MODULAR ABSTRACTIONS FOR EFFICIENT HARDWARE DESIGN

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Rachit Nigam

May 2025

MODULAR ABSTRACTIONS FOR EFFICIENT HARDWARE DESIGN

Rachit Nigam, Ph.D.

Cornell University 2025

Hardware design is primarily concerned with efficiency: the need to implement the fastest circuit using the least amount of resources and power. Coupled with the staggering amounts of resources poured into designing, manufacturing, and deploying hardware, optimization decisions dominate the design of tools for hardware design. Modularity, or the separation of concerns, allows for the design of reusable components and has been a primary driver of the software revolution. However, in hardware design, it has taken a backseat; modular design obfuscates key properties of circuits which may lead to inefficient implementation. In the specialization era, where performance gains are driven by designing hardware for specific computations, the need for modular and efficient abstractions for hardware design is dire.

This thesis identifies *explicit reasoning about time* as a key ingredient for the design of such abstractions and embodies them in three systems. First, Dahlia, an imperative language that compiles to hardware and uses time-sensitive reasoning to ensure that surface programs compile to efficient hardware. Second, Calyx, a compiler and an intermediate language for transforming Dahlia-like languages into hardware descriptions. Calyx bridges the gap between computational descriptions and circuit implementations using a novel intermediate language that mixes software-like control flow and hardware-like structural constructs. Calyx further resolves the tension between precise modeling of cycle-level time and scalable compiler optimizations by exploiting the observation that time-sensitive execu-

tion schedules are a *refinement* of time-insensitive ones. Finally, Filament, a new hardware description language that directly models cycle-level constraints in the interfaces of modules and ensures, at compile-time, that designs do not have any structural hazards. Together, these systems demonstrate that appropriate modeling of time at each abstraction layer is crucial for building hardware design tools that are both modular and efficient. This work provides a foundation for scaling up hardware design in the era of specialized accelerators.

BIOGRAPHICAL SKETCH

Rachit Nigam was born in 1997 in Lucknow, Uttar Pradesh, India. He completed his Bachelor of Science from the University of Massachusetts Amherst in 2018 and graduated from the Honors College with the thesis *"Execution Control for JavaScript"*. During his PhD at Cornell, Rachit moved several times to be a visiting scholar at Facebook Reality Labs, the University of Washington, and the Massachusetts Institute of Technology. Rachit will start as an Assistant Professor of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. Rachit wrote this dissertation from Jamaica Plain, Boston.

To my mother, Dr. Manisha Nigam.

कौन कहता है कि आसमां पे सुराख नहीं होता
कोई तबीयत से पत्थर तो उछालो यारो।

– दुष्यंत कुमार

Who says that we can't make holes in the sky
Just throw a stone with earnestness, my friends

- Dushaynth Kumar

ACKNOWLEDGEMENTS

Many people incorrectly believe that a PhD is about research. A PhD, like all other endeavors in life, is about people. My PhD is no exception: the work contained in this dissertation is an amalgamation of ideas and effort of dozens of people; without them, this work would neither be possible nor worth doing.

Sam Thomas was my first research mentee and watching him grow as a researcher made me realize that I will stay an academic for far too long. His unparalleled creativity coupled with his insatiable curiosity made him a joy to work with. His fingerprints are all over Dahlia and Calyx. Christophe Gyurgik is a force of nature and led the creation of the TVM and CIRCT frontends for Calyx. Caleb Kim and Pai Li together created the Piezo extensions for Calyx which finally enabled it to compete with industrial tools. Ethan Gabizon and Edmund Lam worked on the parametric extensions to Filament and shared in my joy of building type systems. Ted Bauer, Kenneth Fang, Yuwei Ye, YooNa Chang, Karen Zhang, YoungSoek Na, Crystal Hu, Apurva Koti, Alma Thaler, Nathaniel Navarro, Parth Sarkar, and Ananya Goenka all tolerated my mentoring; I hope they learned from me as much as I learned from them.

Certain papers came about as excuses to work with friends. During the long year of the pandemic, between discussions of books and games, Griffin and I worked together on building a simulator for Calyx. Pedro lured me in with his delicious cooking and taught me some category theory (a worthwhile sacrifice); he and I worked on Filament's theory. Unlike me, Alexa thought that program synthesis could be used to make compilers work, so we built one together; she was right. Anshuman joined the lab in my fourth year, and we became quick friends over our shared love of food. He applied his impeccable sense of style to the Piezo paper and pushed it through to the end. Andrew Butt shared my passion for building robust

tools that people want to use and collaborating with him has been exceptionally fun.

Despite my best efforts, my friends kept me sane. Jonathan emphasized the need for shoe horns and joie de vivre. Alex, Armin, and Shreyas were fantastic roommates and easy to live with. Ryan, Rolph, Kiran, and Raunak played music with me. Jesse, Kate, Soham, Mahimna, and Kwang were the original wingz crew and made my first year fun. Sam provided a safe haven in Princeton when I was fed up with Ithaca. Greg and Kiran went on long bike rides with me, full of discussions about everything and nothing. Soham repeatedly told me to quit my PhD and pursue something with real stakes. On this and many other unimportant things, we disagreed. Vishnu was an ever-present anchor in my journey; seeing him throughout my time as a PhD student reminded me that there are other important things in life. Rini and Malena are unapologetically passionate about who they are and everything they do; I am unreasonably lucky to have met them.

After the pandemic, I exiled myself from Cornell and spent a wonderful year at the University of Washington within the PLSE group. Max, Gus, Anjali, Oak, Thia, Andrew, Vishal, and Gilbert made it an easy home. In my time there, Zach Tatlock showed me how to build a community. Instead of graduating, I moved to MIT to hang out with Jonathan Ragan-Kelley's group. Jonathan set the benchmark for what good taste in research looks like for me. Serena, Zoe, Andy, Nishant, Sylee, Jeremy, Anushka, and Edwin gave me a sense of home in Boston.

Many people spent their precious time mentoring me. Chris Batten and Zhiru Zhang taught me about computer architecture and made sure I worked on important problems, instead of merely interesting ones. Nate Foster always demanded a clarity of thought and made sure I did not forget my PL roots. Stephen Neuendorffer and Chris Leary spent countless hours teaching me about important problems

in compilers and hardware design. Shriram Krishnamurthi spent a summer teaching me the beauty of programming languages. As a freshman at UMass, I stumbled into Arjun Guha's office and rambled about algorithms for constructing permutations. Unperturbed, he offered to do research with me and gave me my first shot. Adrian Sampson always chose to be kind to me, even when I was not kind to myself. Long after I have forgotten our technical discussions, I will remember this about him.

My family is my rock and my source of strength and perseverance. My father, Sharad Nigam, took a leap of faith and sent me to the United States to pursue my dream of learning computer science. My sister, Ishita, at the age of six, gave up our time together so I could go study. Finally, my mother, Dr. Manisha Nigam, gave up her career, her time, and her youth to help me become who I am.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hardware design was once the exclusive domain of computer architects, who delivered exponential performance improvements to the software stack without exposing the murky details of transistors. With the end of physical scaling laws that powered this growth, high-performance and energy-efficient computing has increasingly become reliant on specialized *accelerators*—circuits that trade off computational generality for efficiency. However, the tools used to design such accelerators use the same, low-level abstractions that are used to design general-purpose processors. This dissertation examines the design of new programming models for hardware design through the dual lenses of efficiency—the primary concern of hardware design—and modularity—the ability to reuse components—to enable hardware design to scale up in the era of accelerators.

1.1 End of Scaling

Early computer design rode the dual waves of Dennard scaling [48]—which allowed for boosted clock frequencies without additional power—and Moore's law [110]—which predicted shrinkage of transistor feature sizes and better unit economics allowing for increased transistor counts in the same die area. This was an era of plenty: general-purpose processors could justify using increasingly complex mechanisms to extract parallelism from single-threaded programs while simultaneously providing improved clock frequencies. The software stack transparently benefited from these improvements and programmers left hardware design to the architects.

In the early 2000s, Dennard scaling ended, leading to dramatic challenges with

increasing clock frequencies. However, Moore's law still provided additional transistors on the chip and so processor designs pivoted to using multicore designs which utilized multiple independent processors capable of executing programs in parallel. However, this change did not improve the performance of single-threaded programs. To utilize the parallel processing capabilities of multicore systems, programmers had to redesign fundamental abstractions [16, 131] and use specialized abstractions [19, 121] which expose the parallelism of the machine.

However, multicore scaling is limited by two factors. First, even if a workload can perfectly utilize the parallelism of the machine, it is limited by Amdahl's law [8, 77]: if 90% of the workload is parallelizable and 10% is sequential, the maximum achievable speedup is 10× of the original program since the performance of the sequential section becomes the bottleneck in the presence of excess parallel resources. Second, a chip design is limited by the need to efficiently disperse heat leading to a power wall [53, 93]. While modern chips have continued to see increased transistor counts, they are unable to keep all the transistors powered-on at the same time. These dual problems mean that multicore scaling has provided a limited solution to the end of Dennard scaling.

Accelerators tackle both challenges by sacrificing computational generality for efficiency [75]. By focusing on particular computations, accelerators can utilize specialized computational circuits and memory hierarchies that exploit domain-specific knowledge. This tackles the Amdahl's law limit by improving the performance of sequential code as well as the power wall challenge since accelerators can be strategically powered-on for specific computations. Originally reserved for well-understood computations like encryption and audio-video decoding, accelerators have seen use in diverse areas like networking [10, 99, 113, 114], machine learn-

ing [31, 32, 61, 84], genomics [59, 152], databases [28, 158, 159], zero-knowledge proofs [137], etc. Like the multicore era, this ubiquitous adoption has led to a reexamination of fundamental system-level abstractions [62, 100], vertically integrated, language-to-accelerator compiler pipelines [9, 64, 126, 135], and widespread interest in customized hardware design.

This dissertation re-examines another fundamental abstraction in the context of these trends: the languages and tools used to design and implement hardware.

1.2 Tools to Build Hardware

Chip design progresses roughly at four levels of abstraction: functional, timed, netlist, and physical. Higher-level abstractions are suited for design-space exploration of architectural features such as functional units, memory hierarchies while lower-level abstractions decide the physical characteristics—placement of logical gates, routing, fan-out factors—which influence the performance of the final design. Because of the costs involved, chip design largely progresses in a waterfall manner: once the architectural features are finalized at the functional level, they are not changed during physical modeling.[1]

Early hardware design occurred at the physical level since it allowed engineers to have low-level control on the placement, sizing, and routing between transistors. Hardware description languages (HDLs) like Verilog [78] and VHDL [79] were introduced to *model the behavior* of these physical circuits and to catch bugs before implementation occurred. Because of this, HDLs were primarily designed for fast

[1] Certain physical constraints—such as the length of wires between compute and memory—might require architectural changes and are only discovered during the final stages. However, chip design methodologies aim to limit the impact of such changes [148].

and efficient simulation [55]. However, with the introduction of standard cell-based design [106] and improvements in tools for logic synthesis [22], users were able to compile HDL programs to circuit implementations. The convenience of using the same description for specification and implementation quickly allowed HDLs to become the dominant tool for hardware design.

However, the fundamental mismatch between the simulation-focused abstractions used by HDLs and the physical nature of circuits has led to repeated problems in scaling up hardware design [146, 147]. Modular design—the ability to separately implement components and reuse them across programs—has been a key mechanism for scaling up the software ecosystem but has largely been absent from hardware design. While designers can license coarse-grained intellectual property (IP) blocks like processor cores or USB drivers, there is little reuse of lower-level building blocks. This is because, unlike software programming, hardware design is primarily concerned with efficiency; if modularity obfuscates details for efficient design, it is discarded. Given this, designers end up implementing their own components for each design to achieve maximum performance.

Modularity in hardware design is a widely studied topic with many proposed solutions: using software programs to generate circuit implementations [107, 139], embedding HDLs in software languages [1, 4, 12, 13, 40, 81, 101], transactional semantics for hardware design [21, 119], transforming existing software languages to circuit implementations [5, 25, 124, 157, 167], and most recently, using domain-specific languages for hardware design [29, 44, 52, 73, 74, 89, 94, 95]. While promising in different ways, these solutions have not provided a principled answer to the fundamental tension between efficiency and modularity.

1.3 My Thesis

> *Explicitly reasoning about time enables the design of modular and efficient abstractions for hardware design.*

Hardware design, unlike software programs, has to reason about *structure*—the physical circuits that perform computations—and *time*—how the circuits are reused over time to perform logical computations. Programming abstractions for hardware design have focused on precisely expressing structure but not time. For example, traditional HDLs like Verilog and VHDL, owing to their roots as simulation-focused languages, use event-driven abstractions to model the behavior of hardware ("if event E occurs, perform action A"). While this yields efficient simulators, it does not provide first-class reasoning for clock signals which are the primary way to model time in synchronous circuits. Similarly, while high-level programming models provide mechanisms to control how much circuitry is allocated—through code annotations that unroll loops or mark out particular arrays to be mapped onto physical memories—they do not reason about the timing constraints of these resources. Finally, compilers that transform high-level programs to circuit implementations pick representations of timing behavior and corner themselves into inflexible or inefficient implementation choices.

This dissertation demonstrates that by appropriately modeling time, each abstraction layer can provide efficient and predictable programming models. The representation of time necessarily depends on the level of abstraction: cycle-level time might be appropriate for new HDLs but too low-level for domain-specific languages.

1.4 Organization

This dissertation presents three systems developed at different levels of abstractions to support my thesis:

- Dahlia (§3) C-like programming language that compiles to circuit implementations. Dahlia models the timing constraints of circuit resources using *time-sensitive affine types*—a novel, sub-structural type system—and demonstrates that well-typed programs express *predictable* resource-performance trade-offs.
- Calyx (§§ 4–6), an intermediate language (IL) and a compiler infrastructure for transforming domain-specific languages to hardware designs. Calyx accomplishes the tightrope walk of providing precise hardware description with scalable compiler analyses by intermixing software-like control-flow operators—loops, conditionals—with hardware-like structure—modules, wires, cycle-level timing. This is enabled by two ideas: *groups* which are similar to basic blocks in software compilers but instead encapsulate hardware structure, and time-sensitive reasoning through *refinement*. Together, Calyx has enabled the design of several frontend compilers, is adopted by the LLVM CIRCT compiler infrastructure, provides dozens of optimizations, and is the basis of several tools that utilize its IL.
- Filament (§§ 7 and 8), a new hardware description language that guarantees that well-typed programs do not have pipelining bugs. Unlike other typed HDLs [12, 13, 119], Filament's type system reasons about resource reuse and ensures that pipelining cannot introduce hard-to-debug issues caused by structural hazards. Furthermore, because Filament utilizes a type system to

ensure this property, once compiled, Filament designs are as efficient as hand-written HDL programs. We extend Filament's guarantees to parameterized programs allowing us to eliminate pipelining bugs from *families of circuits*.

Chapter 2 overviews the evolution of programming models for hardware design. Chapter 3 describes the Dahlia programming language and time-sensitive affine type which enable predictable resource-performance trade offs. Chapters 4 and 5 overview the design of the Calyx intermediate language and compiler infrastructure. Chapter 6 evaluates the Calyx infrastructure by describing several frontend compilers and analyzing the impact of its optimization passes. Chapter 7 discusses the Filament hardware description language which uses a type system to guarantee that hardware designs are free of pipelining bugs at compile time. Chapter 8 extends Filament's guarantees to parameteric programs enabling it to reason about families of circuits. Chapter 9 concludes the thesis by discussion future directions.

1.5 Previously Published Material

This thesis draws upon the work of the following previously published materials:

- Chapter 3: 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. doi: 10.1145/3385412.3385974 [115].
- Chapters 4 and 6: Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *ACM In-*

ternational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. doi: 10.1145/3445814.3446712 [116].

- Chapters 5 and 6: Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and Rachit Nigam. Unifying static and dynamic intermediate languages for accelerator generators. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2024. doi: 10.1145/3689790 [86].
- Chapter 7: 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. doi: 10.1145/3591234 [117].
- Chapter 8: Rachit Nigam, Ethan Gabizon, Edmund Lam, and Adrian Sampson. Correct and compositional hardware generators. 2024. doi: arXiv: 2401.02570 [118].

CHAPTER 2

PROGRAMMING MODELS FOR HARDWARE DESIGN

This chapter overviews various programming models for hardware design from traditional hardware description languages (HDLs) to high-level programming models that transform software languages to circuit implementations.

2.1 Basics of Hardware Design

The goal of hardware design is to build the smallest, fastest, most energy-efficient circuit to perform the task you want. For example, a general-purpose processor aims to execute an instruction stream as fast as possible within its power and area budgets. We provide a brief overview of the physical constraints involved in hardware design to discuss the merits of various programming models for hardware design.

Registers, combinational logic, and clock Digital circuits are built using two fundamental components: registers and combinational logic. Registers are collections of flip-flops, each storing a single bit of information. Combinational logic consists of logic gates, such as `AND` or `XOR`, which implement pure boolean functions. Each logic gate has a *propagation delay*, the time it takes for the output to change after the input changes. A digital circuit can be viewed as a cloud of combinational logic connected to a large register that stores the state of the circuit.

Most digital designs use a *clock signal* to synchronize register updates due to varying propagation delays. At the start of each clock cycle, registers update their output wires, and the signals propagate through the combinational logic. The

resulting outputs are then written back into the registers at the end of the clock cycle. To ensure synchronous updates, the clock signal must be slower than the propagation delay of the slowest path through the combinational logic, known as the *critical path*. The critical path determines the maximum clock frequency.

Power, performance, and area (PPA) The three metrics used to evaluate a design. The area of the design accounts for implementation resources such as transistors and wires while performance refers to the application or task-specific measures as the instructions-per-cycle (IPC) of a processor. The power consumption of a design depends on the transistor technology, clock frequency, and the resources used and is divided into dynamic power—consumed when transistors switch—and static power—lost due to leakage effects. For example, increasing the number of registers will increase the static power—since each register will leak power—while increasing the clock frequency will increase the dynamic power—since the circuit switches more often.

There are only trade-offs PPA constraints induce a complex trade-off space for hardware designs: there are rarely obvious optimizations. For instance, a long combinational path might limit a circuit's clock frequency. Adding a register can break the computation into two steps, but this has both functional and physical consequences. By splitting the computation across two cycles, the output appears one cycle later requiring adjustments to signal consumers. The additional registers increase power consumption and the clock signal improvement might be negated by the signal delay. Effective hardware design requires understanding the design space and exploiting trade-offs to achieve PPA goals. Programming models for hardware design should help designers express and manage these trade-offs.

2.2 Hardware Description Languages

Early HDLs [36, 56, 76] evolved as specification languages intended to describe the behavior of digital circuits. HDLs used the register transfer level (RTL) abstraction which views hardware designs as a cloud of logic connected to a set of registers. Hardware implementation, on the other hand, used *structural* abstractions such as netlists, which represent hardware as a collection of logic gates connected by wires. These abstractions were reconciled after the VLSI revolution [106], which standardized transistor features, and improvements in logic synthesis techniques [22], which automatically transformed RTL programs to netlists. Verilog was originally designed to simulate designs described at both RTL and netlist levels [55], and repurposed for hardware implementation with logic synthesis tools like Synopsys's Design Compiler [2].

Event-driven abstractions RTL languages like Verilog provide event-driven constructs to model the behavior of circuits. The following example implements the logic for a half-adder:

```
reg sum, carry;
always @(a or b) begin
    sum = a ^ b;
    carry = a & b;
end
```

The `always` construct in Verilog models event-driven behavior: when the values of either `a` or `b` change, the computation within the block is triggered, updating the values of `sum` and `carry` (which might themselves trigger a computation). On the other hand, a netlist-level implementation instantiates low-level modules like `XOR` and `AND`, which might be a part of the standard library for a particular process design kit (PDK), to represent the hardware design closer to physical reality:

```
reg sum, carry;
XOR xor1 (.a(a), .b(b), .y(sum));
AND and1 (.a(a), .b(b), .y(carry));
```

The netlist simply describes the connections between modules; it has no event-driven behavior. Logic synthesis tools transform event-driven models of hardware to netlists and need to infer intent from source programs which is often tricky to get right [146, 147]. The abstraction mismatch has a deeper consequence: key properties of hardware designs, like clock signals and resource reuse are not described within a hardware description. Instead, these properties are implicitly reasoned about by programmers and inferred by tools.

Parametric design HDLs like SystemVerilog [11] (a set of extensions to Verilog) provide compile-time constructs like loops and conditionals to produce different circuits based on compile-time values. For example, an adder might be parameterized using the input bitwidth and produce different circuits based on the compile-time value. Parameterization can enable design reuse since users can customize components based on the particular context. Unlike more powerful metaprogramming systems in software languages [54], these language constructs cannot introspect on the circuit being generated and therefore cannot express advanced transformations like automatically adding extra registers to break up long computations.

