. Because we have a pipeline with initiation interval of 1, the fabric can unconditionally move data every cycle and guarantee the right value will be read.

Fixed contraction dimension. While static interfaces allow for efficient, pipelined execution, they can limit computational flexibility. For example, an output-stationary matrix-multiply systolic array should be able to multiply matrices of sizes $i \times k$ and $k \times j$ for any value of k. However, this requires dynamic control flow: the computation needs to repeat k times where k is a runtime value. Piezo abstractions support this with ease: we use a `while` loop to execute the systolic array's logic k times. Furthermore, because the control program in the loop body is purely static, Piezo's special handling ensures that the body executes every cycle (§4.3).

Supporting fused post operations. A common optimization in machine learning frameworks [15] *fuses* matrix multiplication with elementwise *post operations*, such as nonlinearities, to avoid writing the intermediate matrix back to memory. These post operations can be either fundamentally static or dynamic. Our goal is to decouple the implementation of post operations from the systolic array: to keep the code generation modular without sacrificing efficient interfaces. We implement two post operators (POs): (1) a static ReLU operation, `x > 0 ? x : 0`, and (2) a dynamic leaky ReLU [25] operation, `x > 0 ? x : 0.01*x`. The latter is dynamic because the true branch can directly forward the output while the false branch requires a multiplication.

Figure 8 overviews the architecture. We instantiate the systolic array and PO components for the number of rows in the resulting matrix. If the PO is dynamic, the *PO controller* instantiates buffers to queue the output stream but elides them for static POs. The interface between the systolic array and PO is pipelined: a row's PO starts its computation as soon as an output is available. Most of the code—the systolic array, the controller, the PEs—is reused regardless of the PO's interface; Piezo's unified abstractions enable this reuse.

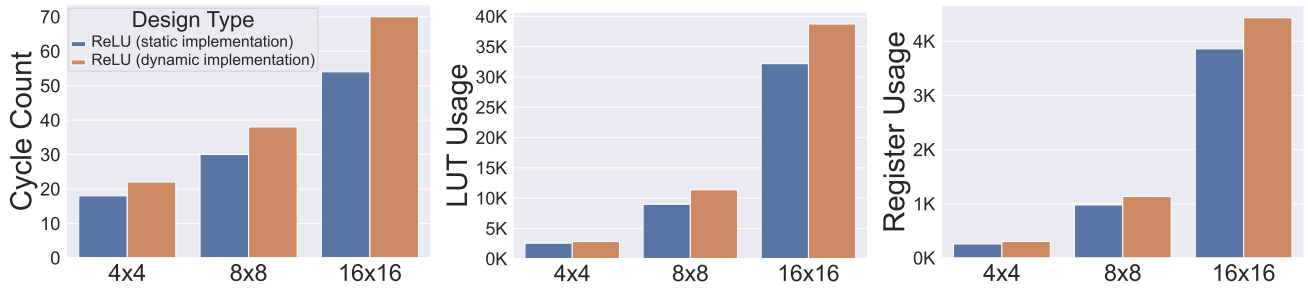
7.2 Evaluation

Our evaluation seeks to answer the following questions:

- Does the pipelined Piezo-generated systolic arrays outperform the existing Calyx-generated designs?
- Can Piezo implement a runtime-configurable contraction dimension for systolic arrays with low overhead?
- Do cross-boundary optimizations let Piezo eliminate overheads when the systolic array is coupled with a static post operation?

Effect of pipelining. For the 16 × 16 design, the pipelined implementation in Piezo achieves a max frequency of 270 MHz and performs the computation in 52 cycles in comparison to the original design's 250 MHz and 248 cycles. The latency improvement is from the pipelined execution and the frequency improvement from simplified control logic.

Configurable matrix dimensions. We compare systolic arrays with *flexible* and *fixed* matrix size support. The flexible design takes 1 extra cycle to finish, uses 8% more LUTs (for logic to check the loop iteration bound), and uses the same number of registers. The flexible design pays some



(a) Cycle counts. **(b)** LUT usage. **(c)** Register usage.

Figure 9. Performance and FPGA resource utilization of two implementations of a fused matrix-multiply–ReLU kernel on Piezo-compiled systolic arrays. We compare static and dynamic interfaces for the ReLU unit.

overhead to gain dynamic functionality, while the fixed design is fully static, thereby eliminating dynamic overhead: Piezo expresses both with minimal code changes.

Overhead of dynamic post operations. We perform a synthetic experiment to quantify overhead of a dynamic interface between the systolic array and the PO: we use the simple ReLU post operation in its default, static form and compare it against a version that artificially wraps it in a dynamic interface. Since the computation is the same, the only difference is the interface. Figures 9a–9c report the cycle counts, LUTs, and register usage of the resulting designs. In addition to a higher cycle count, the dynamic implementation also has higher LUT and register usage, stemming from the extra control logic and buffers respectively.

We also implemented a truly dynamic post operator, leaky ReLU. We omit its measurements here because it conflates the costs of the operation and static–dynamic interaction.

8 Related Work

Piezo builds on a rich body of prior work on compilers for accelerator design languages (ADLs): high-level programming models for designing computational hardware. However, these compilers tend to prioritize either static or dynamic interfaces in the hardware they generate—or, when they combine both strategies, to disallow fluid transitions between the two styles.

Traditional C-based high-level synthesis (HLS) compilers [3, 4, 16, 27, 33, 48] intermix static and dynamic-latency operations, such as dividers. They do so using software ILs like LLVM [22], which ties them to C-like, sequential computational models. Critically, traditional HLS tools are monolithic: they do not expose consistent intermediate representations that support modular pass development, decoupled frontends and backends, and layered correctness arguments. Piezo contributes a stable IL that includes both software- and hardware-like abstractions and thus supports modular passes that address both static and dynamic control.

