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

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 submodules, 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 a frontend for an existing ADL, a systolic array generator, and a packet-scheduling hardware generator to demonstrate its optimizations and the static–dynamic interactions it enables.

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

Additional Key Words and Phrases: Intermediate Language, Accelerator Design Language

ACM Reference Format:
Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and Rachit Nigam. 2024. Unifying Static and Dynamic Intermediate Languages for Accelerator Generators. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 350 (October 2024), 26 pages. https://doi.org/10.1145/3689790

1 Introduction

Accelerator design languages (ADLs) [Cong and Wang 2018; Durst et al. 2020; Hegarty et al. 2014; Koeplinger et al. 2018; Nigam et al. 2020] raise the level of abstraction for hardware design. The idea is analogous to traditional software compilation: instead of making users work with gates, wires, and clock cycles, we provide them with high-level or domain-specific abstractions such as tensor operations [Lai et al. 2019], functional programs [Durst et al. 2020; Pu et al. 2017], and

[*]Equally contributing authors.

Authors' Contact Information: Caleb Kim, Cornell University, Ithaca, USA, cmk265@cornell.edu; Pai Li, Cornell University, Ithaca, USA, pl582@cornell.edu; Anshuman Mohan, Cornell University, Ithaca, USA, amohan@cs.cornell.edu; Andrew Butt, Cornell University, Ithaca, USA, atb78@cornell.edu; Adrian Sampson, Cornell University, Ithaca, USA, asampson@cs.cornell.edu; Rachit Nigam, Cornell University, Ithaca, USA, rnigam@cs.cornell.edu.

ACM 2475-1421/2024/10-ART350
https://doi.org/10.1145/3689790

recurrence equations [Cong and Wang 2018]. 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 [Majumder and Bondhugula 2024; Sharifian et al. 2019; Urbach and Petersen 2022; Xu et al. 2022].

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 [Murray and Betz 2014]. *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 [Josipović et al. 2018; Nigam et al. 2021], static interfaces [Majumder and Bondhugula 2024], or both [Sharifian et al. 2019; Xu et al. 2022]. 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 [Cheng et al. 2020; Sharifian et al. 2019; Xu et al. 2022]. 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.

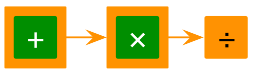
We introduce Piezo, an IL and compiler for accelerator designs that supports boths static and dynamic interfaces in a single, unified language. The key insight is that static IL constructs are *refinements* of their dynamic counterparts: they admit a subset of the run-time behaviors. This unified approach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

We implement Piezo as an extension to the dynamic-first Calyx infrastructure [Nigam et al. 2021]. 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.

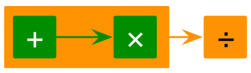
We first evaluate Piezo's new optimizations using a frontend that translates from an HLS-like ADL [Nigam et al. 2020] to Piezo, and show that time-sensitive optimizations result in faster execution times (0.82× as many cycles as Calyx) and smaller designs (0.52× as many LUTs as Calyx). We also implement 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. We lift several limitations of the original Calyx generator, and additionally find that, because we have unlocked pipelining, we require only 0.18× as many cycles.

2 Hardware Interfaces

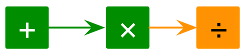
Consider compiling the integer computation $(a + b) \times c \div d$ into hardware. Generating a hardware datapath entails orchestrating physical units—such as adders, multipliers, and dividers—over time. For this example, we use sequential, i.e., non-pipelined, hardware units. While many hardware



(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.

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

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 wires to start the 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 that we are working with 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 [Nigam et al. 2021] and Dynamatic [Josipović et al. 2018], 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 [Majumder and Bondhugula 2024], 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 the stratified approach, which is used by DASS [Cheng et al. 2020], for combining 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 [Nigam et al. 2021], 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.

```
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;
      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;
    }
  }
}
```

Fig. 2. A Piezo component that computes $(a + b) \times c \div d$. Our extensions to Calyx are shown in green, with deletions shown in red.

3 The Piezo Intermediate Language

This section introduces Piezo, a *unified* IL for compiling hardware accelerators. Piezo extends Calyx [Nigam et al. 2021], an existing dynamic IL. Calyx has a growing family of frontends, such as for Halide [Granell Escalfet 2023; Ragan-Kelley et al. 2013] and MLIR dialects in CIRCT [Urbach and Petersen 2022; Zang et al. 2023], 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 green. In red, we show lines of code that are required in Calyx but must be deleted when porting to Piezo. 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 [Nigam et al. 2021] intermixes software-like *control operators* with hardware-like *structural resources*. 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 ports (`a` through `d`) and one output port (`out`). A component has three sections: **`cells`**, **`wires`** (which can be organized into `groups`), and **`control`**.

Cells. The **`cells`** section instantiates subcomponents. Cells can be either other Calyx components or *external definitions* defined in a standard HDL. In Figure 2, 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. Further, different threads in a **`par`** block may start at different times, so no thread may rely on the timing of another [Berlstein et al. 2023]. Not only is dynamic timing necessary to support portions of the design for which there is no known worst-case latency (e.g., an off-chip memory access), but its semantic flexibility allows the compiler to optimize programs by adjusting inter-group timing.

Despite its advantages, however, latency insensitivity is expensive [Murray and Betz 2014]. To 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.[1] These hints do not affect the program's semantics, so the compiler may disregard or erase them—and it often does in practice [The Calyx Authors 2022, 2023b]. This optional nature makes Calyx's semantics-free `@static` hint challenging to use correctly, both for frontends that generate Calyx IL code and within the compiler's optimization and lowering passes.