Embedded hardware description languages Embedded hardware description languages (eHDLs) embed themselves in a software host language like Scala, Python, or OCaml to provide metaprogramming capabilities to users [12, 40, 81, 101]. This is accomplished by providing a deep embedding of the HDL constructs within the software language allowing it to construct, manipulate, and introspect HDL programs. While such languages improve design productivity, they rely on

RTL abstractions and are bound by their limitations.

2.3 High-Level Programming Models

High-level programming models eschew the RTL abstractions of HDLs and instead transform familiar computational abstractions of software languages, to circuit implementations. Broadly, there are two approaches for this: transforming general-purpose programming languages like C or Python to circuit designs, or designing custom domain-specific languages (DSLs) to express specific classes of computations and transforming them to hardware designs.

2.3.1 High-Level Synthesis

Instead of mapping simple operators such as adders, high-level synthesis (HLS) tools aim to automatically compile entire programs written in software languages into hardware designs. Early tools [5, 91] avoided the inherently parallel nature of HDLs and instead adopted the sequential semantics of software languages. In these languages, each assignment would correspond to a single clock cycle.[1] For example, both the assignments $x = a + b$ and $y = a + b + c + d$ would be scheduled to execute in one cycle. This relationship between program order and cycle-level behavior is often understated but is essential for effective hardware synthesis and providing sufficient control to designers to meet PPA objectives.

Modern tools incorporate pipeline synthesis [71, 162, 167] to automatically break long datapath computation and generate control logic. The relationship be-

[1] Cong et al. [43, §2] provide a more complete overview of these early tools.

tween program order and cycle-level timing is often completely abstracted away in such tools and touted as a feature to enable software programmers to build their own hardware. However, in practice, meeting PPA objectives requires substantial understanding of the cycle-level behavior of generated hardware and structuring code in particular ways to achieve the desired microarchitecture.[2]

The goal of high-level programming models is to provide improved design productivity. A key aspect of design productivity is being able to understand and optimize a design. However, existing languages often obscure the relationship between program order and cycle-level behavior, leading to unpredictable trade-offs. Chapter 3 overviews the design of a new HLS language that uses a novel type system to connect program order to cycle-level behavior, enabling designers to make predictable and informed trade-offs.

2.3.2 Domain-Specific Languages

DSLs [20, 87, 88, 129, 145, 149, 166] provide domain-specific abstractions that allow users to concisely express computations and compilers to aggressively optimize implementations. DSLs for hardware generation generally focus on generating domain-specific architectures such as streaming pipelines [52, 73, 74, 128] or systolic arrays [44, 94]. By limiting both the input language and the generatable micro-architectural features, DSLs for hardware generation can generate high-performance accelerators while improving design productivity.

While powerful in theory, to fully take advantage of a DSL, implementors must design optimizing compilers that generate designs competitive with hand-

[2]Verilog users experience similar problems when using *behavioral synthesis* tools which attempt to automatically compile arbitrary Verilog to hardware.

written code. Such compilers are Herculean efforts requiring expertise across language design, compiler optimization, and hardware design. Compiler infrastructure like LLVM [96] centralize such expertise and allow implementors to rapidly design a high-performance DSL. Such infrastructures must balance between flexibility—which allows many languages to target them—and encoding target constraints—which allows for optimization and efficient code generation.

Compiler infrastructures for hardware generation bridge the gap between computational specifications and circuit implementations and must encode both types of abstractions. The abstraction for modeling time shows up as a key design choice: latency-insensitive interfaces [27] abstract away timing details from communication channels between modules but can impose high performance and validation costs [111] while latency-sensitive interfaces expose timing details provide efficient interfacing but can be difficult to maintain. Compiler infrastructures for hardware design have proposed using latency-sensitive [103, 141], latency-insensitive [83], and a combination of the two [33, 140, 163]. This choice of representation constrains the types of optimizations the compiler can perform. A latency-insensitive design prevents the compiler from reasoning about the timing behavior of modules, which is essential for generating efficient interfaces. Conversely, a latency-sensitive design requires the compiler to assume implicit synchronization across different parts of the design by counting clock cycles, which can hinder the scalability of compiler analyses.

Chapter 4 overviews the design of Calyx, an intermediate language (IL) and a compiler infrastructure that intermixes software-like control flow with hardware-like structure. Calyx's *group* abstraction acts like basic blocks in software compilers but instead encapsulates hardware circuits. Execution of groups is scheduled using

control operators like `while` loops and conditionals which allow the compiler to optimize the design's control flow as well as hardware structure. Groups and control together allow for precise specification of hardware *and* scalable analyses. Chapter 5 shows how Calyx can accommodate both latency-sensitive and latency-insensitive reasoning within the same IL by observing that latency-sensitive execution schedules are a *refinement* of latency-insensitive ones.

2.4 Summary

With the growing interest in accelerator design, numerous new programming models for hardware design are emerging at various levels of abstraction. Each abstraction offers distinct trade-offs: modern HDLs provide detailed control over circuit specifications, while high-level programming models bridge computational descriptions with hardware implementations. However, to be effective, each abstraction must address both the structural and temporal aspects of hardware design, enabling robust reasoning and optimization. The remainder of this dissertation explores the design of programming models at various levels of abstraction and demonstrates how explicitly modeling timing behavior enables modular and efficient hardware design.

CHAPTER 3

PREDICTABLE ACCELERATOR DESIGN WITH TIME-SENSITIVE AFFINE TYPES

High-level synthesis (HLS) tools offer an alternative programming model for designing and automatically generating hardware. Such tools are a particularly compelling idea for programming reconfigurable hardware such as field programmable gate array (FPGA). However, existing HLS tools repurpose legacy languages such as C or C++ and use ad-hoc code annotations (called pragmas) to specify crucial features of the resulting design such as the memory partitioning, pipelining, and vectorized execution. While HLS tools enable users to rapidly generate a working hardware design, the absence of predictability means that users are unable to quickly iterate and optimize their implementations. Seemingly small changes to source-level programs cause dramatically different and unexpected results in the final design. This *unpredictability* makes it challenging for the user to develop a precise mental model to optimize against.

This chapter identifies the underlying problem: efficient design requires carefully understanding and working with *structural* and *temporal* constraints of circuits; however, HLS tools attempt to completely abstract away this information thereby limiting the user's ability to understand why a design behave poorly and how to make it work better. The solution here is to use a novel substructural type system to surface these constraints at the source-level enabling users to retain the productivity of HLS tools while being able to precise reason about hardware constraints.

The central insight is that we can extend an affine type system [151] to model physical constraints of circuits. Components in a hardware design are finite and

expendable: a subcircuit or a memory can only do one thing at a time, so a program needs to avoid conflicting uses. Previous research has shown how to apply substructural type systems to model classic computational resources such as memory allocations and file handles [18, 70, 105, 151] and to enforce exclusion for safe shared-memory parallelism [14, 39, 67]. Unlike those classic resources, however, the availability of hardware components changes with time. We extend affine types with *time sensitivity* to express that repeated uses of the same hardware is safe as long as they are temporally separated.

While all circuit components have reuse constraints, we focus solely on reasoning about physical memories. This is because HLS compilers have a lot more leeway in handling more computational circuits; if a loop needs more multipliers, the tool can just generate more multipliers. However, once the shape of the memory is specified by the user (through pragmas and other mechanisms), it must be respected by the tool. Therefore, all other hardware reuse constraints are limited by reasoning about memory.

This chapter embodies these ideas in Dahlia, a predictable high-level synthesis language. Section 3.1 studies the predictability pitfalls with HLS tools. Section 3.2 describes the Dahlia language through examples. Section 3.3 develops the formalism of *time-sensitive affine types*. Section 3.4 empirically demonstrates Dahlia's effectiveness in rejecting unpredictable designs and make area-performance tradeoffs in common accelerator designs.

3.1 Predictability Pitfalls in Traditional HLS

Figure 3.1 depicts the design of a traditional high-level synthesis (HLS) compiler. A typical HLS tool adopts an existing open-source C/C++ frontend and adds

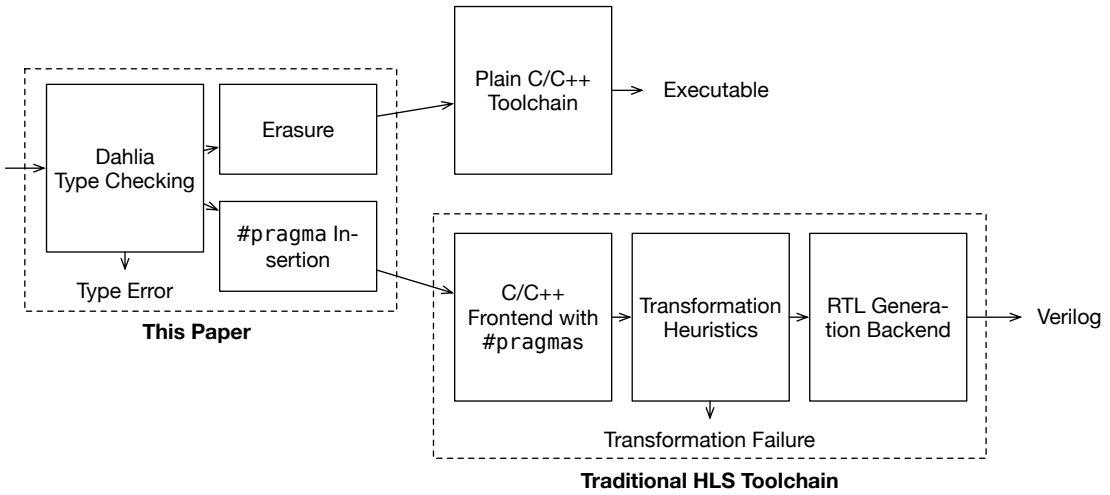


Figure 3.1: Overview of a traditional high-level synthesis toolchain and how Dahlia layers type safety on top.

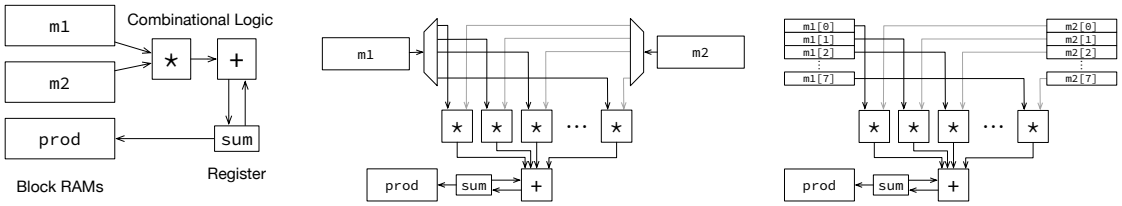
a set of *transformation heuristics* that attempt to map software constructs onto hardware elements along with a backend that generates RTL code [24, 42]. The transformation step typically relies on a constraint solver, such as an LP or SAT solver, to satisfy resource, layout, and timing requirements [41, 71]. Programmers can add `#pragma` hints to guide the transformation—for example, to duplicate loop bodies or to share functional units.

HLS tools are best-effort compilers: they make a heuristic effort to translate *any* valid C/C++ program to RTL, regardless of the consequences for the generated accelerator architecture. Sometimes, the mapping constraints are unsatisfiable, so the compiler selectively ignores some `#pragma` hints or issues an error. The generated accelerator's efficiency depends on the interaction between the code, the hints, and the transformation heuristics that use them.

The standard approach prioritizes automation over predictability. Small code changes can yield large shifts in the generated architecture. When performance is poor, the compiler provides little guidance about how to improve it. Pruning such

```
int m1[512][512], m2[512][512], prod[512][512];
int sum;
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum; } }
```

Figure 3.2: Dense matrix multiplication in HLS-friendly C.



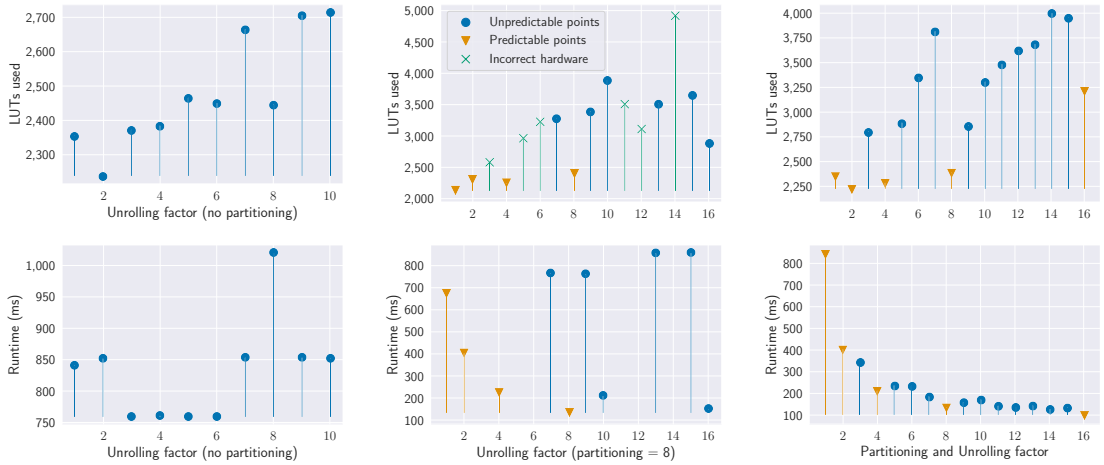
(a) Original. (b) Unrolled. (c) Unrolled and Banked.

Figure 3.3: Three accelerator implementations for code in Figure 3.2.

unpredictable points from the design space would let programmers explore smaller, smoother parameter spaces.

3.1.1 An Example in HLS

Programming with HLS centers on arrays and loops, which correspond to memory banks and logic blocks. Figure 3.2 shows the C code for a matrix multiplication kernel. This section imagines the journey of a programmer attempting to use HLS to generate a fast FPGA-based accelerator from this code. We use Xilinx's SDAccel [161] compiler (v2018.3.op) and target an UltraScale+ VU9P FGPA on an AWS F1 instance [7] to perform the experiments in this section.



(a) Unrolling without partitioning.

(b) Unrolling with 8-way partitioning.

(c) Unrolling and banking in lockstep.

Figure 3.4: Look-up table count (top) and execution latency (bottom) for the kernel in Figure 3.2 with varying parameters.

Initial accelerator Our imaginary programmer might first try compiling the code verbatim. The HLS tool maps the arrays `m1`, `m2`, and `prod` onto on-chip memories. FPGAs have SRAM arrays, called *block RAMs* (BRAMs), that the compiler allocates for this purpose. The loop body becomes combinational logic consisting of a multiplier, an adder, and an accumulator register. Figure 3.3a depicts this configuration.

This design, while functional, does not harness any parallelism that an FPGA can offer. The two key metrics for evaluating an accelerator design are performance and area, i.e., the amount of physical chip resources that the accelerator occupies. This initial configuration computes the matrix product in 841.1 ms and occupies 2,355 of the device's lookup tables (LUTs). However, the target FPGA device has over 1 million LUTs, so the programmer's next job is to expend more of the FPGA area to improve performance.

Loop unrolling The standard tool that HLS offers for expressing parallelism is an `UNROLL` annotation, which duplicates the logic for a loop body. A programmer might attempt to obtain a better accelerator design by adding this annotation to the innermost loop on lines 6–8 in Figure 3.2:

```
#pragma HLS UNROLL FACTOR=8
```

This unrolling directive instructs the HLS tool to create 8 copies of the multiplier and adder, called *processing elements* (PEs), and attempt to run them in parallel. Loop unrolling represents an area–performance trade-off: programmers can reasonably expect greater unrolling factors to consume more of the FPGA chip but yield lower-latency execution.

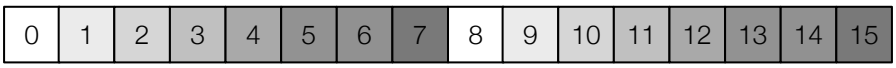
The `UNROLL` directive alone, however, fails to achieve this objective. Figure 3.4a shows the effect of various unrolling factors on this code in area (LUT count) and performance (latency). There is no clear trend: greater unrolling yields unpredictably better and worse designs. The problem is that the accelerator's memories now bottleneck the parallelism provided by the PEs. The BRAMs in an FPGA have a fixed, small number of *ports*, so they can only service one or two reads or writes at a time. So while the HLS tool obeys the programmer's `UNROLL` request to duplicate PEs, its scheduling must serialize their execution. Figure 3.3b shows how the HLS tool must insert additional *multiplexing* hardware to connect the multipliers to the single-ported memories. The additional hardware and the lack of parallelism yields the unpredictable performance and area for different PE counts.

Memory banking to match parallelism To achieve expected speedups from parallelism, accelerators need to use multiple memories. HLS tools provide annotations to *partition* arrays, allocating multiple BRAMs and increasing the access throughput. The programmer can insert these partitioning annotations to allocate

8 BRAMs per input memory:

```
#pragma HLS ARRAY_PARTITION VARIABLE=m1 FACTOR=8
#pragma HLS ARRAY_PARTITION VARIABLE=m2 FACTOR=8
```

Banking uses several physical memories, each of which stores a subset of the array's data. The compiler partitions the array using a "round-robin" policy to enable parallel access. In this example, elements 0 and 8 go in bank 0, elements 1 and 9 go in bank 1, etc.:



(Each shade represents a different memory bank.) Figure 3.3c shows the resulting architecture, which requires no multiplexing and allows memory parallel access.

Combining banking and unrolling, however, unearths another source of unpredictable performance. While the HLS tool produces a good result when both the banking factors and the loop unrolling factor are 8, other design choices perform worse. Figure 3.4b shows the effect of varying the unrolling factor while keeping the arrays partitioned with factor 8. Again, the area and performance varies unpredictably with the unrolling factor. Reducing the unrolling factor from 9 to 8 can counter-intuitively *improve* both performance and area. In our experiments, some unrolling factors yield hardware that produces incorrect results. (We show the area but omit the running time for these configurations.)

The problem is that some partitioning/unrolling combinations yield much simpler hardware than others. When both the unrolling and the banking factors are 8, each parallel PE need only access a single bank, as in Figure 3.3c. The first PE needs to access elements 0, 8, 16, and so on—and because the array elements are "striped" across the banks, all of these values live in the first bank. With unrolling

factor 9, however, the first PE needs to access values from *every* bank, which requires complicated memory indirection hardware. With unrolling factor 4, the indirection cost is smaller—the first PE needs to access only bank 0 and bank 4.

From the programmer's perspective, the HLS compiler silently enforces an unwritten rule: *When the unrolling factor divides the banking factor, the area is good and parallelism predictably improves performance. Otherwise, all bets are off.* Figure 3.4b labels the points where the unrolling factor divides the banking factor as *predictable points*. The HLS compiler emits no errors or warnings for any parameter setting.

Banking vs. array size Even if we imagine that a programmer carefully ensures that banking factors exactly match unrolling factors, another pitfall awaits them when choosing the amount of parallelism. Figure 3.4c shows the effects of varying the banking and unrolling factor in our kernel *together*. The LUT count again varies wildly.

The problem is that, when the banking and unrolling factors do not evenly divide the sizes of the arrays involved, the accelerator needs extra hardware to cope with the "leftover" elements. The memory banks are unevenly sized, and the PEs need extra hardware to selectively disable themselves on the final iteration to avoid out-of-bounds memory accesses.

Again, there is a predictable subset of design points when the programmer obeys the unwritten rule: *An array's banking factor should divide the array size.* Figure 3.4c highlights the predictable points that follow this rule. Among this subset, the performance reliably improves with increasing parallelism and the area cost scales proportionally.

3.1.2 Enforcing the Unwritten Rules

The underlying problem in each of these sources of unpredictability is that the traditional HLS tool prioritizes automation over programmer control. While automation can seem convenient, mapping heuristics give rise to implicit rules that, when violated, silently produce bad hardware instead of reporting a useful error.

We instead prioritizes the predictability of hardware generation and making architectural decisions obvious in the source code. HLS tools *already* contain such a predictable subset hidden within their unrestricted input language. By modeling resource constraints, we can separate out this well-behaved fragment. Figure 3.1 shows how our checker augments a traditional HLS toolchain by lifting hidden compiler reasoning into the source code and rejecting potentially unpredictable programs.

The challenge, however, is that the "unwritten rules" of HLS are never explicitly encoded anywhere—they arise implicitly from non-local interactions between program structure, hints, and heuristics. A naïve syntactic enforcement strategy would be too conservative—it would struggle to allow flexible, fine-grained sharing of hardware resources.

We design a type system that models the constraints of hardware implementation to enforce these constraints in a composable, formal way. Our type system addresses *target-independent* issues—it prevents problems that would occur even on an arbitrarily large FPGA. We do not attempt to rule out resource exhaustion problems because they would tie programs to specific target devices. We see that kind of quantitative resource reasoning as important future work.

3.2 The Dahlia Language

Dahlia's type system enforces a safety property: that the number of simultaneous reads and writes to a given memory bank may not exceed the number of ports. While traditional HLS tools enforce this requirement with scheduling heuristics, Dahlia enforces it at the source level using types.