The most closely related compilers seek to combine aspects of static and dynamic control [6, 46]. DASS [6] is the first HLS compiler we are aware of to specifically balance static and dynamic scheduling within the same program. In DASS, either the user [6] or some heuristic [7] identifies parts of the high-level design that would benefit from static scheduling. Compilation proceeds in two phases: DASS first compiles all the static islands, and then it uses a second, dynamic, approach to schedule the rest of the program while treating the pre-compiled islands as opaque operators. In contrast, Piezo's unified IL can treat static portions of the program transparently and optimize them in the same framework as dynamic code. Szafarczyk et al. [40] provide the opposite approach to DASS: it finds sections of programs that are amenable to dynamic scheduling in a previously statically-scheduled program. The dynamic sections are decoupled from the static parts and compiled into processing elements that communicate over latency-insensitive channels. Hector [46] is a dialect of MLIR [23] that supports three scheduling styles: pipeline, static, and dynamic. Each style corresponds to a different Hector component type, uses a different a syntax and semantics, and uses a different lowering strategy. In contrast to these, Piezo provides a unified IL in which either the frontend, or a compiler heuristic (§5.1), can easily covert dynamic programs to static and vice-versa, and lowers them using a single compilation pipeline. This lets Piezo reuse optimizations between the two modes and even optimize across the boundary between dynamic and static code.

Other ILs for ADL compilers also give passes control over scheduling, but focus on either static [1, 36, 38] or dynamic interfaces [17, 37]. In particular, HIR [26] is an MLIR-based IL that describes schedules using *time variables* that describe the clock cycles on when each value in a design is available. Filament [31], like HIR, explicitly dictates the cycle-level schedule of hardware operations, but it encodes these time intervals into a type system. Piezo's relative timing guards (§3.3) work similarly and describe the cycle-level schedule

for assignments. However, Piezo's timing guards are relative to the start of each group's execution. This *relative* timing limits the scope of static schedules and enables flexible composition with dynamic groups, scalable reasoning, and efficient lowering (§4.1). Finally, unlike both systems, Piezo supports both static and dynamic interfaces.

9 Conclusion

Latency-sensitive hardware refines the semantics of latency-insensitive hardware. Every practical accelerator compiler must combine the two styles, and this correspondence is the foundation for combining them soundly.

References

[1] C Scott Ananian. Silicon C: A hardware backend for SUIF, May 1998. https://flex.cscott.net/SiliconC/.

[2] Griffin Berlstein, Rachit Nigam, Christophe Gyurgyik, and Adrian Sampson. Stepwise debugging for hardware accelerators. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[3] Cadence. Stratus high-level synthesis. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

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

[5] Jianyi Cheng, Estibaliz Fraca, John Wickerson, and George A. Constantinides. Balancing static islands in dynamically scheduled circuits using continuous petri nets. *IEEE Transactions on Computers*, 2023.

[6] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[7] Jianyi Cheng, John Wickerson, and George A. Constantinides. Finding and finessing static islands in dynamically scheduled circuits. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022.

[8] Jason Cong and Jie Wang. PolySA: Polyhedral-based systolic array auto-compilation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[9] Robert Dockins. *Operational refinement for compiler correctness*. PhD thesis, Princeton University, 2012.

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

[11] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. Using deep programmability to put network owners in control. *SIGCOMM Comput. Commun. Rev.*, 2020.

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

[13] Sergi Granell Escalfet. Accelerating halide on an FPGA. Master's thesis, Universitat Politècnica de Catalunya, 2023.

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

[15] Intel. oneAPI deep neural network library developer guide and reference. https://oneapi-src.github.io/oneDNN/.

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

[17] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

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

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

[20] Hsiang-Tsung Kung. Why systolic architectures? *IEEE Computer*, 1982.

[21] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[22] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *International Symposium on Code Generation and Optimization (CGO)*, 2021.

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

[25] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*, 2013.

[26] Kingshuk Majumder and Uday Bondhugula. HIR: An MLIR-based intermediate representation for hardware accelerator description. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[27] Mentor Graphics. Catapult high-level synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/, 2021.

[28] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. Formal abstractions for packet scheduling. *Proc. ACM Program. Lang.*, 7(OOPSLA2), 2023.

[29] Kevin E. Murray and Vaughn Betz. Quantifying the cost and benefit of latency insensitive communication on FPGAs. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014.

[30] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[31] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. Modular hardware design with timeline types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2023.

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

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

[34] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Trans. Archit. Code Optim.*, 2017.

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

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

[37] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir: An intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[38] Rohit Sinha and Hiren Patel. synASM: A high-level synthesis framework with support for parallel and timed constructs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[39] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. Compiler discovered dynamic scheduling of irregular code in high-level synthesis. In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

[41] The Calyx Authors. Compress static FSMs. https://github.com/cucapra/calyx/issues/936, 2023.

[42] The Calyx Authors. Fix top-down static timing. https://github.com/cucapra/calyx/pull/1338, 2023.

[43] The Calyx Authors. Problems with static FSMs. https://github.com/cucapra/calyx/issues/940, 2023.

[44] Mike Urbach and Morten B. Petersen. HLS from PyTorch to System Verilog with MLIR and CIRCT. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2022.

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

[46] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. Hector: A multi-level intermediate representation for hardware synthesis methodologies. In *International Conference On Computer Aided Design (ICCAD)*, 2022.

[47] Zhenya Zang, Uwe Dolinsky, Pietro Ghiglio, Stefano Cherubin, Mehdi Goli, and Shufan Yang. Building a reusable and extensible automatic compiler infrastructure for reconfigurable devices. In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2023.

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