Frontends cannot use the hint to generate code whose correctness depends on details of its timing. For instance, an efficient systolic array implementation (§7) needs a carefully constructed static pipeline with precise cycle-level timing coordination, and Calyx's `@static` hint cannot implement this kind of schedule. Furthermore, erasable hints are unsuitable for frontends that want

[1]From the Calyx paper: "... latency-sensitive compilation is *just* an optimization—it can be disabled, debugged, and interacted with separately from the compilation pipeline."[Nigam et al. 2021, §4.4; emphasis reproduced from original.]

to integrate with external hardware, such as a module that produces an answer exactly 4 cycles after reading an input. The only workaround is to add a latency-insensitive wrapper, which adds overhead and defeats the purpose of supporting latency sensitivity.

The hint-based approach hampers the compiler's internals because each pass must treat the hint *pessimistically*. Using Calyx's `@static` hint, it is impossible to correctly implement any optimization that transforms a program's schedule in a way that exploits cycle-level scheduling. For example, *schedule compaction* works by coordinating timing across parallel computations (see §5.2); any schedule that employs this optimization depends on guarantees about the timing of each group in the schedule. If downstream passes can ignore the `@static` hint, the upstream optimization has no straightforward way to enforce a particular cycle-level schedule. This confusion has yielded several real bugs in Calyx [The Calyx Authors 2023a,c]. Without timing guarantees in the IL, the only alternative is a *monolithic* design that discovers, optimizes, and lowers statically-timed code in a single compiler pass without ever exposing intermediate code. A monolithic approach is in conflict with the goals of a modular compiler that supports isolated development, reasoning, and testing.

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 $ans = 6 \times 7$:

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

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 on the first line tells us the group is done on cycle 4.

Piezo's relative timing guards resemble cycle-level schedules in some purely static languages [Majumder and Bondhugula 2024; Nigam et al. 2023]. However, they count relative to the start of the *group* rather than the entire *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 $[0, |c_1|)$, command 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 [Berlstein et al. 2023].

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

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

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* [Dockins 2012] 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.

```
int A[N];

// Statically scheduled.
int staticFn(int x) {
  return ((x+2)*x+3)*x+6;
}

// Dynamically scheduled.
int filterSumMult() {
  int res = 0;
  for (int i = 0; i < N; i
      ++) {
    int val = A[i];
    if (val >= 0) {
      int temp = staticFn(
          val);
      res += temp;
    }
    res *= val;
  }
  return res;
}
```

(a) An example program in a high-level language with one statically scheduled function and one dynamically scheduled function.

```
wires {
  ...
  // elided: group `mult_res_val`, registers
       `val` and `res`
  // `geq` is a ">=" comparator
  geq.left = val.out;
  geq.right = 0;
  out = res.out; // return res
}
control {
  repeat N { // for (int i=0; i<N; i++)
    ...
    seq {
      if geq.out {
        static par {...} // staticFn(val)
        // elided: res += staticFn(val)
      }
      mult_res_val; // res *= val;
    }
    ...
  }
}
```

(b) Design after lowering to Piezo.

Fig. 3. We have elided some details in the lowered Piezo code to lighten the presentation. In particular the **static par** block would contain the same computation as `staticFn`. The definition of the group `mult_res_val` has also been elided; it contains multiplication from `filterSumMult`. Because Piezo can implement the entire computation within a single component, it is able to share(§5.3) the multiplier used by `mult_res_val` with the multipliers used in the **static par** block.

Refinement enables *incremental adoption:* a frontend can first generate purely dynamic code, establish correctness using the simpler Calyx semantics, and then opportunistically 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 refinement 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 hybrid optimization would be challenging in a stratified compiler like DASS [Cheng et al. 2020] with separate ILs and lowering paths for static and dynamic code.

Example. To get a better understanding of the types of optimizations that are unlocked by a unified IL, we will walk through an adaptation of the very example used by Cheng et al. [2020] to motivate DASS. This is shown in Figure 3a. The program is written in a high-level language and iterates through an array. On each iteration, if the array entry is greater than zero, it calls `staticFn` on the entry and accumulates it to the result. It then unconditionally multiplies the array entry with the result and stores the answer as the new running result.

In DASS or any other stratified compiler, `staticFn` is compiled completely statically and appears as a black box to the dynamically scheduled `filterSumMult`. The problem with this stratification is

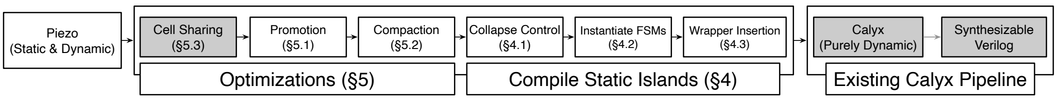
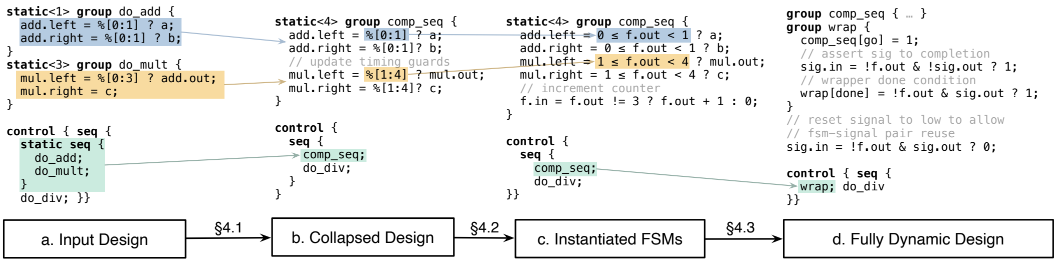