The key ideas in Dahlia are (1) using substructural typing to reason about consumable hardware resources and (2) expressing time ordering in the language to reason about when resources are available. This section describes these two core features (§§ 3.2.1 and 3.2.2) and then shows how Dahlia builds on them to yield a language that is flexible enough to express real programs (§§ 3.2.3–3.2.6).

3.2.1 Affine Memory Types

The foundation of Dahlia's type system is its reasoning about memories. The problem in Section 3.1.1's example is conflicting simultaneous accesses to the design's memories. The number of reads and writes supported by a memory per cycle is limited by the number of ports in the memory. HLS tools automatically detect potential read/write conflicts and schedule accesses across clock cycles to avoid errors. Dahlia instead makes this reasoning about conflicts explicit by enforcing an affine restriction on memories.

Memories are defined by giving their type and size:

```
let A: float[10];
```

The type of `A` is `mem float[10]`, denoting a single-ported memory that holds 10 floating-point values. Each Dahlia memory corresponds to an on-chip BRAM in

the FPGA. Memories resemble C or Java arrays: programs read and mutate the contents via subscripting, as in `A[5] := 4.2`. Because they represent static physical resources in the generated hardware, memory types differ from plain value types like `float` by preventing duplication and aliasing:

```
let x = A[0];  // OK: x is a float.
let B = A;     // Error: cannot copy memories.
```

The affine restriction on memories disallows reads and writes to a memory that might occur at the same time:

```
let x = A[0];  // OK
A[1] := 1;     // Error: Previous read consumed A.
```

While type-checking `A`, the Dahlia compiler removes `A` from the typing context. Subsequent uses of `A` are errors, with one exception: identical reads to the same memory location are allowed. This program is valid, for example:

```
let x = A[0];
let y = A[0];  // OK: Reading the same address.
```

The type system uses access capabilities to check reads and writes [57, 68]. A read expression such as `A[0]` acquires a non-affine *read capability* for index 0 in the current scope, which permits unlimited reads to the same location but prevents the acquisition of other capabilities for `A`. The generated hardware reads once from `A` and distributes the result to both variables `x` and `y`, as in this equivalent code:

```
let tmp = A[0];  let x = tmp;  let y = tmp;
```

However, memory writes use affine *write capabilities*, which are use-once resources: multiple simultaneous writes to the same memory location remain illegal.

3.2.2 Ordered and Unordered Composition

A key HLS optimization is parallelizing execution of independent code. This optimization lets HLS compilers parallelize and reorder dependency-free statements connected by `;` when the hardware constraints allow it—critically, when they do not need to access the same memory banks.

Dahlia makes these parallelism opportunities explicit by distinguishing between *ordered* and *unordered* composition. The C-style `;` connector is unordered: the compiler is free to reorder and parallelize the statements on either side while respecting their data dependencies. A second connector, `---`, is ordered: in `A --- B`, statement `A` must execute before `B`.

Dahlia prevents resource conflicts in unordered composition but allows two statements in ordered composition to use the same resources. For example, Dahlia accepts this program that would be illegal when joined by the `;` connector:

```
let x = A[0]
---
A[1] := 1
```

In the type checker, ordered composition *restores* the affine resources that were consumed in the first command before checking the second command. The capabilities for all memories are discarded, and the program can acquire fresh capabilities to read and write any memory.

Together, ordered and unordered composition can express complex concurrent designs:

```
let A: float[10];  let B: float[10];
{
  let x = A[0] + 1
  ---
  B[1] := A[1] + x  // OK
```

```
};
let y = B[0];  // Error: B already consumed.
```

The statements composed with --- are ordered with each other but *unordered* with the last line. The read therefore must not conflict with either of the first two statements.

Logical time From the programmer's perspective, a chain of ordered computations executes over a series of *logical time steps*. Logical time in Dahlia does not directly reflect physical time (i.e., clock cycles). Instead, the HLS backend is responsible for allocating cycles to logical time steps in a way that preserves the ordering of memory accesses. For example, a long logical time step containing an integer division might require multiple clock cycles to complete, and the compiler may optimize away unneeded time steps that do not separate memory accesses. Regardless of optimizations, however, a well-typed Dahlia program requires at least enough ordered composition to ensure that memory accesses do not conflict.

Local variables as wires & registers Local variables, defined using the `let` construct, do not share the affine restrictions of memories. Programs can freely read and write to local variables without restriction, and unordered composition respects the dependencies induced by local variables:

```
let x = 0;  x := x + 1;  let y = x;  // All OK
```

In hardware, local variables manifest as wires or registers. The choice depends on the allocation of physical clock cycles: values that persist across clock cycles require registers. Consider this example consisting of two logical time steps:

```
let x = A[0] + 1  ---  B[0] := A[1] + x
```

The compiler must implement the two logical time steps in different clock cycles,

so it must use a register to hold the value of `x`. In the absence of optimizations, registers appear whenever a variable's live range crosses a logical time step boundary. Therefore, programmers can minimize the use of registers by reducing the live ranges of variables or by reducing the amount of sequential composition.

3.2.3 Memory Banking

As Section 3.1.1 details, HLS tools can *bank* memories into disjoint components to allow parallel access. Dahlia memory declarations support bank annotations:

```
let A: float[8 bank 4];
```

In a memory type `mem` t`[`n `bank` m`]`, the banking factor m must evenly divide the size n to yield equally-sized banks. HLS tools, in contrast, allow uneven banking and silently insert additional hardware to account for it (see Section 3.1.1).

Affine restrictions for banks Dahlia tracks an affine resource for each memory bank. To physically address a bank, the syntax M{b}`[`i`]` denotes the ith element of M's bth bank. This program is legal, for example:

```
let A: float[10 bank 2];
A{0}[0] := 1;
A{1}[0] := 2;  // OK: Accessing a different bank.
```

Dahlia also supports logical indexing into banked arrays using the syntax M`[`n`]` for literals n. For example, `A[1]` is equivalent to `A{1}[0]` above. Because the index is static, the type checker can automatically deduce the bank and offset.

Multi-ported memories Dahlia also supports reasoning about multi-ported memories. This syntax declares a memory where each bank has two read/write

ports:

```
let A: float{2}[10];
```

A memory provides k affine resources per bank where k is the number of ports in a memory. This rule lets multi-ported memories provide multiple read/write capabilities in each logical time step. For example, Dahlia accepts this program:

```
let A: float{2}[10];
let x = A[0];
A[1] := x + 1;
```

Dahlia does not guarantee data-race freedom in the presence of multi-ported memories. Programs are free to write to and read from the same memory location in the same logical time step and should expect the semantics of the underlying memory technology. Extensions to rule out data races would resemble race detection for parallel software [112, 127].

Multi-dimensional banking Banking generalizes to multi-dimensional arrays. Every dimension can have an independent banking factor. This two-dimensional memory has two banks in each dimension, a total of $2 \times 2 = 4$ banks:

```
let M: float[4 bank 2][4 bank 2];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The physical and logical memory access syntax similarly generalizes to multiple dimensions. For example, `M{3}[0]` represents the element logically located at `M[1][1]`.

3.2.4 Loops and Unrolling

Fine-grained parallelism is an essential optimization in hardware accelerator design. Accelerator designers duplicate a block of logic to trade off area for performance: n copies of the same logic consume n times as much area while offering a theoretical n-way speedup. Dahlia syntactically separates parallelizable *doall* `for` loops, which must not have any cross-iteration dependencies, from sequential `while` loops, which may have dependencies but are not parallelizable. Programmers can mark `for` loops with an `unroll` factor to duplicate the loop body logic and run it in parallel:

```
for (let i = 0..10) unroll 2 { f(i) }
```

This loop is equivalent to a sequential one that iterates half as many times and composes two copies of the body in parallel:

```
for (let i = 0..5) {  f(2*i + 0);  f(2*i + 1)  }
```

The doall restriction is important because it allows the compiler to run the two copies of the loop body in parallel using unordered composition. In traditional HLS tools, a loop unrolling annotation such as `#pragma HLS unroll` is always allowed—even when the loop body makes parallelization difficult or impossible. The toolchain will replicate the loop body and rely on complex analysis and resource scheduling to optimize the unrolled loop body as well as it can.

Resource conflicts in unrolled loops are errors. For example, this loop accesses an unbanked array in parallel:

```
let A: float[10];
for (let i = 0..10) unroll 2 {
  A[i] := compute(i)  // Error: Insufficient banks.
}
```

Unrolled memory accesses Dahlia uses special *index types* for loop iterators to type-check memory accesses within unrolled loops. Index types generalize integers to encode information about loop unrolling. In this example:

```
for (let i = 0..8) unroll 4 { A[i] }
```

The iterator `i` gets the type `idx{0..4}`, indicating that accessing an array at `i` will consume banks 0, 1, 2, and 3. Type-checking a memory access with `i` consumes all banks indicated by its index type.

Unrolling and ordered composition Loop unrolling has a subtle interaction with ordered composition. In a loop body containing `---`, like this:

```
let A: float[10 bank 2];
for (let i = 0..10) unroll 2 {
  let x = A[i]
  ---
  f(x, A[0]) }
```

A naive interpretation would use parallel composition to join the loop bodies at the top level:

```
for (let i = 0..5) {
  { let x0 = A[2*i]      ---  f(x0, A[0]) };
  { let x1 = A[2*i + 1]  ---  f(x1, A[0]) } }
```

However, this interpretation is too restrictive. It requires *all* time steps in each loop body to avoid conflicts with all other time steps. This example would be illegal because the access to `A[i]` in the first time step may conflict with the access to `A[0]` in the second time step. Instead, Dahlia reasons about unrolled loops *in lockstep* by parallelizing *within* each logical time step. The loop above is equivalent to:

```
for (let i = 0..5) {
  { let x0 = A[2*i]; let x1 = A[2*i + 1] }
  ---
  { f(x0, A[0]);     f(x1, A[0]) } }
```

The lockstep semantics permits this unrolling because conflicts need only be avoided between unrolled copies of the same logical time step. HLS tools must enforce a similar restriction but leave the choice to black-box heuristics.

Nested unrolling In nested loops, unrolled iterators can separately access dimensions of a multi-dimensional array. Nested loops also interact with Dahlia's read and write capabilities. In this program:

```
let A: float[8 bank 4][10 bank 5];
for (let i = 0..8) {
  for (let j = 0..10) unroll 5 {
    let x = A[i][0]
    ---
    A[i][0] := j;  // Error: Insufficient write
  } }              //        capabilities.
```

The read to array $A[i][0]$ can be proved to be safe because after desugaring, the reads turn into:

```
let x0 = A[i][0];  let x1 = A[i][0] ...
```

The access is safe because the first access acquires a read capability for indices `i` and `0`, so the subsequent copies are safe. Architecturally, the code entails a single read *fanned out* to each parallel PE. However, the write desugars to:

```
A[i][0] := j;  A[i][0] := j + 1 ...
```

which causes a write conflict in the hardware.

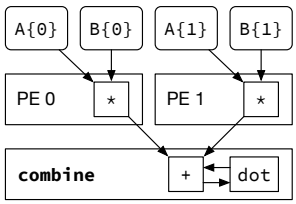
3.2.5 Combine Blocks for Reduction

In traditional HLS, loops can freely include dependent operations, as in this dot product:

```
for (let i = 0..10) unroll 2 { dot += A[i] * B[i] }
```

However, the `+=` update silently introduces a dependency between every iteration which is disallowed by Dahlia's doall `for`-loops. HLS tools heuristically analyze loops to extract and serialize dependent portions. In Dahlia, programmers explicitly distinguish the non-parallelizable reduction components of `for` loops. Each `for` can have an optional `combine` block that contains sequential code to run after each unrolled iteration group of the main loop body. For example, this loop is legal:

```
for (let i = 0..10)
unroll 2 {
  let v = A[i] * B[i];
} combine {
  dot += v;
}
```

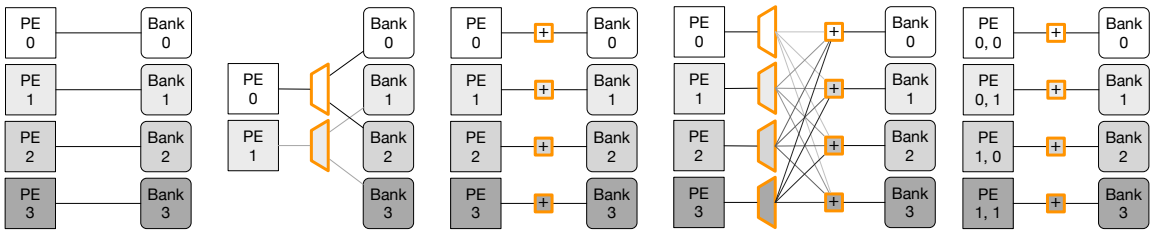


There are two copies of the loop body that run in parallel and feed into a single reduction tree for the `combine` block.

The type checker gives special treatment to variables like `v` that are defined in `for` bodies and used in `combine` blocks. In the context of the `combine` block, `v` is a *combine register*, which is a tuple containing all values produced for `v` in the unrolled loop bodies. Dahlia defines a class of functions called *reducers* that take a combine register and return a single value (similar to a functional fold). Dahlia defines `+=`, `-=`, `*=`, `/=` as built-in reducers with infix syntax.

3.2.6 Memory Views for Flexible Iteration

In order to predictably generate hardware for parallel accesses, Dahlia statically calculates banks accessed by each PE and guarantees that they are distinct. Figure 3.5a shows the kind of hardware generated by this restriction—each PE is directly connected to a bank.



(a) Banked (b) Shrink (c) Suffix (d) Shift (e) Split

Figure 3.5: Hardware schematics for each kind of memory view. Highlighted outlines indicate added hardware cost.

To enforce this hardware generation, Dahlia only allows simple indexing expressions like `A[i]` and `A[4]` and rejects arbitrary index calculations like `A[2*i]`. General indexing expressions can require complex indirection hardware to allow any PE to access any memory bank. An access like `A[i*i]`, for example, makes it difficult to deduce which bank it would read on which iteration. For simple expressions like `A[j+8]`, however, the bank stride pattern is clear. Traditional HLS tools make a best-effort attempt to deduce access patterns, but subtle changes in the code can unpredictable prevent the analysis and generate bad hardware.

Dahlia uses *memory views* to define access patterns that HLS compilers can compile efficiently and to convince the Dahlia type checker that a parallel access will be predictable. The key idea is to offer different *logical* arrangements of the same underlying physical memory. By logically re-organizing the memory, views can simply reuse Dahlia's type-checking to ensure that complex access patterns are predictable. Furthermore, this allows views to capture the hardware cost of an access pattern in the source code instead of relying on black-box analysis in HLS tools. For Dahlia's HLS C++ backend, views are compiled to direct memory accesses.

The rest of this section describes Dahlia's memory views and their cost in terms of hardware required to transform bank and index values to support the iteration

pattern.

Shrink To directly connect PEs to memory banks, Dahlia requires the unrolling factor to match the banking factor. To allow lower unrolling factors, Dahlia provides *shrink views*, which reduce the banking factors of an underlying memory by an integer factor. For example:

```
let A: float[8 bank 4];
view sh = shrink A[by 2]; // sh: float[8 bank 2]
for (let i = 0..8) unroll 2
  sh[i]; // OK: sh has 2 banks. Compiled to: A[i].
```

The example first defines a view `sh` with the underlying memory `A` and divides its banking factor by 2. Dahlia allows `sh[i]` here because each PE will access a distinct set of banks. The first PE accesses banks 0 and 2; the second accesses banks 1 and 3. The hardware cost of a shrink view, as Figure 3.5b illustrates, consists of multiplexing to select the right bank on every iteration. The access `sh[i]` compiles to `A[i]`.

Suffix A second kind of view lets programs create small slices of a larger memory. Dahlia distinguishes between suffixes that it can implement efficiently and costlier ones. An efficient *aligned suffix* view uses this syntax:

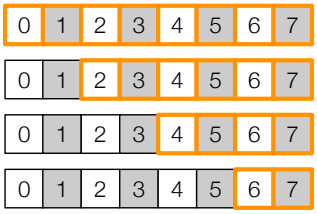
```
view v = suffix M[by k * e];
```

where view `v` starts at element $k \times e$ of the memory `M`. Critically, k must be the banking factor of `M`. This restriction allows Dahlia to prove that each logical bank in the view maps to the same physical bank while the indices are offset by the indexing expression. The hardware cost of a suffix view is the *address adapter* for each bank. A view access `v{`b`}[`i`]` is compiled to `M{`b`}[`$e+i$`]`.

For example, generating suffixes in a loop results in this pattern, where the dig-

its in each cell are the indices, the shades represent the banks, and the highlighted outline indicates the view:

```
let A: float[8 bank 2];
for (let i = 0..4) {
  view s = suffix A[by 2*i];
  s[1]; // reads A[2*i + 1]
}
```



A suffix view defined using `view v = suffix M[by k*e]` and accessed using `v[i]` is compiled to `M[k*e + i]`.

Shift Shifted suffixes are like standard suffixes but allow unrestricted offset expressions:

```
view v = shift M[by e];
```

Since `e` is unrestricted, Dahlia assumes that *both* the bank and the indices need to be adapted and that each PE accesses *every* bank. Figure 3.5d shows the hardware cost of a shift view: each PE is connected to every bank and the index expression is transformed using an address adapter. The distinction between suffix and shift views allows Dahlia to capture the cost of different accessing schemes.

Even in this worst-case scenario, Dahlia can reason about the disjointness of bank accesses. This loop is legal:

```
let A: float[12 bank 4];
for (let i = 0..3) {
  view r = shift A[by i*i];  // r: float[12 bank 4]
  for (let j = 0..4) unroll 4
    let x = r[j];  // accesses A[i*i + j]
}
```

The view `r` has a memory type, so Dahlia can guarantee that the inner access `r[j]` uses disjoint banks and is therefore safe to parallelize. An access `r[`i`]` to a view declared with `shift M[by` e`]` compiles to `M[`$e+i$`]`.

Split Some nested iteration patterns can be parallelized at two levels: globally, over an entire array, and locally, over a smaller window. This pattern arises in blocked computations, such as this dot product loop in C++:

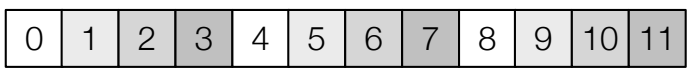
```
float A[12], B[12], sum = 0.0;
for (int i = 0; i < 6; i++)
  for (int j = 0; j < 2; j++)
    sum += A[2*i + j] * B[2*i + j];
```

Both the inner loop and the outer loop represent opportunities for parallelization. However, Dahlia cannot prove this parallelization to be safe:

```
let A, B: float[12 bank 4];
view shA, shB = shrink A[by 2], B[by 2];
for (let i = 0..6) unroll 2 {
  view vA, vB = suffix shA[by 2*i], shB[by 2*i];
  for (let j = 0..2) unroll 2 {
    let v = vA[j] + vB[j];
  } combine {
    sum += v; }}
```

While Dahlia can prove that the inner accesses into the views can be predictably parallelized, it cannot establish the disjointness of the parallel copies of the views `va` and `vb` created by the outer unrolled loop.

Split views allow for this reasoning. The key idea is to create logically *more* dimensions than the physical memory and reusing Dahlia's reasoning for multidimensional memories to prove safety for such parallel accesses. A *split view* transforms a one-dimensional memory (left) into a two-dimensional memory (right):



Using these split-view declarations:

```
view split_A = split A[by 2];
view split_B = split B[by 2];
```

$$
\begin{array}{l}
x \in \text{variables} \quad a \in \text{memories} \quad n \in \text{numbers} \\
b ::= \textbf{true} \mid \textbf{false} \qquad v ::= n \mid b \\
e ::= v \mid \textbf{bop}\ e_1\ e_2 \mid x \mid a[e] \\
c ::= e \mid \textbf{let}\ x = e \mid c_1 \multimap c_2 \mid c_1\ \textbf{;}\ c_2 \mid \textbf{if}\ x\ c_1\ c_2 \mid \\
\qquad \textbf{while}\ x\ c \mid x := e \mid a[e_1] := e_2 \mid \textbf{skip} \\
\tau ::= \textbf{bit}\langle n \rangle \mid \textbf{float} \mid \textbf{bool} \mid \textbf{mem}\ \tau[n_1]
\end{array}
$$

Figure 3.6: Abstract syntax for the DCore core language.

Each view has type `mem float[2 bank 2][6 bank 2]`. A row in the logical view represents a "window" for computation. The above example can now unroll both loops, by changing the inner access to:

```
let v = split_A[j][i] * split_B[j][i];
```

As Figure 3.5e illustrates, split views have similar cost to aligned suffix views: they require no bank indirection hardware because the bank index is always known statically. They require an address adapter to compute the address within the bank from the separate coordinates. A split view declared `view sp = split M[by` k`]` on a memory `M` with k banks translates the access `sp[`i`][`j`]` to `M{bank}[idx]` where:

$$\text{bank} = i * k + (j \bmod b) \quad \text{idx} = \left\lfloor \frac{j}{b} \right\rfloor$$

3.3 Formalism

This section provides an overview of the time-sensitive affine type system that underlies Dahlia's in a core language, DCore. We give both a large-step semantics, which is more intelligible, and a small-step semantics, which enables a soundness proof. Appendix A provides the complete operational semantics for DCore and provides the proof of soundness.

3.3.1 Syntax

Figure 3.6 lists the grammar for DCore. DCore statements c resemble a typical imperative language: there are expressions, variable declarations, conditions, and simple sequential iteration via `while`. DCore has ordered composition c_1 —— c_2 and unordered composition c_1 ; c_2. It separates memories a and variables x into separate syntactic categories. DCore programs can only declare the latter: a program runs with a fixed set of available memories.

3.3.2 Large-Step Semantics

DCore's large-step operational semantics is a *checked semantics* that enforces Dahlia's safety condition by explicitly tracking and getting stuck when it would otherwise require two conflicting accesses. Our type system (Section 3.3.3) aims to rule out these conflicts.

The semantics uses an environment σ mapping variable and memory names to values, which may be primitive values or memories, which in turn map indices to primitive values. A second context, ρ, is the set of the memories that the program has accessed. ρ starts empty and accumulates memories as the program reads and writes them.

The operational semantics consists of an expression judgment $\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v$ and a command judgment $\sigma_1, \rho_1, c \Downarrow \sigma_2, \rho_2$. We describe some relevant rules here, but elide the complete semantics and proof to Appendix A.

Memory accesses Memories in DCore are mutable stores of values. Banked memories in Dahlia can be built up using these simpler memories. The rule for a memory read expression $a[n]$ requires that a not already be present in ρ, which would indicate that the memory was previously consumed:

$$\frac{a \notin \rho_1 \qquad \sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2(a)(n) = v}{\sigma_1, \rho_1, a[e] \Downarrow \sigma_2, \rho_2 \cup \{a\}, v}$$

Composition Unordered composition accumulates the resource demands of two commands by threading ρ through:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \ ; \ c_2 \Downarrow \sigma_3, \rho_3}$$

If both commands read or write the same memory, they will conflict in ρ. Ordered composition runs each command in the same initial ρ environment and merges the resulting ρ:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \multimap c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

3.3.3 Type System

The typing judgments have the form $\Gamma_1, \Delta_1 \vdash c \dashv \Gamma_2, \Delta_2$ and $\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2$. Γ is a standard typing context for variables and Δ is the affine context for memories.

Affine memory accesses Memories are affine resources. The rules for reads and writes check the type of the index in Γ and remove the memory from Δ:

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \mathbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Delta_2 = \Delta_3 \cup \{a \mapsto \mathbf{mem}\,\tau[n_1]\}}{\Gamma, \Delta_1 \vdash a[e] : \tau \dashv \Delta_3}$$

Composition The unordered composition rule checks the first statement in the initial contexts and uses the resulting contexts to check the second statement:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \;;\; c_2 \dashv \Gamma_3, \Delta_3}$$

Ordered composition checks both commands under the same resource set, Δ_1, but threads the non-affine context through:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_1 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \multimap c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

The rule merges the resulting Δ contexts with set intersection to yield the resources not consumed by either statement.

3.3.4 Small-Step Semantics

We also define a small-step operational semantics for DCore upon which we build a proof of soundness. The semantics consists of judgments $\sigma_1, \rho_1, e \to \sigma_2, \rho_2, e'$ and $\sigma_1, \rho_1, c \to \sigma_2, \rho_2, c'$ where σ and ρ are the environment and the memory context respectively. The main challenge is sequential composition, which uses an intermediate command form $c_1 \overset{\rho}{\sim} c_2$ to thread ρ to c_1 and c_2. Appendix A provides the full rules for the small-step semantics.

3.3.5 Desugaring Surface Constructs

DCore desugars surface language features present in Dahlia.

Memory banking A banked memory declaration like this:

```
let A: float[m bank n];
```

desugars into several unbanked memories:

```
let A_0: float[m/n]; let A_1: float[m/n]; ...
```

Desugaring transforms reads and writes of banked memories to conditional statements that use the indexing expression to decide which bank to access.

Loop unrolling Desugaring of `for` loops uses the technique described in Section 3.2.4, translating from:

```
for (let i = 0 .. m) unroll k { c1 --- c2 ... }
```

into a `while` loop that duplicates the body:

```
let i = 0;
while (i < m/k) {
  { c1[i ↦ k*i+0]; c1[i ↦ k*i+1] ... }
  ---
  { c2[i ↦ k*i+0]; c2[i ↦ k*i+1] ... }
  ...
  i++;  }
```

where $c[x \mapsto e]$ denotes substitution.

Memory views For views' operational semantics, a desugaring based on the mathematical descriptions in §3.2.6 suffices. To type-check them, however, would require tracking the underlying memory for each view (transitively, to cope with

views of views) and type-level reasoning about the bank requirements of an access pattern. Formal treatment of these types would require an extension to DCore.

Multi-ported memories Reasoning about memory ports requires quantitative resource tracking, as in bounded linear logic [63]. We leave such an extension of DCore's affine type system as future work.

3.3.6 Soundness Theorem

We state a soundness theorem for DCore's type system with respect to its checked small-step operational semantics.

Theorem 3.3.1. *If* $\emptyset, \Delta^* \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\emptyset, \emptyset, c \xrightarrow{*} \sigma, \rho, c'$ *and* $\sigma, \rho, c' \not\rightarrow$, *then* $c' = \textbf{skip}$.

where Δ^* is the initial affine context of memories available to a program. The theorem states that a well-typed program never gets stuck due to memory conflicts in ρ. We prove this theorem using progress and preservation lemmas:

Lemma 3.3.2 (Progress)**.** *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *then* $\sigma, \rho, c \rightarrow \sigma', \rho', c'$ *or* $c = \textbf{skip}$.

Lemma 3.3.3 (Preservation)**.** *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *and* $\sigma, \rho, c \rightarrow \sigma', \rho', c'$, *then* $\Gamma', \Delta' \vdash c' \dashv \Gamma'_2, \Delta'_2$ *and* $\Gamma', \Delta' \sim \sigma', \rho'$.

In these lemmas, $\Gamma, \Delta \sim \sigma, \rho$ is a well-formedness judgment stating that all variables in Γ are in σ and all memories in Δ are not in ρ. Using an extension of the syntax in Figure 3.6, we prove the lemmas by induction on the small-step relation (Appendix A).

3.4 Evaluation

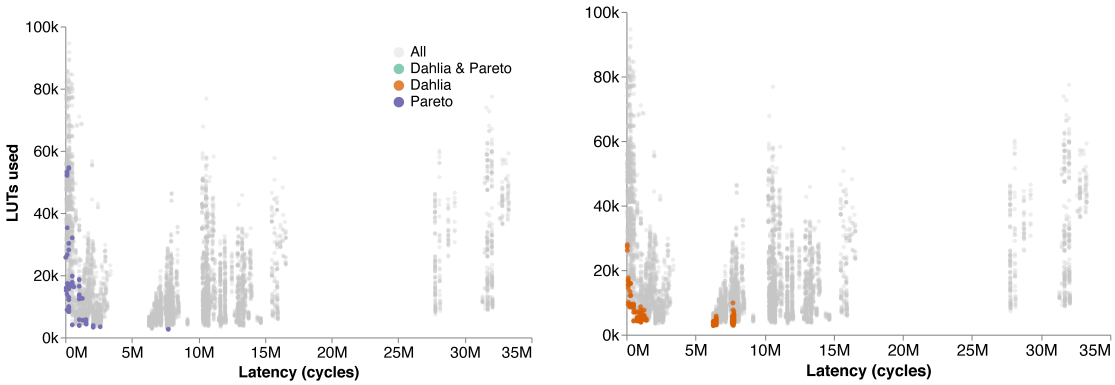
Our evaluation measures whether Dahlia's restrictions can improve predictability without sacrificing too much sheer performance. We conduct two experiments: (1) We perform an exhaustive design space exploration for one kernel to determine how well the restricted design points compare to the much larger unrestricted parameter space. (2) We port the MachSuite benchmarks [132] and, where Dahlia yields a meaningful design space, perform a parameter sweep.

3.4.1 Implementation and Experimental Setup

We implemented a Dahlia compiler in 5200 LoC of Scala. The compiler checks Dahlia programs and generates C++ code using AMD's Vitis HLS's `#pragma` directives [162]. We execute benchmarks on AWS F1 instances [7] with 8 vCPUs, 122 GB of main memory, and an AMD UltraScale+ VU9P. We use the SDAccel development environment [161] and synthesize the benchmarks with a target clock period of 250 MHz.

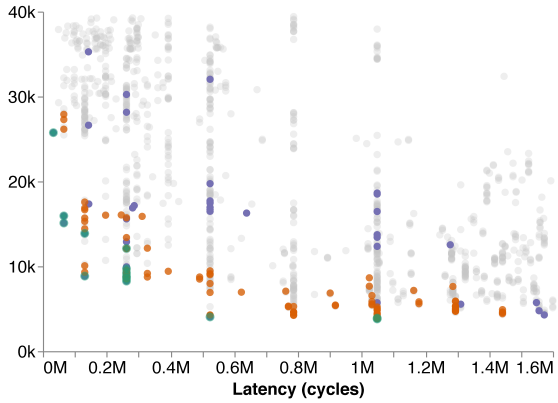
3.4.2 Case Study: Unrestricted DSE vs. Dahlia

In this section, we conduct an exhaustive design-space exploration (DSE) of a single benchmark as a case study. Without Dahlia, the HLS design space is extremely large—we study how the smaller Dahlia-restricted design space compares. We select a blocked matrix multiplication kernel (gemm-blocked from MachSuite) for its large but tractable design space. The kernel has 3 two-dimensional arrays (two operands and the output product) and 5 nested loops, of which the inner 3 are parallelizable.



(a) Pareto-optimal points.

(b) Points accepted by Dahlia.



(c) Cluster of Pareto points.

Figure 3.7: Results from exhaustive design space exploration for gemm-blocked.

We define parameters for the 6 banking factors (two dimensions for each memory) and 3 unrolling factors. We explore a design space with banking factors of 1–4 and unrolling factors of 1, 2, 4, 6, and 8. This design space consists of 32,000 distinct configurations.

We exhaustively evaluated the entire design space using Vitis HLS's estimation mode, which required a total of $2,700$ compute hours. We identify Pareto-optimal configurations according to their estimated cycle latency and number of look-up tables (LUTs), flip flops (FFs), block RAMs (BRAMs), and arithmetic units (DSPs).

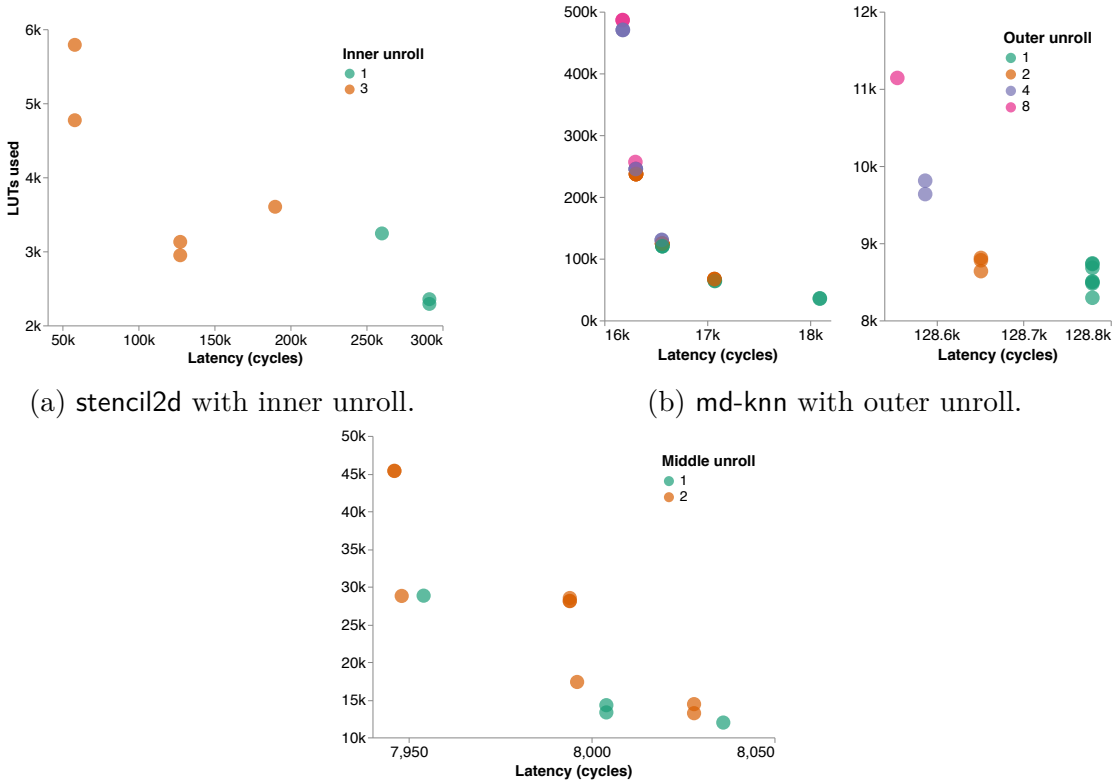
Dahlia accepts 354 configurations, or about 1.1% of the unrestricted design space. But the smaller space is only valuable if it consists of *useful* design points—

a broad range of Pareto-optimal configurations. Figures 3.7a and 3.7b show the Pareto-optimal points and the subset of points that Dahlia accepts, respectively. (Pareto optimality is determined using all objectives, but the plot shows only two: LUTs and latency.) Figure 3.7c shows a zoomed-in view of the tight cluster of Pareto points in the bottom-left of the first two graphs. Dahlia-accepted points lie primarily on the Pareto frontier and allow area-latency trade-offs. The optimal points that Dahlia rejects expend a large number of LUTs to reduce BRAM consumption which, while Pareto optimal, don't seem to be of practical use.

3.4.3 Dahlia-Directed DSE & Programmability

We port benchmarks from an HLS benchmark suite, MachSuite [132], to study Dahlia's flexibility. Of the 19 MachSuite benchmarks, one (backprop) contains a correctness bug and two fail to synthesize correctly in Vivado, indicating a bug in the tools. We successfully ported all 16 of the remaining benchmarks without substantial restructuring.

From these, we select 3 benchmarks that exhibit the kind of fine-grained, loop-level parallelism that Dahlia targets as case studies: sencil2d, md-knn, and md-grid. As the previous section illustrates, an unrestricted DSE is intractable for even modestly sized benchmarks, so we instead measure the breadth and performance of the much smaller space of configurations that Dahlia accepts. For each benchmark, we find all optimization parameters available in the Dahlia port and define a search space. The type checker rejects some design points, and we measure the remaining space. We use Vitis HLS's estimation mode to measure the resource counts and estimated latency for each accepted point. Figure 3.8 depicts the Pareto-optimal points in each space. In each plot, we also highlight the effect a single parameter



(a) stencil2d with inner unroll.

(b) md-knn with outer unroll.

(c) md-grid with middle unroll.

Figure 3.8: The design spaces for three MachSuite benchmarks. Each uses a color to highlight one design parameter.

has on the results.

The rest of this section reports quantitatively on each benchmark's design space and reports qualitatively on the programming experience during the port from C to Dahlia.

stencil2d MachSuite's stencil2d is a filter operation with four nested loops. The outer loops scan over the input matrix and the inner loops apply a 3×3 filter. Our Dahlia port unrolls the inner two loops and banks both input memories. We use unrolling factors from 1 to 3 and bank each dimension of the input array by factors 1 to 6. The resulting design space has 2,916 points. Dahlia accepts 18 of

these points (0.6%), of which 8 are Pareto-optimal within the set.

Figure 3.8a shows the Pareto frontier among the Dahlia-accepted points. The figure uses color to show the unrolling factor for the innermost loop. This unrolling factor has a large effect on the design's performance, while banking factors and the other loop explain the rest of the variation.

The original C code uses single-dimensional arrays and uses index arithmetic to treat them as matrices:

```
for (r=0; r<row_size-2; r++)
  for (c=0; c<col_size-2; c++)
    for (k1=0; k1<3; k1++)
      for (k2=0; k2<3; k2++)
        mul = filter[k1*3 + k2] *
              orig[(r+k1)*col_size + c+k2];
```

In the Dahlia port, we must use proper two-dimensional arrays because the compiler rejects arbitrary indexing expressions. Using views, programmers can decouple the storage format from the iteration pattern. To express the accesses to the input matrix `orig`, we create a shifted suffix view (Section 3.2.6) for the current window:

```
for (let row = 0..126) {
  for (let col = 0..62) {
    view window = shift orig[by row][by col];
    for (let k1 = 0..3) unroll 3 {
      for (let k2 = 0..3) unroll 3 {
        let mul = filter[k1][k2] * window[k1][k2];
```

The view makes the code's logic more obvious while allowing the Dahlia type checker to allow unrolling on the inner two loops. It also clarifies why parallelizing the outer loops would be undesirable: the parallel views would require overlapping regions of the input array, introducing a bank conflict.

md-knn The md-knn benchmark implements an n-body molecular dynamics simulation with a k-nearest neighbors kernel. The MachSuite implementation uses data-dependent loads in its main loop, which naïvely seems to prevent parallelization. In our Dahlia port, however, we hoist this serial section into a separate loop that runs before the main, parallelizable computation. Dahlia's type system helped guide the programmer toward a version of the benchmark where the benefits from parallelization are clear.

For each of the program's four memories, we used banking factors from 1 to 4. We unrolled each of the two nested loops with factors from 1 to 8. The full space has 16,384 points, of which Dahlia accepts 525 (3%). 37 of the Dahlia-accepted points are Pareto-optimal.

Figure 3.8b shows two Pareto frontiers that Dahlia accepts at different scales. The color shows the unrolling factor of the outer loop. The frontier on the right uses an order of magnitude fewer resources but is an order of magnitude slower. In this kernel, the dominant effect is the memory banking (not shown in the figure), which determines which frontier the designs fall into. The outer unrolling factor (shown in color) affects the two regimes differently: on the right, it allows area–latency trade-offs within the frontier; on the left, it acts as a second-order effect that expends LUTs to achieve a small increase in performance.

md-grid Another algorithm for the same molecular dynamics problem, md-grid, uses a different strategy based on a 3D grid implemented with several 4-dimensional arrays. It calculates forces between neighboring grid cells. Of its 6 nested loops, the outer three are parallelizable. We use banking factors of 1 to 4 for each dimension of each array, and we try unrolling factors from 1 to 8 for both loops. The full space

has 21,952 points, of which Dahlia accepts 81 (0.4%). 13 of the Dahlia-accepted points are Pareto-optimal.

Figure 3.8c again shows the Pareto-optimal design points. The *innermost* loop unrolling factor (not shown in the figure) determines which of three coarse regimes the design falls into. The color shows the *second* loop unrolling factor, which determines a second-order area–latency trade-off within each regime. Unrolling enables latency-area trade-offs in both the cases.

3.5 Discussion

High-level synthesis (HLS) tools offer a new programming model for hardware design but fail to provide *predictable* performance trade-offs. By explicitly modeling *timing constraints* of memories, Dahlia demonstrates how HLS tools can provide predictable resource-performance trade-offs. When users are likely to make unpredictable trade-offs, Dahlia's type system provides an explanation for why the trade-off will be unpredictable. This is a different approach from the large number of design space exploration (DSE) tools [85, 125, 143] which attempt to completely automate the process of optimizing designs; when such tools produce an efficient or inefficient design point, it is not possible for the user to debug the performance characteristics of the program.

Dahlia's modeling of logical time has inspired various other systems at different levels of the stack. Spade [142], a hardware description language, implements an affine type system inspired by Dahlia to track the reads and writes to wires every clock cycle. PDL [164] specifies separation between pipeline stages using a logical timestep operator inspired by Dahlia's ordered composition.

CHAPTER 4

A COMPILER INFRASTRUCTURE FOR AUTOMATIC HARDWARE GENERATION

Domain-specific language (DSL) for hardware generation [52, 74, 89, 128] offer an alternative programming model to high-level synthesis (HLS) tools which aim to repurpose legacy programming languages for hardware synthesis. By eschewing generality, such languages can provide high-level abstractions without compromising on efficiency. However, building a high-performance domain-specific language is a Herculean task: designers must design, debug, and optimize a compiler that can effectively generate designs competitive with handwritten register transfer level (RTL) designs. *Compiler infrastructures* like LLVM [96] have been successful in the software community in decoupling the design of languages from optimization tools. By building a modular compiler infrastructure, new frontend languages can target an existing, higher-level representation, and delegate the task of optimization to it.

Chapters 4–6 explore the design of Calyx, a new compiler infrastructure for building DSLs for hardware design. The key ingredient in the design of compiler infrastructures is an intermediate language (IL) that the compiler uses to analyze and transform programs. In order to bridge the semantics of high-level DSLs to circuit implementations, the Calyx IL intermixes software-like control operators with hardware-like circuit operators. Calyx also explicitly reasons about cycle-level timing (§5) in order to optimize generated designs and efficiently with existing RTL.

4.1 Design Considerations

An intermediate language (IL) for automatic hardware generation needs to bridge the gap between control-dominated computational languages, and structure-dominated circuit descriptions. Additionally, the IL must be designed so that compilers can take advantage of the extra control information present when compiling high-level programs to circuits; such information is hard to encode and extract from circuit descriptions and is therefore a valuable source for implementing aggressive analyses and optimizations that are not possible for circuit descriptions. Finally, the IL must support efficient compilation to circuit descriptions; it should not be the case that the IL's own abstractions make it difficult to generate optimized circuits.

We describe four design considerations in the implementation of ILs for hardware generation:

Reasoning about time When bridging the gap between computational languages and circuit descriptions, the IL needs to reconcile two different notions of time. Computational programs express time through program order: sequenced statements must appear as if they execute in the order they are written. On the other hand, synchronous circuits use *clock signals* to explicitly track the passage of time.[1] The clock signal is the fastest toggling signal in a circuit and must be used to track the passage of time and sequence events.

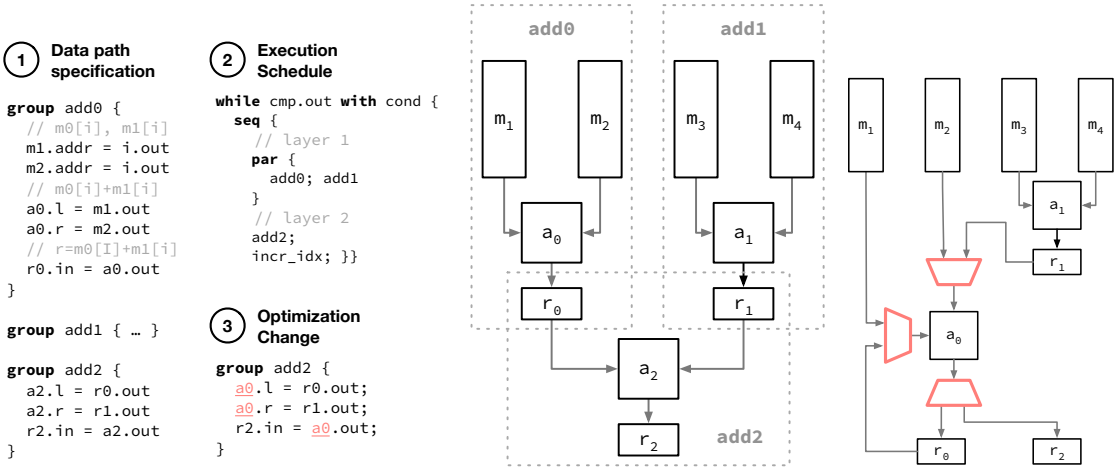
We term ILs that explicitly model the timing behavior of components latency-sensitive ILs while those that abstract away timing behavior latency-insensitive ILs.

[1] Circuits will often use more than one clock signal to run different parts of the circuit at different speeds. For example, the memory circuit might use a clock that's twice as fast as the compute circuit to be able to transfer more data per cycle of the compute circuit.

The fragment of Calyx presented in this chapter is latency-insensitive. Chapter 5 discusses the trade-off space between latency-sensitive and insensitive ILs in more detail and describes the latency-sensitive fragment of Calyx.

Representation of control flow The IL must provide some mechanism of representing the control flow of the high-level program. Languages such as AHIR [136] and SynASM [141] use finite state machines to model a program's execution schedule in a cycle-accurate manner while Calyx, HECTOR [163], and μIR [140] use high-level constructs such as loops and conditionals to express control flow.

Pipelining Pipelining is a key optimization and crucial for generating high-performance hardware. Because of this, ILs may choose to include pipelining as a primitive operation or make it the default semantics of the language. For example, HIR embeds pipelining semantics into the IL and requires programs to ensure no structural hazards are created. One of the sub-languages in HECTOR directly supports expressing pipelines. Similarly, the `pipeline` dialect in LLVM CIRCT [153] is used to express pipelined programs. On the other hand, ILs like Dynamatic [83] *implicitly* pipeline programs: users get a sequential encoding of their program and the compiler automatically pipelines the design. Adding pipelining semantics creates an expressivity–efficiency trade-off: directly expressing pipelines enables efficient encodings but forces high-level programs to generate and interface with non-pipelined designs. For example, HIR only supports fully-pipelined modules and therefore makes it impossible to interface with black-box modules that are partially pipelined. Calyx does not have first-class support for pipelining: while this allows additional flexibility for frontend compilers that want to express different microarchitectural features, it limits Calyx's ability to perform pipeline-aware



(a) Calyx program. Groups `incr_idx` and `cond` elided.

(b) Initial architecture (groups marked).

(c) Optimized architecture.

Figure 4.1: Calyx describes the reduction tree using its split representation. The execution schedule makes the control flow explicit while encapsulate connections between hardware modules. Done signals (§4.3.3) elided from group definitions.

optimizations.

4.2 Overview by Example

This section introduces Calyx by using it to implement a parallel *reduction tree*. A reduction tree applies an operator to many inputs to produce a single output. Figure 4.1b shows a small summation tree on four inputs. The operators within a tree level run in parallel to produce the inputs to the next level. Unlike hardware description languages (HDLs) or high-level synthesis (HLS), Calyx programs are meant to be generated by compiler frontends. We show that with Calyx's control language, compilers can encode the semantics of high-level languages while producing programs amenable to hardware optimization.

4.2.1 Reduction Tree in Calyx

Figure 4.1a shows a Calyx program fragment that implements a parallel reduction tree that computes $(m_0 + m_1) + (m_2 + m_3)$. The program uses *groups* to specify the data path ①. Groups encapsulate hardware connections that implement an action. For example, the group `add0` uses the hardware adder `a0` to compute the sum of the first two inputs and save the result in a register `r0`. The assignments used inside groups correspond to *non-blocking assignments* in RTL languages—updates to the left-hand side of an assignment are immediately propagated to the right-hand side. In this way, each group encapsulates a data flow graph.

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

Figure 4.1b shows the high-level architecture generated from the Calyx program and marks the connections that correspond to the groups. The figure elides the control circuitry generated to implement the schedule.

4.2.2 Optimizing Accelerator Designs

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

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

Consider an optimization that reuses existing circuitry to perform temporally disjoint computations. For example, our reduction tree uses adders `a0` and `a2` in two different stages and never overlaps their execution. Therefore, it would be safe to transform the program to share a single adder for both the stages. Implementing this optimization in RTL, however, is difficult because the structural implementation of a state machine obscures the program's control flow. To determine that the two adders run at different times, an analysis would need to reverse-engineer the execution schedule from the state machine implementation. Furthermore, transforming an RTL program would require pervasive changes. Figure 4.1c shows the optimized architecture. The transformation requires carefully rewiring the input and output signals for `a0` through multiplexers.

In contrast, a Calyx program makes the control flow explicit and enables

straightforward transformation. Given the execution schedule of our Calyx program, it is clear that the groups `add0` and `add2` do not execute simultaneously since they are scheduled using the `seq` operator. Figure 4.1a ③ shows the only change required to implement this optimization. The Calyx program simply renames the uses of `a2` in group `add2` with `a0` and the compiler correctly generates the additional multiplexers and control signals to share the adder.

4.2.3 Structure and Control

Calyx is neither a software IL nor a hardware IL. Software ILs, such as LLVM [96], focus on providing a uniform representation of the control flow and data flow of a program. They do not explicitly represent structural facts, such as the mapping of logical adds onto physical adders. On the other hand, hardware ILs focus on a purely structural representation with explicit use of gates, wires, and clocks while conflating data flow with control signals. By marrying structure and control, Calyx provides access to both structural and control flow facts to enable a new class of optimizations that cannot be captured by either style of ILs.

4.3 The Calyx Intermediate Language

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

and optimize the IL.

4.3.1 Components

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

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

The body includes hardware-like structural listings of *cells* and *wires* (§4.3.2) and software-like *control* code (§4.3.3). The input and output ports form the interface to the component and define their size in bits. For example, a component defining a 32-bit integer adder uses these ports:

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

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

4.3.2 Cells and Wires

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

The cells section instantiates components:

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

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

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

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

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

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

The guarded assignments to the `left` port of the `add` component use the value of `cmp.out` to determine the assignment to activate. Guards are built with ports and a simple language of boolean connectives.

Like its RTL counterparts, Calyx requires that each port have a *unique driver*—activating multiple assignments in the same cycle results in undefined behavior. This requirement also differentiates Calyx's guarded assignments from Bluespec's atomic rules [120]. While Bluespec resolves conflicting assignments by generating scheduling logic to dynamically abort them, Calyx does not. Being an intermediate language, Calyx trades-off the convenient programming abstraction for predictable

compilation.

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

4.3.3 Groups and Control

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

```
group assign_one { x_reg.in = 1; ... } // x_reg = std_reg(32)
group assign_two { x_reg.in = 2; ... }
```

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

The *control program* determines when groups run:

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

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

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

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

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

Interface signals, such as a group's done signal, are used by Calyx to define a *calling convention* (Section 4.5.1). A control program passes control flow to a group by setting a group's `go` to 1. A group returns control back to the control program by setting its own `done` condition to 1. Similarly, components use `go` and `done` interface signals to define a consistent calling convention. Calyx's interface is *latency-insensitive*; it does not reason about the number of cycles needed to execute a computation. Chapter 5 describes the Calyx extension used to reason about latency-sensitive computations.

Groups serve a similar purpose to basic blocks [6] in a software intermediate language. They delineate data-flow from control-flow and allow the compiler to implement scalable analyses and optimizations. However, unlike basic blocks, groups in Calyx can encode arbitrary circuits which means that Calyx's ability to optimize the contents of a group itself is limited.

Combinational groups Calyx additionally supports a restricted, combinational version of groups:

```
comb group compute_cond {
  gt.left = reg.out;  // gt: std_gt(32)
  gt.right = 32'd10;
}
```

Combinational groups only execute assignments for one cycle and are useful for performing simple computations like comparisons.

4.3.4 Control Statements

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

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

invoke Invocations pass the control flow to the given component and connects the provided ports values to the input of the component.

```
component mac(a: 32, b: 32, c: 48) -> (out: 48) { ... }
component main() {
  ...
  control {
    invoke foo(.a = r0.out, .b=r1.out, .c=acc.out)(acc.in = .out)
  }
}
```

par List of control statements that execute once in parallel.

```
par {
  group_a;                    // thread 1
  seq { group_b; group_c; };  // thread 2
```

```
  group_d;                    // thread 3
}
```

The control operator provides *no guarantee* on the execution order of the threads. For example, it is incorrect to assume that `group_a` and `group_d` will start execution in the same clock cycle. This restriction allows the compiler to freely reschedule and combine parallel threads to optimize the program.

seq List of control statements executed in order.

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

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

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

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

```
while port_name with comb_group {
  body_stmt
}
```

4.3.5 Attributes

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

```
component main(clk c: 1)  ...  // ports use the <attr>(<val>) syntax
group foo<"latency"=1> { ... }
```

4.3.6 Synopsis

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

The design principle behind Calyx is thus: in general, frontends generate small groups to perform simple actions, such as incrementing a register or comparing values, and use the control flow program to schedule them. However, when a frontend wants to instantiate a domain-specific construct efficiently, it can utilize the full power of Calyx's structural language and precisely specify the RTL description of the implementation which can then be encapsulated using groups. Calyx's optimization passes (§4.6) can then utilize the control flow information to optimize the design.

4.4 Targeting Calyx

Calyx intermixes software and hardware abstractions. In this section, we demonstrate how these abstractions can be used to express well-known architectures such

as systolic arrays [92] as well as be used in the design of an HLS tool by compiling Dahlia (§3) to Calyx. Chapter 6 extends and evaluates these implementations in more depth.

4.4.1 Systolic Array Generator

Systolic arrays [92] are a class of architecture exploit data reuse opportunities in arithmetically intense computations such as dense linear algebra computations. They power the recent wave of state-of-the-art linear algebra accelerators for machine learning [58, 84]. Figure 4.2 shows an example systolic array. The data moves left-to-right and top-to-bottom through chains of registers. Processing element (PE) attached to the registers perform some computation using the values within the registers. Once the computation is completed, all the data moves to the next register.[2] Systolic arrays can maintain a high throughput because data is reused between PEs and, in steady state, each PE is simultaneously operating on different parts of the computation.

We overview the design of a systolic array in Calyx which is abstract in the processing element being used. We demonstrate how groups allow simple encoding of data movement and computation stages and how the control program enables us to schedule the computation.

Processing elements The design of our systolic array is abstract with respect to the processing element. We only require the processing element to conform to the following signature:

[2]This pattern of interleaved compute and data movement induces a comparison to heartbeats which inspired the name "systolic arrays".

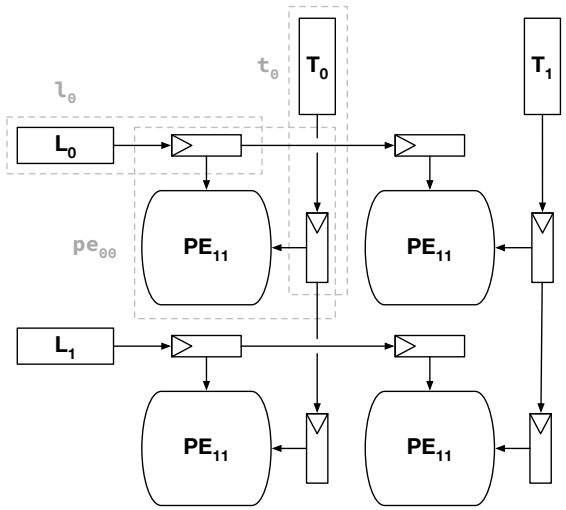


Figure 4.2: Architecture for a 2×2 by 2×2 matrix-multiply systolic array. Highlighted boxes show some of the groups used by the control.

```
component pe(go: 1, left: 32, right: 32) -> (done: 1, out: 32) { ... }
```

The PEs must also track their own internal state. For example, a systolic array for output-stationary matrix multiplication will implement the PE using a multiply-accumulate unit that stores the results in an internal register.

Since Calyx does not reason about latency of the computation, the component presents a `go-done` interface to interact with the rest of the design. Our actual implementation (§6.2) uses a mix of latency-sensitive (§5) and latency-insensitve abstractions.

Architecture Figure 4.2 shows the desired architecture for a 2×2 systolic array. When designing the array in Calyx, we need to instantiate the subcomponents using the *cells* section, connect them together using *groups*, and coordinate their execution using a control program.

Cells We instantiate the name of the registers and PEs using their index on a two-dimensional grid. For example, $\texttt{PE}_{00}$ is the top-left PE while register $\texttt{l}_{00}$ is the top-left register. The memories connected to the systolic array are named $\texttt{L}_i$ and $\texttt{T}_i$ respectively.

```
component sa(...) {
  cells {
    L_0 = std_mem_d1(32, 8, 3); // 8-element, 32-bit memory indexed
        using 3-bit number
    PE_00 = pe(); // Processing element for this array
    l_00 = std_reg(32); // 32-bit register
}}
```

Building blocks Calyx's group abstraction allows us to define easily define various actions that the systolic array must perform. For example, the following group moves the value from register $\texttt{l}_{00}$ to $\texttt{l}_{01}$:

```
group move_l00 {
  l_01.in = l_00.out;
  l_01.write_en = 1;
  move_l00[done] = l_01.done;
}
```

Similarly, the following group reads the final value stored in $\texttt{PE}_0 0$ at the end of the computation into an output register:

```
group read_pe00 {
  out_00.in = PE_00.out;
  out_00.write_en = 1;
  read_pe00[done] = out_00.done;
}
```

While we can manually write down a group to execute a PE's control program, `invoke` statements make it much simpler:

```
invoke PE_00(.left = l_00.out, .top = t_00)(); // .out left unconnected
```

With these elements, we have the basic building blocks for a systolic array.

Scheduling computation The next step is to schedule the computation of a systolic array. We focus on the steady state case and elide details about the pipeline fill and pipeline drain stages. In steady state, all processing elements need to perform their computation and, once they complete, the registers can move their data forward. This process needs to continue while we have a stream of valid data arriving.

```
// pipeline fill
while data_is_valid {
  seq {
    par { invoke PE_00(..); invoke PE_01(..); ...}
    seq { // data movement
      par { move_l0n; move_l1n; ..; move_tn0; move_tn1 .. }
      ...
      par { move_l00; move_l10; ..; move_t00; move_t01; .. }
}}}
// pipeline drain and data read
```

The `data_is_valid` port is left abstract but should assert 1 when there is a valid data element in every memory on the left and the top. Because the processing elements use a latency-insensitive interface, they can each take a variable number of cycles; the first `par` block will wait for all invocations to complete before finishing up. The subsequent `seq` block schedules the data movement groups by first moving data from the second to the rightmost layer to the rightmost layer, and second to bottom-most layer to the bottom-most layer. It repeats this process for each layer of registers.

Discussion Calyx's abstractions simplify the expression of complex architectural designs such as systolic arrays. The latency-insensitive interface allows us to flexibly design our systolic array and easily replace out the processing element implementations. However, the same latency-insensitive interface forces us to carefully orchestrate the data movement between registers. If Calyx allowed us to model the timing behavior of data movement groups, we could exploit the fact that writing

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

(a) Dahlia program

```
group init { x.in = 0; init_x[done] = x.done;}
group one  { x.in = 1; one[done] = x.done;}
group two  { x.in = 2; two[done] = x.done;}
group cond { gt.l = x.out; gt.r = 10; cond[done] = 1;}
...
seq {
  init_x;
  if gt.out with cond { one } else { two }
}
```

(b) Generated Calyx

Figure 4.3: Process of compiling Dahlia programs. Each statement becomes a group and control flow is encoded using control operators.

to a register takes exactly one cycle and enable all data movement groups at the same time allowing us to cut down an n-cycle computation to a single cycle. This tension between abstracting away timing details and exploiting them for efficiency is an ever-present tension in hardware design. Chapter 5 describes an extension to Calyx that allows for precise modeling of timing when needed while composing cleanly with the Calyx's existing latency-insensitive abstractions.

4.4.2 Dahlia

We overview the design of an HLS tool that compiles Dahlia programs (§3) to Calyx. Unlike register transfer level (RTL) languages which require programs to be encoded in a cycle-sensitive manner and utilize explicit control signals to manage resources, Calyx's abstractions are well-suited for the design of a standard, syntax-driven compiler which recursively compiles each statement without needing to reason about other statements. Furthermore, Calyx's explicit parallel and sequential composition operators allow for direct mapping of Dahlia's *ordered* and *unordered* composition operators (§3.2).

Lowered Dahlia Before compiling Dahlia, we desugar high-level language constructs:

- Unroll loops and partition memories using user-specified information (§§ 3.2.3 and 3.2.4). The compilation process follows the desugaring rules from Dahlia to DCore (§3.3). This ensures that we correctly insert Dahlia's ordered and unordered composition operators.
- Transform `for`-loops into `while` loops.
- Compile away views (§3.2.6).

This results in an imperative language with a few constructs: variables, unpartitioned memories, `while` loops, conditionals, and Dahlia's two novel composition operators: *unordered* composition (`;`) and *ordered* composition (`---`).

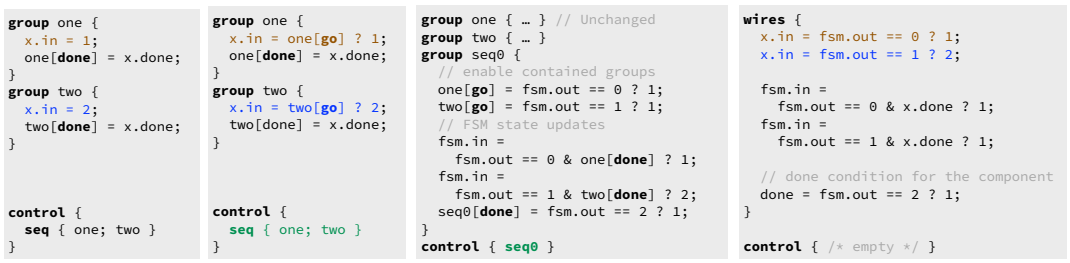
Generating Calyx Figure 4.3 shows how a Dahlia program is compiled to Calyx. Each program statement is mapped to a group and the execution of groups is controlled using the control operators. For example, the statement `x := 1` is encoded using the group `one`.

The compiler walks over the program abstract syntax tree (AST) and allocates new registers for each variable in a scope and instantiates a new circuit for each binary operator it sees. Control constructs have a direct mapping to operators in Calyx: ordered composition maps to `seq`, unordered composition to `par`, conditionals to `if`, and loops to `while`. A complete overview of the compiler is presented in Section 6.3.

Discussion Calyx's software-like abstractions allow easy encoding of high-level programs. Instead of having to directly transform *logical time steps* in Calyx to discrete clock-cycles in an RTL language, we can utilize Calyx's control operators to more naturally encode the programs. While easy to implement, the described compiler has several limitations: (1) it allocates a large number of circuits to implement a particular control-data flow graph, (2) it does not take advantage of latency-information of specific operations, and (3) it does not generated pipelined designs. Calyx's optimizations (§4.6) addresses the first problem and Calyx's static extension (§5) addresses the second problem. The last problem is more fundamental: Calyx requires frontends to explicitly generate pipelines. While this dissertation does not describe such a compiler, Calyx has been used to design HLS tools that generate pipelined designs.

4.4.3 Summary

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



(a) Original (b) GoInsertion (c) CompileControl (d) RemoveGroups

Figure 4.4: Calyx realizes the execution schedule by encoding it with structural components. After the CompileControl pass (c), the `fsm` register encodes the current state for the `seq` statement.

4.5 Compiling Calyx to Hardware

The Calyx compiler optimizes (§4.6) and lowers Calyx programs into synthesizable RTL. Compilation passes use *interface signals*, which define a calling convention, to realize a component's execution schedule. The result is a Calyx program with a flat list of guarded assignments and no control statements or groups. The compiler can then directly translate this flattened form into RTL. The primary compilation passes are:

- GoInsertion: Guards all assignments in a group with the group's `go` interface signal.
- CompileControl: Generates latency-insensitive finite state machines to structurally realize control operators.
- RemoveGroups: Inlines uses of interface signals and eliminates all groups.
- Lower: Translates control-free Calyx to RTL.

Figure 4.4 illustrates the main steps. This section describes the complete compilation process.

4.5.1 Calling Convention

To realize a Calyx program's execution schedule, the compiler needs a mechanism to pass control flow in purely structural programs. We use a pair of *interface signals* to define this interface: when a group sets another group's `go` signal high, control is passed to that group and it can enable assignments within it; when a group sets its own `done` signal high, it passes control back. This interface resembles traditional latency-insensitive hardware design [27] but, unlike a pipelined interface, does not support backpressure signals.

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

4.5.2 Compilation Workflow

We describe the compilation pipeline by compiling the example Calyx program in Figure 4.4a.

Inserting `go` interface signals Calyx's semantics requires that assignments within a group are only enabled when the group executes. To enforce this requirement, the GoInsertion pass inserts the group's go signal into the guards of the contained assignments. Figure 4.4b shows the resulting program: `one[go]` guards assignments in group `one` while `two[go]` guards assignments in group `two`. When all groups are eventually removed, these guards will ensure that the correct set of assignments are active at a given time.

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

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

Figure 4.4c shows these transformations. The pass defines a new group `seq0` to encapsulate the structure required to realize the `seq` statement as well as a new register `fsm` to track the current state. Next, the pass enables the groups contained in the `seq` by writing to their go interface signals and updates the FSM state when the groups set their done signal high. The done condition for `seq0` is when the FSM reaches its final state. Finally, the pass replaces the `seq` control statement with the group `seq0`.

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

1. Add new `go` and `done` *ports* to each component definition and wire them up to the single group enable in the control program.

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

Figure 4.4d shows the resulting program that contains no groups, interface signals, or control statements.

Code generation Each component now contains a flat list of guarded assignments. The Lower pass generates SystemVerilog programs by mapping each component to a module, generating wires for all the ports, and threading a clock signal through the design.

4.5.3 Compiling Control Statements

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

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

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