Fig. 4. Piezo compilation flow. The extended Piezo syntax is optimized (§5) and compiled (§4) to pure Calyx abstractions.



a. Input Design

```
static<1> group do_add {
  add.left = %[0:1] ? a;
  add.right = %[0:1] ? b;
}
static<3> group do_mult {
  mul.left = %[0:3] ? add.out;
  mul.right = c;
}

control { seq {
  static seq {
    do_add;
    do_mult;
  }
  do_div; }}
```

b. Collapsed Design

```
static<4> group comp_seq {
  add.left = %[0:1] ? a;
  add.right = %[0:1]? b;
  // update timing guards
  mul.left = %[1:4] ? mul.out;
  mul.right = %[1:4]? c;
}

control {
  seq {
    comp_seq;
    do_div;
  }
}
```

c. Instantiated FSMs

```
static<4> group comp_seq {
  add.left = 0 ≤ f.out < 1 ? a;
  add.right = 0 ≤ f.out < 1 ? b;
  mul.left = 1 ≤ f.out < 4 ? mul.out;
  mul.right = 1 ≤ f.out < 4 ? c;
  // increment counter
  f.in = f.out != 3 ? f.out + 1 : 0;
}

control {
  seq {
    comp_seq;
    do_div;
}}
```

d. Fully Dynamic Design

```
group comp_seq { … }
group wrap {
  comp_seq[go] = 1;
  // assert sig to completion
  sig.in = !f.out & !sig.out ? 1;
  // wrapper done condition
  wrap[done] = !f.out & sig.out ? 1;
}
// reset signal to low to allow
// fsm-signal pair reuse
sig.in = !f.out & sig.out ? 0;

control { seq {
  wrap; do_div
}}
```

Fig. 5. The stages for implementing Piezo's static control operators.

Table 1. Interfaces between types of control.

Abbr.	**Caller**	**Callee**	**Calling Convention**
$D \longrightarrow D$	Dynamic	Dynamic	Calyx [Nigam et al. 2021]
$S \longrightarrow S$	Static	Static	See §4.1
$D \longrightarrow S$	Dynamic	Static	See §4.3
$S \longrightarrow D$	Static	Dynamic	Not supported

that optimizations cannot span the static and dynamic fragments of the code. For example, because `staticFn` must be compiled in isolation, we cannot save area by sharing its multiplier(s) with the one in the `res *= val` line from `filterSumMult`.

Figure 3b sketches a simplified form of the corresponding Piezo code. In Piezo, the entire function can be lowered in a single, unified component containing both static and dynamic control. This means that Piezo's resource sharing pass (§5.3) can accomplish the sharing that DASS (and other stratified compilers) cannot: by optimizing the component as a whole, it can share the multiplier used by the dynamic group `mult_res_val` with the multipliers used in the **`static par`** block. Furthermore, in DASS, `staticFn` and `filterSumMult` are compiled using completely different strategies, and target two completely different IRs. In Piezo, we can lower both computations using the same IR, making it easier to maintain and extend the compiler.

4 Compilation

Figure 4 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 **`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 $S \rightarrow D$ composition, we can think of any Piezo program as a dynamic control program with interspersed *static islands* [Cheng et al. 2023, 2022].

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

Collapsing is the process of converting a static control statement into a single group. Collapsing preserves latency: it converts a static control statement with latency n into a static group with latency n. Figures 5a–b provides an example of how Piezo converts a **`static seq`**. 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 recursively in a *bottom-up* order: to compile any statement, we first collapse all its children.

Preprocessing. Before collapsing, we preprocess assignments to add timing guards where they are missing: for example, the assignment `mul.right = c` in Figure 5a is normalized to `mul.right = %[0:3] ? c`. In general, each input assignment has this form:

```
dst = guard ? src;
```

where missing guards are assumed to be `1`. The preprocessing step rewrites this assignment into:

```
dst = guard & %[0:|g|] ? src;
```

where $|g|$ is the group's latency. (A separate pass simplifies redundant guards: for example, the guard `[0:10] & [0:2]` can be simplified to `[0:2]`.)

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; }
```

In general, given some **`static par`**:

```
static par {c₁; c₂; ...; cₙ;}
```

We first recursively collapse each child c_i into a group g_i, resulting in:

```
static par {g₁; g₂; ...; gₙ;}
```

Let A_i denote the set of assignments contained in group g_i. We define a new static group `comp_par` that contains assignments $\bigcup_{i=1}^{n} A_i$ and has latency $\max_1^n |c_i|$. Finally, we return `comp_par` as the result of the collapsing procedure.

Sequential composition. To compile **`static seq`**, we merge assignments from child groups while "shifting" their timing guards. After recursively compiling the statement's children, the **`static seq`** has this form:

```
static seq {g1; g2; ...; gn;}
```

Let A_i again be the set of assignments in group g_i. We rewrite each timing guard `%[a:b]` in each group g_i to `%[`$d_i + a : d_i + b$`]` where $d_i = \sum_{j=1}^{i-1} |c_j|$, i.e., the relative start time for the group.

We then combine the time-shifted assignments. If A'_i denotes the modified assignments, we construct a new static group `comp_seq` containing assignments $\bigcup_{i=1}^{n} A'_i$ and with latency $\sum_1^n |c_i|$. `comp_seq` is the result of collapsing this statement.

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. We honor this while compiling:

```
static if cond {c_t} else {c_f}
```

by stashing `cond`'s value in a special register on the first cycle, and leaving the register's value unchanged thereafter. We generate logic to select between c_t and c_f using `cond` directly during the first cycle, and the special register for the remaining cycles.

Specifically, let A_c denote these assignments:

```
cond_reg.in = %0 ? cond;
cond_reg.write_en = %0 ? 1;
cond_wire = %0 ? cond : cond_reg.out;
```

We generate `cond_reg` and `cond_wire` as a one-bit register and wire, respectively. The intuition is that `cond` is used directly on the 0th cycle and stored in `cond_reg` on the remaining cycles.

Next, we recursively collapse each child c_t and c_f into groups g_t and g_f, resulting in:

```
static if cond {g_t} else {g_f}
```

Let A_t and A_f denote the set of assignments contained in group g_t and g_f, respectively. We then modify the assignments in A_t. For each assignment `dst = guard ? src` in A_t, we rewrite each guard to be `guard & cond_wire.out`. Let A'_t denote this modified set of assignments. We modify A_f similarly, using the negation `!cond_wire.out` in the guard, to produce A'_f.

Finally, we define a new static group `comp_if` that contains assignments $A'_t \cup A'_f \cup A_c$ and has latency $\max(|c_1|, |c_2|)$ as the result of the collapsing procedure.

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. This is the one control structure in which not all the assignments collapsed into a single group.

4.2 FSM Instantiation

Figures 5b–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 5c–d illustrate the final compilation step: converting each collapsed, timing-guard-free static group (5c) into a dynamic group (5d).

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 along with 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 the program's 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` must be asserted unconditionally, (2) the group's **done** flag must be 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.

In general, for any cell `c`, three conditions must hold in order for Calyx to infer its latency:: (1) `c`'s **go** port or equivalent (e.g., `write_en` for registers)[2] is asserted unconditionally (2) the group's **done** flag is tied to `c.done`, and (3) Calyx must be able to determine a constant latency `n` for `c`: this information either comes from Calyx primitives with a constant latency, which are hard-coded to provide their latency information, or user-defined components for which Calyx has inferred their latency. If these conditions hold, then Calyx will infer the latency of the group to be `n`.

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 [The Calyx Authors 2023b]. 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; }
```

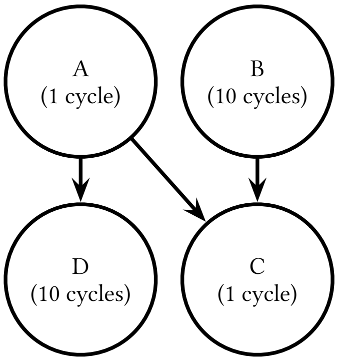
While static control has lower control overhead and enables downstream optimizations, it may incur some costs as well: we introduce *promotion heuristics* to balance these costs. Each static island requires one wrapper interface and one counter register. This cost is constant for each island, while the 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.

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

[2]To determine what the "equivalent" of the **go** port is for a given component, there is a `@go` attribute that can be attached to ports in the signatures of Calyx components. For example, in the Calyx primitive library, there is a `@go` attribute attached to the `write_en` port in the signature of `std_reg`.



(a) Dependency graph.

```
wires {
  // Dummy delay groups
  static<1> group
      delay_1 {}
  static<10> group
      delay_10 {}
}
static par {
  A; B;
  static seq {
      delay_10; C; };
  static seq {
      delay_1; D; };
}
```

(b) Compacted schedule.

Fig. 6. Schedule compaction uses data dependencies to generate an *as-soon-as-possible* schedule.

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 6a 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.

General procedure. In general, compaction works by first calculating a compacted schedule and then constructing a control program to implement this schedule.

It first calculates all potential dependencies between children in a **seq**. These dependencies include read-after-write, write-after-write, or write-after-read dependencies. Dependency analysis is conservative: for example, any group which *may* read or write from a cell counts as a read or write respectively during dependency calculations. Piezo then builds a dependency graph and topologically sorts it to produce a list of children L. It then iterates through L to produce an as-soon-as-possible (ASAP) schedule: for each child $c_i \in L$, it assign a start time $s(c_i)$ according to the following equation: $s(c_i) = \max_{d_i \in D_i}(s(d_i) + |d_i|)$ where D_i represents the set of children that c_i depends on. In other words, it assigns the earliest possible start time for c_i that still honors its dependencies.

Next, it reconstructs a control program to implement this schedule. It emits a **static par** with one thread per child. For each child c_i, it creates an empty static group e with latency $s(c_i)$ and then creates

```
static seq {e; c_i;}
```

Each resulting **static seq** is a thread in the **static par**. An example is shown in Figure 6b. Since all `delay_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 [Nigam et al. 2021] 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.

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 [Cheng et al. 2020; Xu et al. 2022] 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 4.

6 Effects of Piezo Optimizations

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

6.1 Linear Algebra Kernels

Dahlia [Nigam et al. 2020] is an HLS-like imperative programming language that enables predictable hardware design. Dahlia already features two backends: one for Calyx and one for Vitis, a commercial C++ HLS compiler [AMD Inc. 2021]. First, we lower the Dahlia implementations of Polybench benchmarks [Louis-Noel Pouchet 2021] to Calyx. Then we promote the Calyx code to Piezo, thereby automatically benefitting from some of Piezo's new abstractions (§5.1). We report the cycle counts and resource usage on an FPGA, comparing the performance of Piezo against both of Dahlia's existing backends.

The 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) **SH**: Static promotion, then cell sharing.
(2) **SC**: Static promotion, then schedule compaction.
(3) **SH→SC**: Static promotion, sharing, then compaction.
(4) **SC→SH**: Static promotion, compaction, then sharing.

Vitis HLS baseline. We compare both Calyx and Piezo to a commercial HLS compiler to put the differences into context. As a monolithic, proprietary toolchain, AMD's Vitis HLS is fundamentally different from Dahlia, Calyx, and Piezo: it exposes no stable intermediate language and offers no interchangeable suite of modular compiler passes. Further, it is a commercial product with many heuristic-based optimizations that are more mature than Piezo's, so it will probably outperform it. For example, unlike the Dahlia-to-Calyx compiler, it includes *automatic pipelining*, where the compiler searches for a series of latency-balanced stages to implement a given data-flow graph.

Experimental setup. We use Verilator v5.006 [Veripool 2021] to obtain cycle counts. Our synthesis flow uses Vivado 2020.2 and targets the Zynq UltraScale+ XCZU3EG board with a clock period of 7 ns. This is the exact same setup the Calyx authors used for Dahlia's existing backends [Nigam et al. 2021], apart from using slightly different versions of Verilator and Vivado: they use Verilator v4.108 and Vivado 2017.2. We report post place-and-route resource estimates for lookup tables (LUTs, the primary logic resource of FPGAs) and registers.

We also run experiments to explore the threshold parameter (§5.1): while they unsurprisingly yield nonuniform trade-offs between area and latency, we select a default parameter of 2 for the static island size threshold.

Benchmark characterization. The matrix sizes involved in each benchmark are around 8 for each dimension (e.g., a 2-dimensional matrix has 64 elements). For most benchmarks, Piezo's cycle counts are on the order of thousands of cycles, and LUT usage is on the order of hundreds.

We chose the linear algebra kernels from Polybench because of the substantial body of prior work on FPGA acceleration for this category of computation [Chen et al. 2024; Skalicky et al. 2013] that has demonstrated power and performance improvements over CPUs [De Matteis et al. 2020]. While an exhaustive CPU/FPGA comparison is out of scope for this paper, which focuses on comparing FPGA compilation strategies, we include a simple comparison against a CPU to put the main results into context.

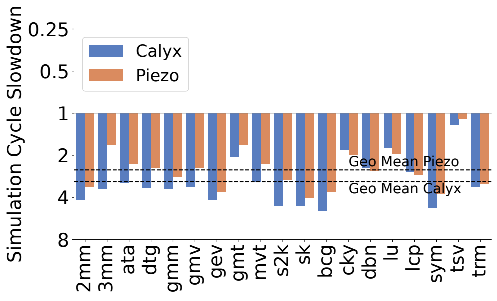
For a CPU baseline, we compile the original Dahlia source code to C++ and then to native code with `-O3` on gcc 11.4.0. We run the benchmarks on a server with dual Intel Xeon Gold 6230 processors [Intel 2024a] with 20 cores (40 total threads) at 2.10 GHz. We used the C++ standard library's `std::chrono::high_resolution_clock` to measure execution time, averaged across 20 runs. The thermal design power (TDP) for this CPU is 125 W.

We compute the FPGA running time as $(p - w_s) \times c$ where p is the clock period, w_s is the worst slack, and c is the cycle count (i.e., we conservatively estimate the minimum clock period and multiply it by the number of cycles). While exact energy characterization on FPGAs requires specialized proprietary tools, we survey related work that uses the same Xilinx ZU3EG part [Dong et al. 2021; Jiang et al. 2022]. The highest power consumption measured in that work is 5.5 W. That measurement is for neural network inference for image classification, a larger design than the Polybench implementations (it uses 61,362 LUTs and 360 DSPs, while the largest utilizations among the Polybench designs are 1,276 LUTs and 9 DSPs). We therefore consider 5.5 W to be a conservative upper bound for the benchmarks' power consumption.

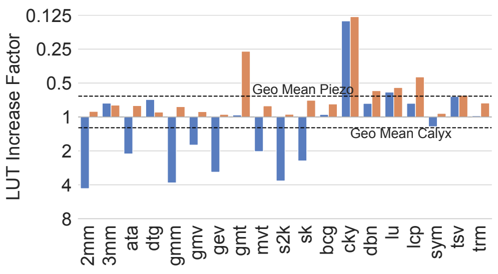
On average, the CPU is 4.4× faster, in exchange for using roughly 23× the power (125 vs. 5.5 W). A primary factor in the difference stems from platforms' clock frequencies: the CPU runs at 2.10 GHz, while the FPGA designs were on average 291.3 MHz. As a consequence of their lower clock frequency, the FPGA-accelerated Polybench kernels can offer improved performance-per-watt efficiency while the CPU in this comparison still wins on raw execution time.