```
control {
  par0;
}
```

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

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

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

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

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

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

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

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

(b) Schedule with resource sharing opportunities.

Figure 4.5: Resource sharing example. Since `incr_r0` and `incr_r1` do not run in parallel, they can share their adders.

4.6 Optimizing Calyx Programs

Unlike RTL languages, where control-flow information is encoded into ad-hoc registers, Calyx's representation explicitly encodes control-flow information along with circuit information. Because of this, Calyx can implement analyses and optimization that transform details of the circuit using control-flow information. Furthermore, since Calyx is not tied to the RTL abstraction, optimizations can change both the precise timing and the structure of the final circuit.[3] We describe two passes in the Calyx compiler that optimize the resource usage of a design. Section 5.4 describes additional optimizations that work with Calyx's latency-sensitive abstractions and Chapter 6 evaluates the optimizations.

4.6.1 Resource Sharing

Resource sharing is an optimization that reuses existing circuits to perform temporally disjoint computations. For example, if an accelerator needs to perform two add operations that are never executed in parallel, it can map them to the

[3]The RTL abstraction considers registers sacrosanct and, with very few exceptions, does not change the timing behavior of the circuit or how it uses the registers.

same physical adder. Calyx is uniquely suited to implement such optimizations which require both control flow facts (if two computations run in parallel) and structural facts (which physical adder performs an add).

Calyx implements a group-level resource sharing optimization: if two groups are guaranteed to never execute in parallel, they can share components. This pass does not attempt to share stateful components because state is visible across groups. Frontends use the `"share"` attribute (§4.3.5) to denote that a component is safe to share.

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

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

Building a conflict graph A conflict graph summarizes potential conflicts—nodes denote groups and edges denote that the groups *may* run in parallel. The pass traverses the control program and adds edges between all children of a `par` block. For example, in Figure 4.5b, the groups `let_r0` and `let_r1` conflict with each other while `incr_r0` and `incr_r1` do not. If the children of the `par` block are themselves control programs, the pass adds edges between the groups contained within each child.

Greedy coloring The pass performs a greedy coloring over the conflict graph to allocate shareable components to each group. If two groups have an edge between them, they cannot have the same components. The result of this step is a mapping from the names of old components to new components. For example, in Figure 4.5a,

`incr_r1` gets the mapping: $a1 \mapsto a0$.

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

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

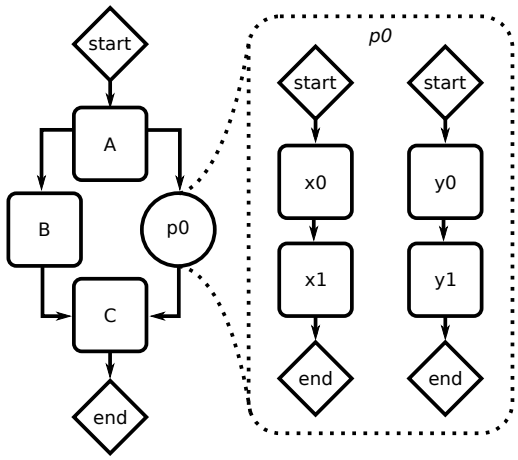
4.6.2 Component Sharing via Live-Range Analysis

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

We overview the component sharing pass using registers as the exemplar component. Our compiler implements a generalized version that supports arbitrary user-level and primitive components. Our live-range analysis has to contend with two problems: (1) inferring which groups read and write to registers, and (2) coping with the `par` blocks in the control program.

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

(a) Calyx program.



(b) A visual representation of a pCFG.

Figure 4.6: A Calyx program along with the corresponding parallel control flow graph (pCFG).

Parallel control flow graphs We handle `par` blocks using parallel control flow graphs (pCFGs) based on the work of Srinivasan and Wolfe [144]. Most control operators in Calyx map directly to a traditional CFG. However, `par` statements need special handling since, unlike an `if` statement which executes one of its two branches, a `par` statement executes *all* its children. While writes to a register in a conditional branch *may* be visible after the `if` statement, writes within children of `par` blocks are *always* visible after the `par` block.

Parallel CFGs introduce a new kind of node—called a *p-node*—to handle `par` blocks ($p0$ in Figure 4.6b). A p-node represents an entire `par` block and recursively contains a set of pCFGs representing its children. In Figure 4.6b the p-node has two children.

Calculating read and write sets Calyx implements a conservative analysis pass to determine the registers that groups and p-nodes read from and write to. Both groups and p-nodes can, in general, contain complex logic, so the pass must conservatively over-approximate these sets. The read set is the set of registers a

group or p-node *may* read from and the write set is the set of registers they *must* write to. The data-flow analysis uses this information to determine the range each register is alive.

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

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

4.7 Discussion

Calyx defines a new intermediate language that bridges the gap between computational programs and circuit descriptions. We overview Calyx's core design principles, which enable modular reasoning of circuits while allowing for precise specification (§4.3), how frontends can generate a variety of domain-specific architectures (§4.4), and how Calyx can exploit the representation to optimize programs (§4.6).

Chapter 5 discusses an extension to Calyx that allows for precise reasoning about timing without compromising on modularity. Chapter 6 implements and evaluates frontends that use Calyx.

CHAPTER 5

COMPOSITIONAL, TIME-SENSITIVE REASONING FOR HARDWARE GENERATION

Chapter 4 describes the initial design of Calyx, an intermediate language for hardware generation. Calyx uses latency-insensitive interfaces to compose programs which allows for scalable and modular program optimizations. However, this compositionality comes at a cost: Calyx cannot correctly express optimizations that fundamentally rely on timing information. In this chapter, we extend Calyx to reason about timing behaviors without losing Calyx's compositional nature. The key insight is that latency-sensitive execution schedules of a program are a *refinement* of latency-insensitive execution schedules. This allows us to embed a latency-sensitive operators into Calyx without losing the compositionality of Calyx's latency-insensitive operators.

5.1 Static Abstractions for Calyx

We refer to Calyx's latency-sensitive extensions as *static* operators since they allow us to statically reason about timing behavior. Latency-insensitive operators are called *dynamic* operators since they do not have statically known timing behaviors. Figure 5.1 shows these extensions in context of a simple component that performs the computation $(a + b) \times c \div d$. The extensions are highlighted in green. Sections 5.1.1 and 5.1.2 overview the extensions to the structural and control sections respectively.

```
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;
      // no done condition
    }
    static<3> group do_mult {
      mult.left = %[0:3] ? add.out;
      mult.right = c; // implicit %[0:3] guard
      // no done condition
    }
    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;
}}}
```

Figure 5.1: A Calyx component that computes $(a+b)\times c\div d$. The static extensions are shown in green.

5.1.1 Static Structural Abstractions

We add new, time-sensitive structural abstractions: static components and static groups.

Static components Where Calyx's existing 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 compiler.

Static groups and relative timing guards Static groups in Calyx 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 5.1, 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.

Calyx's relative timing guards resemble cycle-level schedules in some purely static languages [103, 117]. However, they count relative to the start of the *group* rather than the entire *component*. This distinction is crucial since it lets Calyx use static groups in both static and dynamic contexts.

5.1.2 Static Control Operators

We provide 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 programs may contain both static and dynamic children, static programs must only have static children. We write $|c|$ for the latency of a static program 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 $\max_1^n |c_i|$. Program c_i is active between $[0, |c_i|)$ 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 (§4.3.4).

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. We instead add 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 `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 component.

Group enable A leaf statement can refer to a `static group` (e.g., `do_add` in Figure 5.1). The latency is that of the group.

5.1.3 Unification Through Semantic Refinement

The new static constructs are all semantic *refinements* [50] 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 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. However, the other direction is not true: a `static` program may rely on specific timing behavior for its correctness and cannot be safely scheduled as a dynamic program.

Semantic refinement also enhances optimization (§5.4.3). Optimization passes in Calyx can now utilize timing information to expose more optimization opportunities in static code. New optimizations can also combine information across static and dynamic code enabling hybrid optimizations across static and dynamic operators. Finally, the refinement property also means that the Calyx compiler may automatically infer `static` qualifiers for some code (§5.4.2).

5.2 Targeting Static Abstractions

We overview how frontends can target Calyx's static abstractions by extending the systolic array generator the Dahlia-to-Calyx compiler (§§ 4.4.1 and 4.4.2).

5.2.1 Systolic Array

Static abstractions allow architectures like systolic arrays to precisely orchestrate their computation. We build upon the design described in Section 4.4.1 which uses groups to move data across the registers and schedules their execution using the following control program:

```
while data_is_valid {
  seq {
    par { invoke PE_00(..); invoke PE_01(..); ...}
    seq { // data movement
      par { move_l0n; move_l1n; ..; move_tn0; move_tn1 .. }
      ...
      par { move_l00; move_l10; ..; move_t00; move_t01; .. }
}}}
```

Recall that each step in the systolic array takes N cycles where N is the number of rows because Calyx's `par` operator does not guarantee lockstep execution of the groups. For example, this means that we are moving data from register t_{00} to t_{10}

and t_{10} to t_{20}, we cannot guarantee that the reads and writes from the register will occur in the same cycle. This means that t_{00} might write its value into t_{10} before t_{10} has written its value into t_{20}. To ensure correct date movement, we have to sequence each step.

In contrast, with Calyx's new static abstractions, we can invoke all of these groups in parallel:

```
static<1> group move_l01 {
  l01.in = l00.out; l01.write_en = 1;
}
static<1> group move_l02 {
  l02.in = l01.out; l02.write_en = 1;
}
...
control {
  while data_is_valid {
    seq {
      par { invoke PE_00(..); invoke PE_01(..) }
      static par { move_l00; move_l01; ..; move_tnn; }
}}}
```

This made possible because we know that writes to registers take exactly one cycle and that the `static par` operator starts executing each thread in the same cycle. Furthermore, if the processing element can accept new inputs every clock cycle, we can further optimize the control program.

```
while data_is_valid {
  seq {
    static par {
      invoke PE_00(..); invoke PE_01(..);
      move_l00; move_l01; ..; move_tnn;
}}}}
```

Our full systolic array generator takes advantage of this to implement efficient matrix multiplication followed by post operations (§6.2).

5.2.2 Dahlia Compiler

Static abstractions can also be useful for optimizing Dahlia programs. For example, when multiplying two numbers, we know that the computation will take exactly three cycles. This means we can write the following `static` group:

```
static<3> group do_mult {
  m0.left = a; m0.right = b; m0.go = 1;
}
```

However, it might end up being the case that this would be the only static group surrounded by many other dynamic groups:

```
while cond {
  read_a;  // dynamic
  do_mult; // static
  write_b; // dynamic
}
```

In such cases, the cost of compiling an individual group statically might outweigh the benefits it provides (§5.3). Instead of *requiring* the compiler to schedule this group statically, we would instead like to provide the option to the compiler in case it is useful. For this purpose, the Dahlia compiler will still emit a dynamic group but add an annotation to tell Calyx that this group *may* be scheduled statically if beneficial:

```
group do_mult<"promote"=1> {
  m0.left = a; m0.right = b; m0.go = 1; do_mult[done] = m0.done;
}
```

Our new static promotion pass (§5.4.1) might decide to schedule this group statically if it ends up being beneficial.

This demonstrates an alternative and incremental use case for static abstractions: instead of requiring frontends to make the decision about which parts of programs are best represented statically, we give the ability to add information to

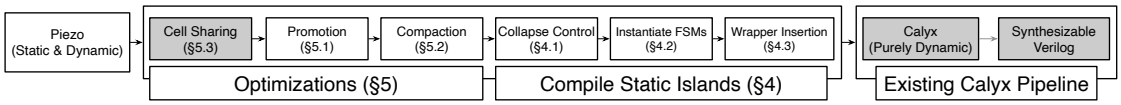


Figure 5.2: New compilation flow. Static operators are optimized (§5.4) and compiled (§5.3) to pure Calyx abstractions.

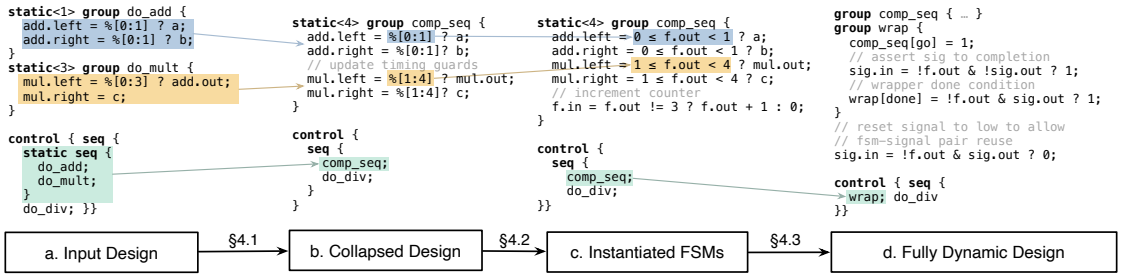


Figure 5.3: Compilation flow for static abstractions. Static groups and control are inlined (§5.3.1) and the relative timing guards are reified using counters (§5.3.2). Dynamic control operators interface with compiled code using wrapper groups (§5.3.3).

provide more optimization opportunities.

5.3 Compilation

Figure 5.2 shows the new compilation flow. After optimizations (§5.4), we translate static constructs to dynamic ones and compile them using the standard Calyx compilation flow (§4.5).

Compilation has to be reason about interfaces for static code, dynamic code, and invocations that cross the static–dynamic boundary. For example, in a con-

Abbr.	Caller	Callee	Calling Convention
$D \to D$	Dynamic	Dynamic	Calyx [116]
$S \to S$	Static	Static	See §5.3.1
$D \to S$	Dynamic	Static	See §5.3.3
$S \to D$	Static	Dynamic	Not supported

Table 5.1: Interfaces between types of control.

trol statement like `seq { a; b; }`, both the parent (the `seq`) and the children (`a` and `b`) could use either static or dynamic control. Table 5.1 lists the four possible cases, denoted $I_p \to I_c$ where the parent and child interfaces I are static (S) or dynamic (D). The all-dynamic case, $D \to D$, is the Calyx baseline. The all-static case, $S \to S$, works by counting cycles (§5.3.1). For $D \to S$, the compiler adds a *dynamic wrapper* around the static child (§5.3.3). $S \to D$ is disallowed and results in 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 $S \to D$ composition, we can think of any Calyx program as a dynamic control program with interspersed *static islands* [34, 35].

Compilation starts by *collapsing* static islands into static groups (§5.3.1) and then generating FSM logic to implement relative timing guards (§5.3.2). Finally, it *wraps* static islands for use in their dynamic context (§5.3.3).

5.3.1 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 5.3a–b provides an example of how Calyx 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.

The compiler collapses 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 5.3a 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 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 §5.3.2) 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.

5.3.2 FSM Instantiation

Figures 5.3b–c illustrate the next compilation step: eliminating static timing guards (§5.1.1). 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.

5.3.3 Wrapper Insertion

Figures 5.3c–d illustrate the final compilation step: converting each collapsed, timing-guard-free static group (5.3c) into a dynamic group (5.3d).

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: Calyx's `while` operator admits multiple possible cycle-level timing behaviors, and we generate a specific one to meet our objectives.

5.4 Optimizations

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

5.4.1 Static Inference and Promotion

A frontend might choose to generate dynamic Calyx code simply out of convenience: the computation only uses latency-sensitive operations, but it is easier to compositionally generate dynamic programs. Automatically *promoting* such code to use static interfaces can save time and resources for dynamic signalling—but it is not always profitable. We decouple the detection of promotion opportunities from the decision to promote any specific subprogram into two passes. *Inference*, detects when dynamic groups and control have a static latency, and *promotion*, 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 `@promote(1)` attribute (§4.3.5) to group `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)[1] 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.

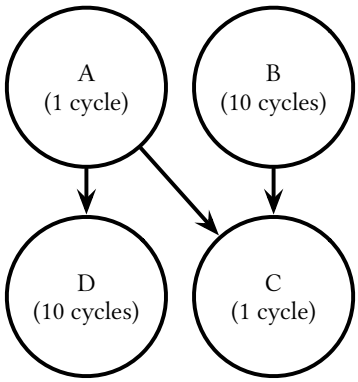
For control operators, e.g., `seq`, inference works bottom-up. If all of a `seq`'s children have `@promote` annotations, the `seq` gets a `@promote(`n`)` annotation where n is the sum of the latencies of its children.

Promoting code from dynamic to static We can promote groups and control based on inferred `@promote` annotations. For example, after inferring the `@promote(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 *thresh-*

[1]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.

Figure 5.4: Schedule compaction uses data dependencies to generate an *as-soon-as-possible* schedule.

old parameter that only promotes static islands above a certain code size, in terms of the number of groups and conditional ports.

5.4.2 Schedule Compaction

We design a new *schedule compaction* optimization to maximize parallelism while respecting data dependencies. For a dynamic program that can be promoted to static, we have a choice in how to implement its static schedule.[2] We can choose a specific static schedule to implement the original dynamic schedule that respects the original data dependencies and performs as many computations in parallel as possible, produces dependencies as soon as possible (ASAP scheduling), or as late as possible (ALAP scheduling).

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 tim-

[2]A program marked `static` by a frontend cannot use this optimization since the compiler is required to implement that program with a specific static schedule.

ing 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.

After static inference, dynamic control programs are marked with the `@promote(n)` attribute. The compaction pass targets such control programs for compaction. Consider the following `seq`:

```
promote(22) seq  A; B; C; D;
```

where Figure 5.4a shows the groups' latencies and data dependencies. If we only perform promotion, it will take $1 + 10 + 1 + 10 = 22$ cycles.

The 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. The pass 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 5.4b. Since all `delay_n` groups are removed during the collapsing step of compilation (§5.3.1), they incur no overhead.

5.4.3 Latency-Aware Component Sharing

Calyx's component sharing pass (§4.6.2) computes the live-ranges of components and uses that information to optimize resource usage. We extend the sharing pass to work with both static and dynamic code. Because of Calyx's unified and compositional representation for static and dynamic programs, the new pass can share cells across the static–dynamic boundary. This is infeasible in ILs that separate static and dynamic operators [33, 163].

Our implementation also improves over sharing in Calyx when it can exploit cycle-level timing in static code.