6.1.1 Comparison to Calyx and Vivado HLS. We use Piezo's **SC→SH** configuration to compare against Calyx and Vitis; Figure 7 shows results relative to Vitis (for each graph, lower is worse). Piezo outperforms Calyx on both latency and LUTs: comparing geometric means, Piezo takes 0.82× as many cycles as Calyx and takes 0.52× as many LUTs. Schedule compaction can explain the faster designs, while using simpler static interfaces reduces LUT usage.

Compared to Vitis HLS, Piezo generates smaller but slower designs: the benchmarks take 0.65× the LUTs but 2.54× as many clock cycles. The primary reason for the difference is that Vitis

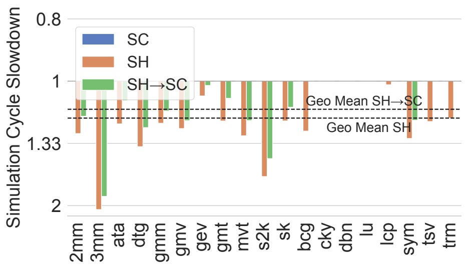


(a) Cycle counts.

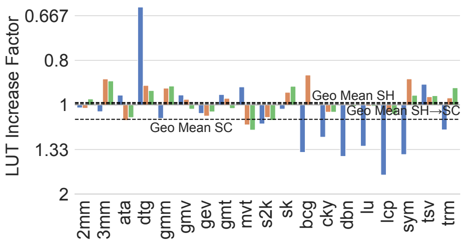


(b) LUT Usage.

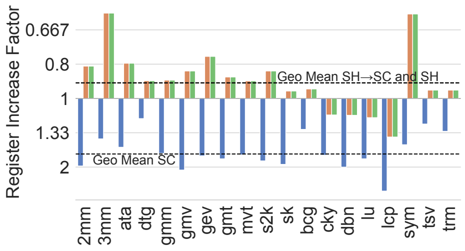
Fig. 7. Cycle count and LUT usage for the 19 linear algebra Polybench benchmarks, relative to Vitis HLS (lower is worse). For cycle counts: Piezo takes a geometric mean of 0.82× compared to Calyx and 2.54× compared to Vitis. For LUTs: Piezo takes 0.52× and 0.65× compared to Calyx and Vitis, respectively.



(a) Cycle counts.



(b) LUT Usage.



(c) Register Usage.

Fig. 8. Performance of Piezo designs compiled with various optimization orderings (lower is worse). Results are relative to **SC→SH**. The cycle counts are identical across the configurations **SC** and **SC→SH**, which is why no blue bars appear in (a).

HLS performs automatic pipelining search while the Dahlia compiler does not. (Piezo can express pipelined designs, but the frontend must decide the stage breakdown.) We anticipate that a frontend that aggressively pipelines could further close the gap with commercial HLS tools.

6.1.2 Effects of Optimizations and Phase Ordering. We first explain the effect of Piezo's various optimizations by comparing **SH** and **SC**. Then, we examine **SH→SC** and **SC→SH** to see the impact of the ordering of these optimizations. Figure 8 shows the results for the various configurations, relative to the **SC→SH** configuration, which is the configuration used in §6.1.1.

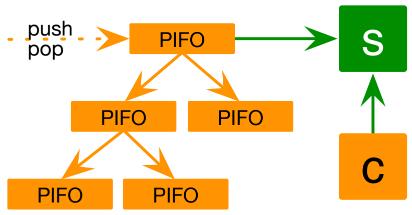


Fig. 9. A PIFO tree, a statistics component (s), and a controller (c). Green is static; orange is dynamic.

Cycle counts. Schedule compaction (**SC**) provides a consistent performance improvement: it yields designs that take a geometric mean of 0.85× the cycles compared to non-compacted designs (**SH**).

LUT and register usage. Designs that share hardware resources (**SH**) use 0.91× the LUTs and 0.53× the registers, compared to **SC** designs.

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. The opposite ordering, **SH→SC**, is slightly slower, taking 1.12× the number of cycles, but still faster than **SH** (1.17×). Sharing impedes some, but not all, opportunities for compaction.

LUT usage. **SH→SC**, **SC→SH**, and **SH** all perform similarly, while **SC** slightly increases LUT usage (1.04×). However, the effects across benchmarks are nonuniform, and various combinations of optimizations can sometimes outperform other combinations depending on the benchmark.

Register usage. Running sharing first (**SH→SC**) achieves identical register reduction to **SH** alone: they both use 0.9× the registers compared to **SC→SH**. However, **SC→SH** is still significantly better than **SC** alone, which increases register usage by a factor of 1.68×. Running **SC** first only opportunistically adds parallelism; the designs still have some fundamental sequential behavior that allows sharing.

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) [Foster et al. 2020], *programmable packet scheduling* offers flexible policies for allocating bandwidth and ordering packet delivery. *PIFO trees* [Mohan et al. 2023; Sivaraman et al. 2016] 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 9. We *push* incoming packets into the PIFO tree by inserting it into a leaf node and adding priority metadata to each parent node. 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 explicitly annotated Piezo that is further promoted. 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 for each design. The best values are in bold.

Table 2. Performance for a 10,000 packet workload using different compilation strategies.

Configuration	LUTs	Registers	Cycles per Push
Calyx	957	310	133.05
Promoted to Piezo	959	302	123.55
Annotated Piezo	968	302	130.05
Annotated, Promoted Piezo	**920**	**292**	**120.55**

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 cycles per push measure because the compiler exploits small opportunities for static promotion and compaction in all components, including components that are understood to be dynamic. The manually annotated Piezo implementation also improves on the baseline's cycles per push measure—domain knowledge lets the human guide the compiler. However, the annotated, promoted Piezo implementation performs the best of all, in both area and latency. Human insight, along with compiler optimizations, allow us to uncover more promotion and compaction opportunities than either one alone, decreasing cycles per push, LUTs and registers.

7 Systolic Arrays

Systolic arrays [Kung 1982] are a class of architecture commonly used in machine learning [Fowers et al. 2018; Jouppi et al. 2017] 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.

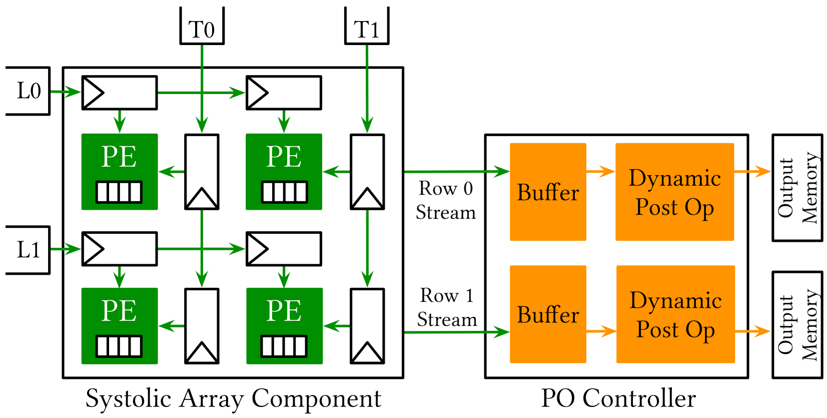


Fig. 10. Our 2×2 systolic array with a *dynamic* post op. The buffers in the post op controller are not necessary for static post ops. Green is static and orange is dynamic. Input memories (`L0`, `L1`, `T0`, `T1`) may have non-fixed length.

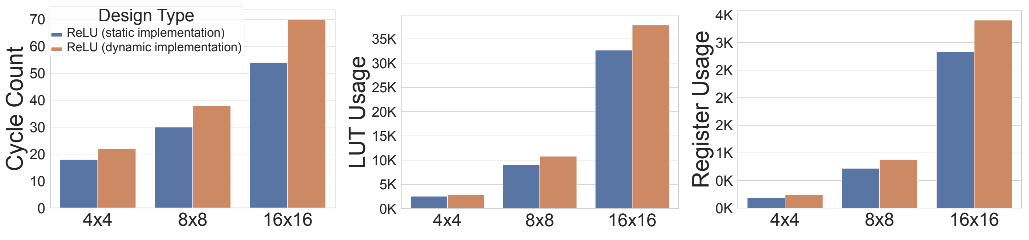
This case study addresses three main limitations:

- 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 operations 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).



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

Fig. 11. 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.

Supporting fused post operations. A common optimization in machine learning frameworks [Intel 2024b] *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 [Maas et al. 2013] 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 10 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

The setup for systolic array evaluation is identical to Polybench and packet scheduling, apart from the fact that we target the larger Alveo U250 board. 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 284 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 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 11a–11c 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 [Cadence 2024; Canis et al. 2011; Intel 2021; Mentor Graphics 2021; Pilato and Ferrandi 2013; Zhang et al. 2008] intermix static and dynamic-latency operations, such as dividers. They do so using software ILs like LLVM [Lattner and Adve 2004], 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 [Cheng et al. 2020; Xu et al. 2022] seek to combine aspects of static and dynamic control. DASS [Cheng et al. 2020] 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 [Cheng et al. 2020] or some heuristic [Cheng et al. 2022] 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. [Szafarczyk et al. 2023] 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 [Xu et al. 2022] is a dialect of MLIR [Lattner et al. 2021] 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.

The existing ILs for ADL compilers also give passes control over scheduling, but they focus on either static [Ananian 1998; Sahasrabuddhe et al. 2007; Sinha and Patel 2012] or dynamic interfaces [Josipović et al. 2018; Sharifian et al. 2019; Xu et al. 2023]. In particular, HIR [Majumder and

Bondhugula 2024] 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 [Nigam et al. 2023], 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.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback. Kevin Laeufer also provided feedback on early drafts of this paper.

This research was support by NSF awards #1845952, #2124045, and #2118709 and gifts from Google. Andrew Butt was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Data-Availability Statement

The software that supports Piezo is available on Zenodo [Kim et al. 2024] as a VirtualBox image. The source code for the compiler can be found on Github [The Piezo Authors 2024].

References

AMD 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

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

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

Cadence. 2024. *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

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. https://doi.org/10.1145/1950413.1950423

Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (jun 2024). https://doi.org/10.1145/3656401

Jianyi Cheng, Estibaliz Fraca, John Wickerson, and George A. Constantinides. 2023. Balancing Static Islands in Dynamically Scheduled Circuits Using Continuous Petri Nets. *IEEE Trans. Comput.* (2023). https://doi.org/10.1109/TC.2023.3292590

Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. https://doi.org/10.1145/3373087.3375297

Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Finding and Finessing Static Islands in Dynamically Scheduled Circuits. https://doi.org/10.1145/3490422.3502362

Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. https://doi.org/10.1145/3240765.3240838

Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2020. FBLAS: Streaming Linear Algebra on FPGA. In *International Conference for High Performance Computing, Networking, Storage and Analysis.* https://doi.org/10.1109/SC41405.2020.00063

Robert Dockins. 2012. *Operational refinement for compiler correctness.* Ph. D. Dissertation. Princeton University.