```
par {
  seq {
    write_a; // lat = 1
    read_a;  // lat = 1
    do_mult0; // lat = 3
```

```
  }
  seq {
    do_mult1; // lat = 3
    write_b; // lat = 1
    read_b;  // lat = 1
  }
}
```

Consider the following program that executes two threads in parallel that use registers `a` and `b` respectively. The first threads writes to and reads from register `a` in the first two cycles while the second thread writes and reads from register `b` in the last two cycles. However, because Calyx's `par` operator does not provide any guarantees on how the threads are scheduled to execute (§4.3.4), the pass has to conservatively assume that the live ranges of `a` and `b` extend to the end of the `par` block.

With our new latency-sensitive abstractions, a `static par`, will guarantee precisely when each group executes allowing the pass to precisely compute the live ranges. Using this information, our new pass can automatically share registers `a` and `b`.

5.5 Discussion

We overview the static extension to Calyx which allow for latency-sensitive reasoning of circuits. Using the insight that latency-sensitive schedules are *refinements* of latency-insensitive ones, our extended IL retains the compositional and modular nature of Calyx. We demonstrated that (1) the extension can be used to precisely encode cycle-accurate circuits (§5.2), (2) they can be efficiently compiled to preserve their timing guarantees to existing abstractions in Calyx, and (3) they improve the optimization power of the compiler (§5.4). Chapter 6 studies the com-

plete Calyx system through case studies and how Calyx has been used in the wild.

CHAPTER 6

EVALUATING CALYX

Chapters 4 and 5 overview the design of Calyx. This chapter discusses the implementation of the Calyx compiler, how it is used to design frontends, and how its optimizations improve the performance of generated programs.

6.1 Implementation of the Calyx Infrastructure

The Calyx infrastructure is implemented using Rust, Python, and C++. The core compiler infrastructure is implemented in 33,000 lines of code and structured as several packages:

- `calyx-frontend`: Parses the Calyx intermediate language (IL) textual format and converts into an abstract syntax tree (AST).
- `calyx-ir`: The core data structures implementing the intermediate representation (IR).
- `calyx-opt`: Analyses and passes used by the compiler. Each pass consumes and generates a valid Calyx program.
- `calyx-backed`: SystemVerilog and FIRRTL [80] backends for the Calyx programs.

The Calyx ecosystem also implements various other tools to simplify compilation from frontends, simulation, debugging [17], and execution on field programmable gate arrays (FPGAs) amounting to a total of 80,000 lines of code.

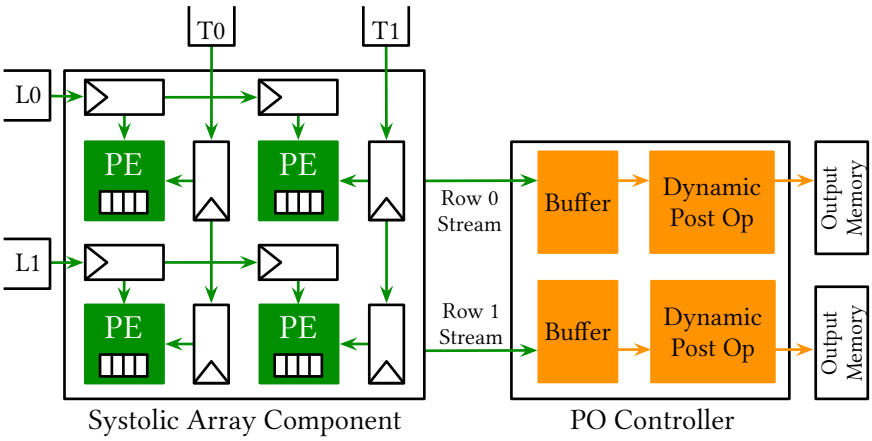


Figure 6.1: Our 2×2 systolic array with a *dynamic* post op. The buffers in the post op controller are not necessary for static post ops. Green is static and orange is dynamic. Input memories (`L0`, `L1`, `T0`, `T1`) may have non-fixed length.

CIRCT Calyx has been adopted as a part of the LLVM CIRCT infrastructure [153] and is implemented as a separate MLIR dialect [97]. This implementation of Calyx is capable of generating the native Calyx textual format which can be passed to the native, Rust-based Calyx compiler. The native compiler is capable of produce an MLIR complaint textual representation allowing the two systems to interoperate.

6.2 Systolic Arrays

Sections 4.4 and 5.2 overviewed the implementation of simple systolic arrays [92] in Calyx. In this section, we explore a complete design of a systolic array *generator* in Calyx. The generator is parameterized by the contraction dimension of the resulting matrix and implements an output-stationary dataflow. For example, if matrices of size $A \times N$ and $N \times B$ were multiplied, the generated matrix would have size $A \times B$. Furthermore, our generator implements and connects various *post*

operations that can be performed on the resulting matrix.

6.2.1 Design Considerations

We describe various requirements of systolic arrays generated by our frontend:

- **Architecture:** Our systolic arrays should support output-stationary dataflow which stores which allow both statically-known and runtime specified size of the contraction dimension.
- **Expressiveness:** The architecture should accommodate both latency-sensitive and latency-insensitive post operators.
- **Efficiency:** The systolic array should efficiently pipeline the computation using the fully-pipelined processing elements (PEs).

These requirements make the design of the compiler frontend challenging. For example, to accommodate both latency-sensitive and latency-insensitive components, a compiler that targets register transfer level (RTL) language would have to carefully track the information and generate control logic that orchestrates the computation correctly and leverages the timing information for efficiency. On the other hand, a compiler that targets an high-level synthesis (HLS) tool will need to carefully structure the C++ code to ensure that a systolic array architecture is generated. Finally, if using a custom intermediate language (IL), systems that only expose static interfaces [103] or dynamic interfaces [83] will limit expressiveness or efficiency respectively.

A Calyx-based frontend works around these problems. First, Calyx's control operators make it easy to orchestrate the systolic computations. Second, the hardware-

like structural operators enable precise specification of the architectural components. Finally, Calyx's unified representation for static and dynamic interfaces enables us to easily support static and dynamic PEs and post operations.

6.2.2 Implementation

Figure 6.1 overviews the architecture. The systolic array fabric is instantiated using pipelined processing elements and connected to the post-operations (POs) using a controller. The POs can be either static or dynamic and instantiated for each row in the output matrix. If the POS is dynamic, the *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. This is because Calyx allows programs to freely intermix static and dynamic abstractions.

Pipelined processing elements Processing elements are implemented as components that conform to the following interface:

```
static<1> component pe(go: 1, top: 32, left: 32) -> (out: 32) { ... }
```

The requirement that the component is marked `static<1>` ensures that it can accept a new input every clock cycle. This also allows the systolic array to unconditionally move data every cycle and guarantee the right value will be read.

Flexible iteration logic While static interfaces allow for efficient, pipelined execution, they can limit computational flexibility. For example, an output-stationary

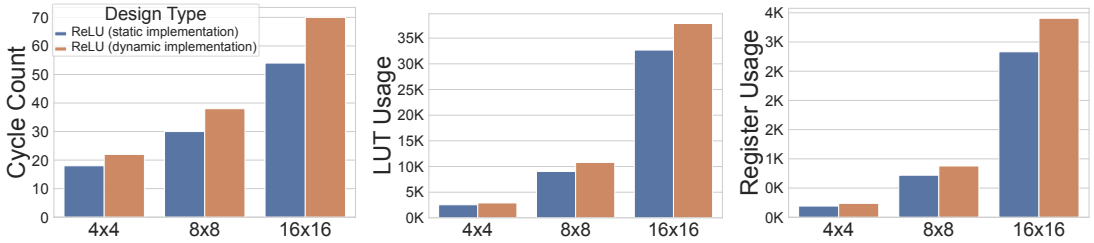
matrix-multiply systolic array should be able to multiply matrices of sizes $i \times k$ and $k \times j$ for any value of k. However, this requires dynamic control flow: the computation needs to repeat based on the run-time value k. Calyx abstractions support this with ease: we simply use a `while` loop to execute the systolic array's logic k times. Because the control logic to execute the processing elements and move the data are completely static, Calyx's special handling of loops (§5.3.3) executes the body every cycle.

Supporting fused post operations Machine learning frameworks [9, 30, 135] *fuse* matrix multiplication with element-wise *post operations*, such as non-linearities, to avoid writing the intermediate matrix back to memory. These post operations can be either 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 [102] 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.

6.2.3 Evaluation

Our evaluation of the systolic arrays seeks to answer the following questions:

- How do static abstractions improve the systolic array?
- How is the cost of implementing run-time-configurable contraction dimension for systolic arrays?



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

Figure 6.2: Performance and FPGA resource utilization of two implementations of a fused matrix-multiply–ReLU kernel on Calyx-compiled systolic arrays. We compare static and dynamic interfaces for the ReLU unit.

- Do cross-boundary optimizations let Calyx 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 Calyx achieves a max frequency of 270 MHz and performs the computation in 52 cycles in comparison to the 250 MHz and 284 cycles taken by a Calyx design that does not use static abstractions. 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 look-up tables (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: Calyx 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:

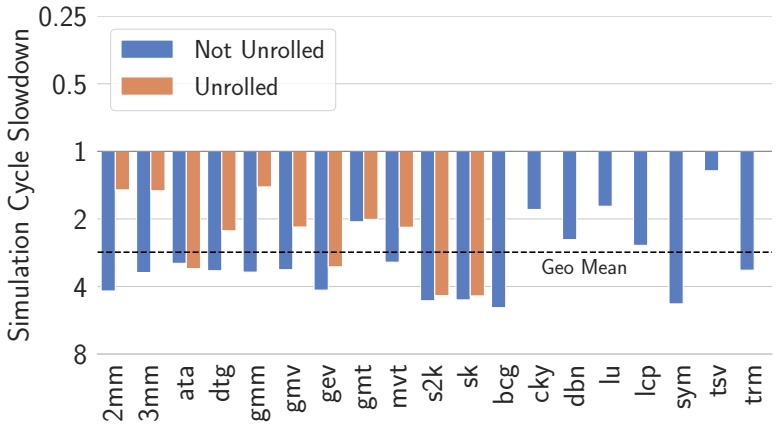
we use the simple ReLU post operation in its default, static form and compare it against a version that artificially wraps it in a dynamic interface. Since the computation is the same, the only difference is the interface. Figures 6.2a–6.2c report the cycle counts, LUTs, and register usage of the resulting 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.

6.3 Dahlia Compiler

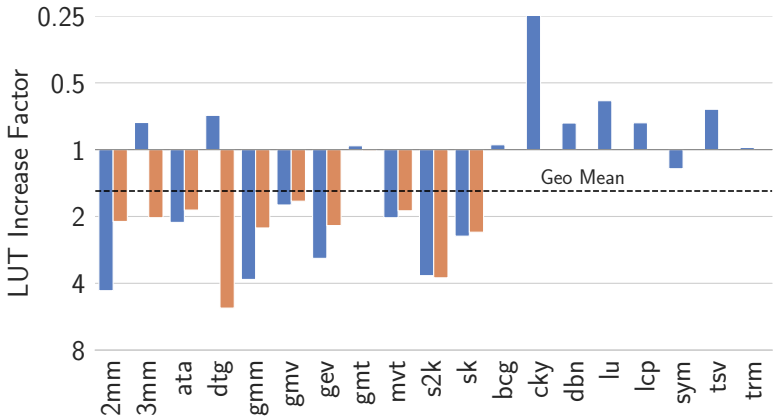
The Dahlia compiler (§§ 4.4.2 and 5.2.2) transforms Dahlia programs (§3) to Calyx. We compare the Calyx-generated register transfer level (RTL) against the original Dahlia compiler which emits C++ and relies on Vitis HLS to generate hardware designs. We implement all 19 kernels from the linear algebra category of the PolyBench [3] benchmark suite and, for the 11 benchmarks Dahlia's type system allows it, unroll the loops to unlock parallelism. We synthesize and place-and-route each design using Vivado and targeting the Zynq UltraScale+ board with a 7ns target frequency to produce resource numbers. We produce cycle counts by simulating each design using Verilator [155].

Comparison against HLS We collected cycles counts (Figure 6.3a) and LUT usage (Figure 6.3b) for each benchmark with all optimizations turned on and normalized them to the corresponding Vitis HLS implementation. For the unrolled designs, we normalize against the corresponding unrolled HLS designs. Since DSP and BRAM usage is almost identical for all benchmarks, we elide them.

On average, the Calyx generated designs are 3.1× slower than the designs



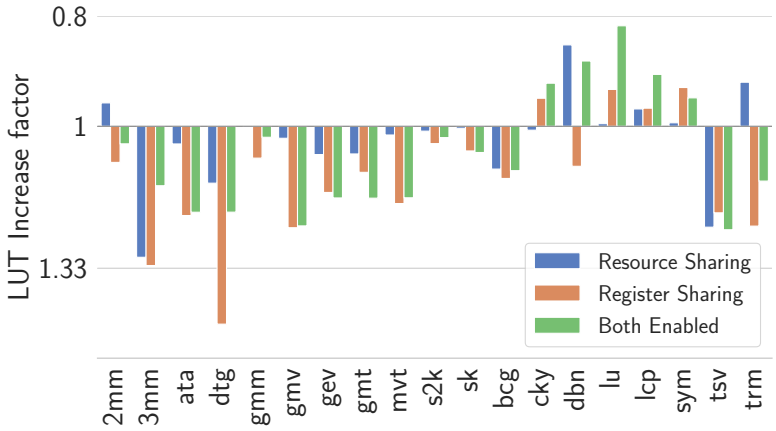
(a) Cycle slowdown of Calyx designs compared to Vitis HLS. Designs below the y-axis are slower.



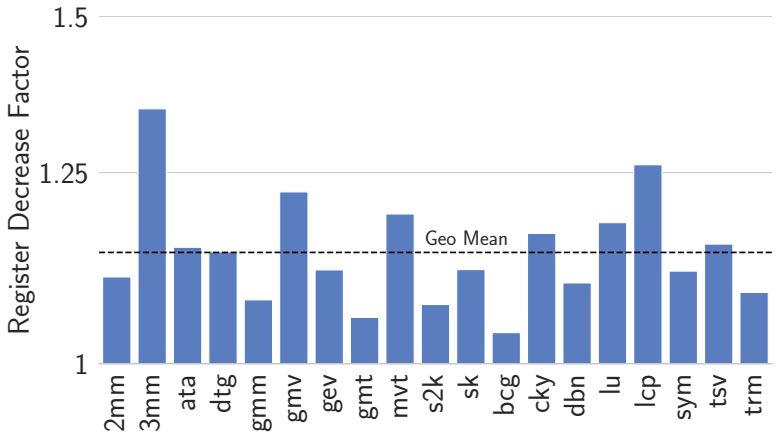
(b) LUT increase of Calyx designs over Vitis HLS. Designs below the y-axis are larger.

Figure 6.3: Resource and cycle count comparison for Dahlia-generated Calyx designs and HLS designs for PolyBench benchmarks. Missing unrolled bars indicate that the benchmark was not unrollable in Dahlia.

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



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



(b) Register decrease from the register sharing optimization.

Figure 6.4: Effects of optimization passes. All graphs use logarithmic scales.

6.4 Effect of Optimizations

We implement several optimization passes to share resources and improve performance of Calyx designs and study their impact.

6.4.1 Resource Sharing

To demonstrate Calyx's ability to express control-flow based optimizations, we wrote a resource sharing pass (§4.6.1) and a register sharing pass (§4.6.2). We

perform an ablation study to characterize their effects on the final designs.

Figure 6.4a reports the resource utilization of PolyBench benchmarks in three configurations: (1) resource sharing enabled, (2) register sharing enabled, and (3) both resource sharing and register sharing turned on. We normalize the resource counts against baselines with both passes disabled.

While both optimization passes find opportunities to share hardware components, there is not a uniform drop in the LUT usage. On average, the resource sharing pass increases LUT usage by 3% and the register sharing pass increases LUT usage by 11%. Sharing hardware components causes additional multiplexers to be instantiated which makes the resource usage worse in some cases.

Figure 6.4b shows the effects of the register sharing pass on the number of registers used in the designs. On average, the pass reduces register usage by 12% and finds register sharing opportunities in every benchmark. Registers, compared to multiplexers, are more expensive in ASIC processes which represents another opportunity for heuristics in a future version of the Calyx compiler.

6.4.2 Impact of Static Abstractions

Calyx's static abstractions allow programs to be optimized and rescheduled based on resource and performance requirements. We extend the resource sharing passes to take advantage of the implicit synchronization information present in static programs (§5.4.3). Additionally, we study the impact of the *promotion* and *compaction* optimizations enabled by static abstractions (§5.4.2).

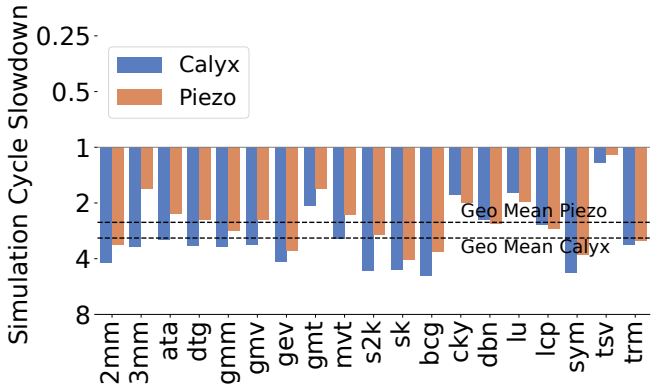
Configurations Since the resource sharing and compaction optimizations interfere with each other, we generate several configurations and vary the order of the optimizations.

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.

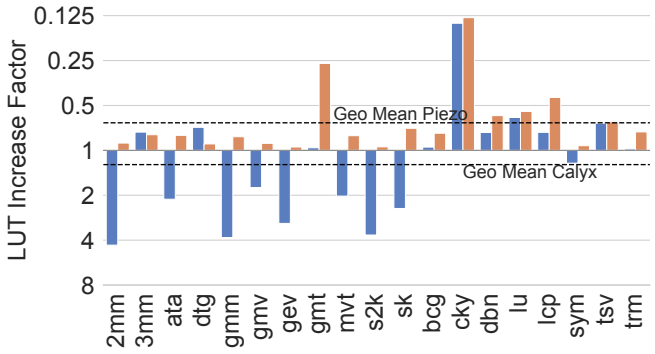
We compare Polybench designs writte using Calyx's static abstractions (referred to as 'Piezo' in the experiments) as well as vanilla Calyx and Vitis HLS. The Vitis HLS implementations provide an external baseline while the vanilla Calyx designs serve to quantify the impact of static optimizations. Our experiments to explore the threshold parameter for promotion (§5.4.1): while they unsurprisingly yield nonuniform trade-offs between area and latency, we select a default parameter of two for the static island size threshold.

Comparison to Calyx and Vitis HLS

We use Piezo's **SC→SH** configuration to compare against Calyx and Vitis; Figure 6.5 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.



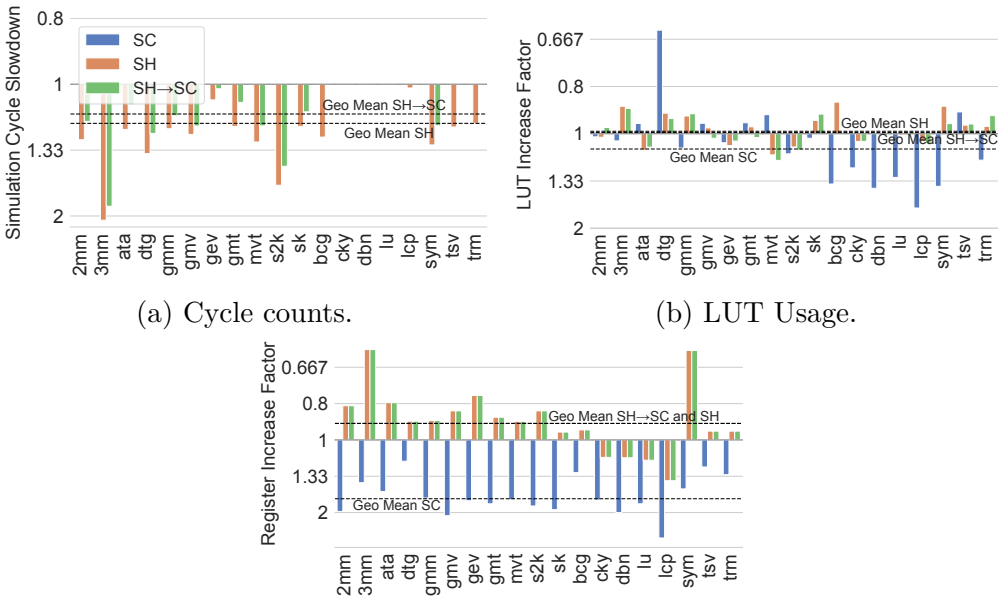
(a) Cycle counts.



(b) LUT Usage.

Figure 6.5: 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.

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 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.



(a) Cycle counts.

(b) LUT Usage.

(c) Register Usage.

Figure 6.6: 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).

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 6.6 shows the results for the various configurations, relative to the **SC→SH** configuration, which is the configuration used in §6.4.2.

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.

CHAPTER 7

MODULAR HARDWARE DESIGN WITH TIMELINE TYPES

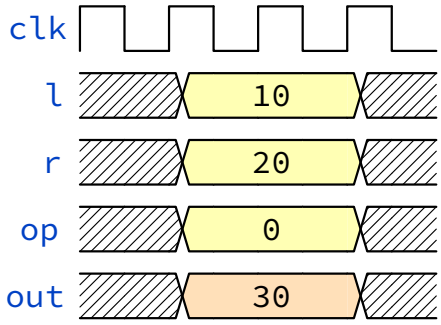
Dahlia (§3) and Calyx (§§ 4–6) demonstrate the possibilities of high-level programming models for hardware design. However, such programming models excel by specializing to particular domains and generating particular domain-specific architectures. While some work has explored the design of general-purpose architectures using such tools [104], a vast majority of general-purpose hardware design is still done using hardware description languages (HDLs). Automatic hardware generation (AHG) tools are more widely being used to design more complex datapath designs but HDLs are still used for carefully designing control logic and gluing together automatically generated hardware blocks.

However, existing HDLs complicate efficient modular design. Unlike software, hardware designs correspond to physical circuits and inherit constraints from them. Timing constraints—which cycle a signal arrives, when an input is read—and structural constraints—how often a multiplier accepts new inputs—are fundamental to hardware interfaces. Existing HDLs do not provide a way to encode these constraints; a user must read documentation, build scripts, or in the worst case, a module's implementation to understand how to use it. These problems are even more challenging to deal with in the presence of AHG tools which automatically generate and change the cycle-level timing behavior of circuits. Unlike Calyx, abstracting away timing details is not an option; timing details are critical for efficient composition and an HDL that abstracts away such details will impose unacceptable overheads.

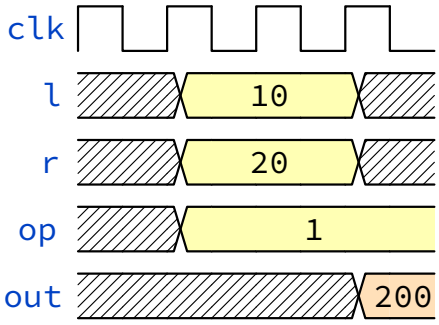
We present Filament, a language for modular hardware design which uses a type system to encode and enforce timing constraints. This chapter overviews the