Zhen Dong, Yizhao Gao, Qijing Huang, John Wawrzynek, Hayden K.H. So, and Kurt Keutzer. 2021. HAO: Hardware-aware Neural Architecture Optimization for Efficient Inference. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. https://doi.org/10.1109/FCCM51124.2021.00014

David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. https://doi.org/10.1145/3385412.3385983

Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. 2020. Using Deep Programmability to Put Network Owners in Control. *SIGCOMM Comput. Commun. Rev.* (2020). https://doi.org/10.1145/3431832.3431842

Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. https://doi.org/10.1109/ISCA.2018.00012

Sergi Granell Escalfet. 2023. *Accelerating Halide on an FPGA.* Master's thesis. Universitat Politècnica de Catalunya.

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* (2014). https://doi.org/10.1145/2601097.2601174

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

Intel. 2024a. *Intel Xeon Gold 6230 Processor.* Retrieved August 31, 2024 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

Intel. 2024b. *oneAPI Deep Neural Network Library Developer Guide and Reference.* https://oneapi-src.github.io/oneDNN/

Aihui Jiang, Yufeng Li, Jiangtao Li, and Chenhong Cao. 2022. A Reusable Convolutional Accelerator for CNN on Resource-limited FPGA. In *2022 IEEE Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta)*. https://doi.org/10.1109/SmartWorld-UIC-ATC-ScalCom-DigitalTwin-PriComp-Metaverse56740.2022.00104

Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. https://doi.org/10.1145/3174243.3174264

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. https://doi.org/10.1145/3079856.3080246

Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and Rachit Nigam. 2024. Reproduction Package for "Unifying Static and Dynamic Intermediate Languages for Accelerator Generators". https://doi.org/10.5281/zenodo.13388203

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. https://doi.org/10.1145/3192366.3192379

Hsiang-Tsung Kung. 1982. Why systolic architectures? *IEEE Computer* (1982). https://doi.org/10.1109/MC.1982.1653825

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. https://doi.org/10.1145/3289602.3293910

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. https://doi.org/10.1109/CGO.2004.1281665

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO51591.2021.9370308

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/

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

Kingshuk Majumder and Uday Bondhugula. 2024. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. https://doi.org/10.1145/3623278.3624767

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

Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal Abstractions for Packet Scheduling. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 269 (2023), 25 pages. https://doi.org/10.1145/3622845

Kevin E. Murray and Vaughn Betz. 2014. Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs. https://doi.org/10.1145/2554688.2554786

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. https://doi.org/10.1145/3385412.3385974

Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. https://doi.org/10.1145/3591234

Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. https://doi.org/10.1145/3445814.3446712

Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. https://doi.org/10.1109/FPL.2013.6645550

Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* (2017). https://doi.org/10.1145/3107953

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. https://doi.org/10.1145/2491956.2462176

Sameer D Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P Desai. 2007. AHIR: A hardware intermediate representation for hardware generation from high-level programs. https://doi.org/10.1109/VLSID.2007.28

Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μIR: An Intermediate Representation for Transforming and Optimizing the Microarchitecture of Application Accelerators. https://doi.org/10.1145/3352460.3358292

Rohit Sinha and Hiren Patel. 2012. 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). https://doi.org/10.1109/TCAD.2012.2198474

Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2934872.2934899

Sam Skalicky, Sonia López, Marcin Łukowiak, James Letendre, and David Gasser. 2013. Linear algebra Computations in Heterogeneous Systems. In *Conference on Application-Specific Systems, Architectures and Processors.* https://doi.org/10.1109/ASAP.2013.6567589

Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. https://doi.org/10.1109/FPL60245.2023.00009

The Calyx Authors. 2022. *Disable top-down-st from default compilation pipeline.* Retrieved March 8, 2022 from https://github.com/calyxir/calyx/pull/941

The Calyx Authors. 2023a. *Compress static FSMs.* Retrieved November 22, 2023 from https://github.com/cucapra/calyx/issues/936

The Calyx Authors. 2023b. *Fix top-down static timing.* Retrieved November 22, 2023 from https://github.com/cucapra/calyx/pull/1338

The Calyx Authors. 2023c. *Problems with static FSMs.* Retrieved November 22, 2023 from https://github.com/cucapra/calyx/issues/940

The Piezo Authors. 2024. *Piezo Source Code.* https://github.com/calyxir/calyx

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

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

Jiahui Xu, Emmet Murphy, Jordi Cortadella, and Lana Josipovic. 2023. Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* https://doi.org/10.1145/3543622.3573196

Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A Multi-level Intermediate Representation for Hardware Synthesis Methodologies. In *International Conference On Computer Aided Design (ICCAD)*. https://doi.org/10.1145/3508352.3549370

Zhenya Zang, Uwe Dolinsky, Pietro Ghiglio, Stefano Cherubin, Mehdi Goli, and Shufan Yang. 2023. Building a Reusable and Extensible Automatic Compiler Infrastructure for Reconfigurable Devices. https://doi.org/10.1109/FPL60245.2023.00062

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

Received 2024-04-05; accepted 2024-08-18