```
module Add(a: 32, b: 32) -> (o: 32);
module Mul(a: 32, b: 32) -> (o: 32);
module Mux(sel: 1, a: 32, b: 32) -> (o: 32);
module ALU(op: 1, l: 32, r: 32) -> (o: 32) {
 Mul M(l, r); Add A(l, r);
 Mux Mx(op, A.out, M.out); o = Mx.out; }
```

(a) HDL implementation of ALU



(b) Waveform generated for addition.



(c) Waveform generated for multiplication.

Figure 7.1: ALU implementation and waveforms generated when executing addition and multiplication.

basic design of Filament and how it enables compositional design of individual circuits. Chapter 8 extends Filament to reason about *families of circuits* allow it to correctly and efficiently compose designs generated from AHG tools. Filament offers a new correctness-efficiency trade-off in the design of HDLs: by effectively modeling circuit constraints within a type system, we enable correct reasoning without overheads.

7.1 Example

We will discuss the challenges associated with compositional hardware design by implementing a pipelined arithmetic logic unit (ALU).

7.1.1 Traditional Hardware Description Languages

Figure 7.1a shows the implementation of the ALU in a traditional HDL. The interfaces for the modules specify the inputs and outputs along with their bitwidths. The ALU's circuit consists of an adder and a multiplier, which perform their computations in parallel, and a multiplexer, which selects between the two outputs using the `op` signal.

We will use waveform diagrams to understand the execution behavior of this module. A waveform diagram explains the flow of signals in the circuit over time and usually with respect to the global clock signal. Figure 7.1b shows the waveform generated when the ALU is provided with the inputs 10 and 20 and the op code 0. Note that the output 30 is produced in the same cycle as the inputs. However, Figure 7.1c shows what happens when we attempt to execute the multiplication operation by setting `op` to 1. The timing behavior of the ALU changes—the product is produced two cycles after the input is provided. Additionally, if the `op` is not asserted for an additional cycle, the output is wrong. The problem is that an adder is *combinational*—it produces its output in the same cycle as the inputs—while a multiplier is *sequential*—it takes several cycles to produce its output. `op` is required for an extra cycle because the multiplier output is produced later than the adder and the multiplexer needs to select the correct output in a later cycle using the `op` input.

The interfaces for `ALU`, the adder, and the multiplier do not capture these details. One option to sidestep this problem is to "wrap" every module in a *latency-insensitive* interface, such as ready–valid handshaking. But these interfaces incur overhead that can be prohibitive for fine-grained composition [111]. In this chapter, we design a hardware description language (HDL) to specify efficient, *latency-sensitive* interfaces based on clock cycles and to statically rule out misuses of these

interfaces.

7.1.2 Filament

Filament is an HDL that allows users to directly *specify* and *check* the timing behavior of their modules. Each component can be parameterized by multiple events which are used to specify its timing behavior. Our ALU implementation has behaves unpredictably because adders and multipliers have different timing behavior. Filament allows us to encode their timing behavior explicitly using *events* which parameterize modules:

```
extern comp Add<T>(
  @interface[T] go: 1, left: [T, T+1] 32, right: [T, T+1] 32) -> (out: [
      T, T+1] 32);
extern comp Mult<T>(
  @interface[T] go: 1, left: [T, T+1] 32, right: [T, T+1] 32) -> (out: [
      T+2, T+3] 32);
```

Both components use the event T to specify their timing behavior. The adder is *combinational*—it produces outputs in the same cycle as the inputs. This fact is encoded by the *availability intervals* of the inputs and outputs: the inputs are provided in the half-open interval $[T, T+1)$, which corresponds to the first cycle of execution of the component, and the output is produced during the same interval. In contrast, a multiplier is *sequential*—it takes two cycles to produce its output. This is encoded by stating that the output is available in the interval $[T+2, T+3)$, two cycles after the inputs are provided in the interval $[T, T+1)$. In order to signal that the event T has occurred, a user of these modules must set the *interface port* `go` to 1, provide the inputs according to their required intervals, and read the output when they are available. Multiplexers (not shown) are also combinational and take all their inputs in the same cycle.

```
comp ALU<G>(
  interface[G] en: 1 op: [G, G+1] 1, l: [G, G+1] 32, r: [G, G+1] 32,
) -> (o: [G+2, G+3] 32) {
  A := new Add; M := new Mult; Mx := new Mux;
  a0 := A<G>(l, r);
  m0 := M<G>(l, r);
  mux := Mux<G>(op, m0.out, a0.out);
  o = mux.out; }
```

Like our HDL implementation, our Filament implementation of the ALU explicitly instantiates all the hardware resources it needs to use. The key difference is how Filament expresses the use of the hardware instances through *invocations*. An invocation schedules the execution of a hardware instance using a particular set of events and provides all inputs. For example, the invocation `a0` of the adder `A` is scheduled using the event G. By naming uses, Filament can check the timing behavior of the module. There is no assignment for the `go` port of the adder—it is automatically inserted by the compiler using the scheduling event G. Invocations are a logical construct that are compiled away by Filament (§7.4). Similarly, the multiplier and multiplexer are also scheduled using the event G. Instead of using outputs from the instance, the multiplexer uses the ports on the invocations, reflecting the output from a particular use.

7.1.3 Checking Timing Behavior

However, when we attempt to compile this program, Filament gives us the following error:

```
mux := Mux<G>(op, m0.out, a0.out);
Available for [G+2, G+3) but required during [G, G+1)
```

The error states that the use of our multiplexer expects all of its inputs during $[G, G+1)$ while the multiplier's output, `m0.out`, is available in $[G+2, G+3)$. Filament requires that all inputs be available for at least as long as the correspond-

ing argument's requirement. This was the problem in our original HDL design (§7.1.1)—the output of the adder is available in a different cycle from the multiplier which results in unexpected timing behavior. Filament's type system statically catches this error.

The solution is to use *registers* to store values and make them available in future cycles. A register's signature captures its timing behavior—the output is available one cycle after the input[1]:

```
comp Reg<G>(en: interface[G], in: [G, G+1] 32) -> (out: [G+1, G+2] 32)
```

The corrected implementation uses two registers to make the sum available in the same cycle as the multiplier. The outputs from the first and second registers are available in $[G+1, G+2)$ and $[G+2, G+3)$, respectively. We schedule the execution of the multiplexer in cycle $G+2$ when both the outputs are available. This design is still problematic because the `op` is only available in $[G, G+1)$ while the multiplexer reads it in $[G+2, G+3)$. We fix this by making `op` signal available in $[G, G+3)$. This results in a correct ALU implementation.

```
comp ALU<G>(@[G, G+3] op: 32, ...) {
 a0 := A<G>(l, r); R0 := new Reg; R1 := new Reg;
 r0 := R0<G>(a0.out); r1 := R1<G+1>(r0.out);
 mux := Mux<G+2>(op, r1.out, m0.out); ... }
```

However, it is not clear when the ALU is ready to accept new inputs: should we wait till outputs are produced or can the module process multiple inputs in parallel?

(a) Sequential processing (b) Pipelined processing

Figure 7.2: Difference between sequential and pipelined processing. A pipelined module can process multiple inputs at the same time.

7.1.4 Pipelining

Pipelining is a common optimization that enables hardware to process multiple inputs in parallel. A sequential module processes its inputs one at a time (Figure 7.2a), while pipelined module can overlap the processing of multiple inputs (Figure 7.2b).

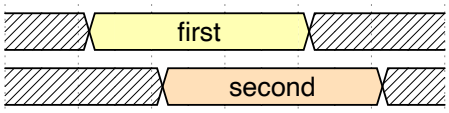
Pipelining is challenging because it requires reasoning about the interaction between multiple, concurrent executions of the same physical resources—correctly pipelining requires using values from the correct pipeline stage and ensuring there are no *structural hazards*, i.e., there are no conflicting uses of internal components.

```
comp Add<T:1>(...); comp Mult<T:3>(...); comp ALU<G:1>(...)
```

Filament presents a concise solution: each event has an associated *delay* that specifies how many cycles to wait before accepting new inputs. We can update the signature of the adder and multiplier to reflect this. Since the adder is combinational, it can accept new inputs every cycle. However, the multiplier accepts new inputs every 3 cycles. For user-level components, Filament ensures that the *delay* for each event is correct, i.e., the component can be correctly pipelined. We'll redesign our ALU to be pipelined and accept new inputs every cycle by specifying that the delay of G is 1. Since we know our design is not pipelined, Filament will generate errors explaining why the design cannot be pipelined.

[1]This is a simplified interface for a register. Full interface provided in §7.2.6.

```
comp ALU<G:1>(
    Event may retrigger every cycle
op: [G, G+3] 1, Lasts for 3 cycles
```



Our first problem is that the signature requires input signal `op` to be available for three cycles whereas the pipeline may trigger every cycle. The waveform diagram demonstrates the problem—the input for `op` from the first iteration will overlap with the input for the second iteration. However, `op` is a *physical port* in a circuit and can only hold one value at a time; this is a fundamental physical constraint of hardware design. Filament requires that the delay of an event is at least as long as the length of any availability interval that uses it; we must make `op`'s availability interval 1-cycle long. We choose $[G+2, G+3)$ since the multiplexer uses `op` during this interval.

```
comp Mult<T: 3>(
Event may retrigger every 3 cycles
comp ALU<G: 1>(
Event may retrigger every cycle
 m0 := M<G>(l, r);
       Cannot safely pipeline
```

Next, Filament complains that while our ALU pipeline may accept new inputs every cycle, the multiplier `M` can accept new inputs every 3 cycles. This is a fundamental limitation of the multiplier circuit we're using; to fix it, we must use a different multiplier. Filament catches yet another pipelining bug that arises from composition: every subcomponent used in a pipeline must be able to process inputs at least as often as the pipeline itself. Fixing this will result in a correct, fully pipelined ALU. A key goal of Filament is to ensure that changing the pipelining behavior of a component does not create additional bugs—the pipelined ALU, like the sequential ALU, only uses signals when they are semantically valid.

```
comp ALU<G: 1>(@interface[G] en: 1, @[G+2, G+3] op: 1, ...) {
  A := new Add; Mx := new Mux; R0 := new Reg; R1 := new Reg;
     FM := new FastMult; // delay = 1
  a0 := A<G>(l, r); r0 := R0<G>(a0.out); r1 := R1<G+1>(r0.out); m0 := FM
     <G>(l, r);
```

```
def init(left) -> (acc, q):
  # Initialize the
      computation
def nxt(a, q, div) -> (an,
    qn):
  # One step of the
      computation
def div(l, r):
  (qn, an) = init(l)
  for _ in range(0, 8):
    (qn, an) = nxt(an, qn, r
        )
  return qn
```

(a) Pseudocode for restoring division.

```
comp Comb<G: 1>(...) -> (
  out: [G, G+1] 8) {
 i := new Init<G>(left);
 n0 := new Nxt<G>(i.A, i.Q,
     r);
 ...
 n7 := new Nxt<G>(n6.A, n6.Q
     , r);
 out = n7.Q;
```

(b) Fully combinational divider.

```
comp Pipe<G: 1>(...) -> (@[G+7, G+8] q:
    8) {
 i := new Init<G>(left); // Instantiate
     and invoke
 n0 := new Nxt<G>(i.A, i.Q, r);
 ra0 := new Reg<G>(n0.A);
 rq0 := new Reg<G>(n0.Q);
 n2 := new Nxt<G+1>(ra0.out, div, rq0.out
     );
 ...
 out = n7.Q;
```

(c) Pipelined divider. Instances scheduled in successive cycles.

```
comp Iter<G: 8>(...) -> (@[G+7, G+8] q:
    8) {
  I := new Init<G>(left);
  N := new Nxt; RA := new Reg; RQ := new
      Reg;
  n0 := N<G>(i.A, i.Q, r);
  ra0 := RA<G>(n0.A); rq0 := RQ<G>(n0.Q);
  n1 := N<G+1>(ra0.out, rq0.out);
  ra1 := RA<G+1>(n1.A); rq1 := RQ<G+1>(n1
      .Q); ...
  out = n7.Q;
```

(d) Iterative divider. Components reused over multiple cycles.

Figure 7.3: Implementations of 8-bit restoring division demonstrating area-throughput trade-off. Filament's type system ensures that each implementation is correctly pipelined and introduces no resource reuse conflicts.

```
  mux := Mux<G>(op, r1.out, m0.out); o = mux.out; }
```

7.1.5 Area-Throughput Trade-offs with Filament

While pipelining improves the throughput of a component, it also increases its resource usage. For large circuits, like floating-point multipliers, it often makes sense to reuse the same circuit over multiple clock cycles. However, circuit reuse affects pipelining behavior: the ability of a component to start new iterations depends upon how sub-components are being shared. Filament's type system tracks

resource reuse and ensures that a well-typed component does not create structural hazards for reuses components.

To demonstrate how Filament enables safe exploration of *area-throughput trade-offs*, we implement three different versions of a divider using a restoring division algorithm (Figure 7.3a). The combinational components `Init` and `Nxt` compute a quotient (`.Q`) and an accumulator value (`.A`). For an 8-bit value, we must apply `Nxt` 8 times.

Combinational divider Figure 7.3b implements a combinational divider which computes the output in the same cycle when the inputs are provided. All `Nxt` instances are scheduled using the event G which means that they'll execute in the same cycle. While the latency of the design is 1, it is quite inefficient because it schedules a lot of complex logic in the same clock cycle and forces the design to operate at a low frequency. However, combinational designs are a good starting point to ensure that our algorithm is correct.

Pipelined divider To make our design run at a higher frequency, we can pipeline it by scheduling each `Nxt` instance to execute in successive cycles. To correctly forward the values, we instantiate registers to hold onto values of the quotient and the accumulator for each `Nxt` component. Figure 7.3c shows the implementation: the delay of the module remains 1, allowing it to process a new value every cycle, but the latency is now 8 cycles unlike the combinational implementation. The pipelining also breaks up the long combinational path allowing the design to operate at a higher frequency.

Iterative divider Both the combinational and pipelined inputs can process a new input every cycle but require a large amount of hardware since they instantiate 8 instances of the `Nxt` component and 16 registers for the pipelined version. We can instead use the same `Nxt` component and registers by implementing an *iterative design*.

```
comp Nxt<T:1>(...)
Delay requires uses to be 1 cycle apart
s0 := N<G>(i.A, div, i.Q); First use
s1 := N<G>(s0.AN, div, s0.QN); Second use
```

We start with our combinational design and change all the invocations to use the same instance N. Filament tells us that this design is buggy. We're attempting to send two different inputs into the `Nxt` instance in the same cycle. However, `Nxt` is a physical circuit and can only process one input every cycle. Therefore, we must schedule the uses of the instance in different cycles and add registers to hold onto the values, similar to the pipelined implementation.

```
comp Iter<G:1>(...)
Event may trigger every cycle
causing shared uses to conflict
s0 := N<G>(i.A, div, i.Q);
      First use
s7 := N<G+7>(s6.AN, div, s6.QN);
      Last use
```

With these changes, Filament complains with a new error message. Since we're sharing the instance `Nxt` over 8 cycles, the divider cannot start processing new inputs every cycle. Again, this is because `Nxt` is a physical circuit that can only process one input a cycle. To fix this, we can change the delay to 8 cycles which guarantees to Filament that the instance will only be run every 8 cycles, resulting in the final design (Figure 7.3d). This ensures that all iterations using the instance `N` complete before new inputs are provided. Implicitly, Filament showed us that reusing the instance is a trade-off: while we use fewer resources, our throughput

is also reduced since our iterative implementation can only process a new input every 8 cycles compared to every cycle for the pipelined implementation.

7.1.6 Summary

Filament is an HDL for safe design and composition of static pipelines. Specifically, Filament programs can *specify* and *check* timing properties of hardware modules and ensure that:

1. Values on ports and wires are only read when they are *semantically* valid.
2. Hardware instances are not used in a conflicting manner.

These properties ensure that the resulting pipelines are safe, i.e., there are no resource conflicts, and efficient, i.e., they can overlap computation as specified by their interface without any overhead. Filament's utility extends to components defined outside the language as well. By giving external modules a type signature, users can safely compose modules. Section 7.2 overviews the constructs in Filament, §7.3 explains how Filament's type system checks pipeline safety, and §7.4 shows how Filament's high-level constructs are compiled to efficient hardware.

7.2 The Filament Language

Figure 7.4 gives an overview of the Filament language. Filament's level of abstraction is comparable to *structural* HDLs where computation must be explicitly mapped onto hardware. Filament only has four constructs: components, instanti-

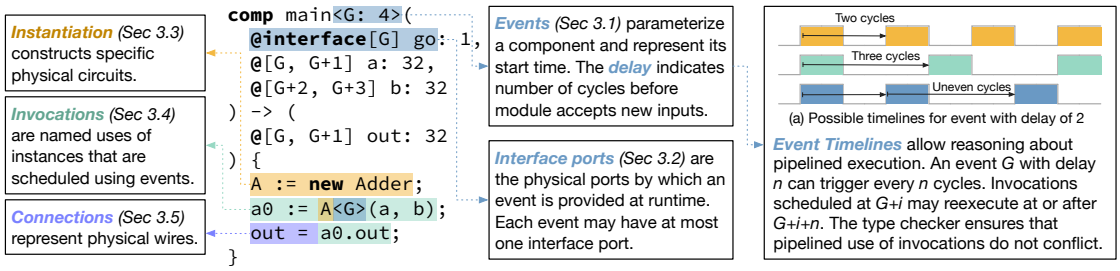


Figure 7.4: Overview of the Filament language. Programs are a sequence of *component* definitions which correspond to individual modules. The signature of the component is parameterized using *events*. The body of component consists of three types of statements: Instantiations, connections, and invocations.

ation, connections, and invocations. The first three have direct analogues in traditional HDLs while invocations are a novel construct.

7.2.1 Events and Timelines

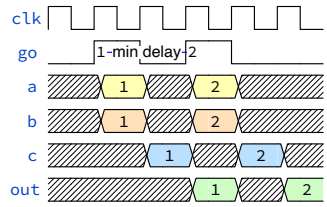
Events are the core abstraction of time in Filament. Instead of using a clock signal, designs use events to schedule computation. The Filament compiler generates efficient, pipelined finite state machines to reify events (§7.4.2).

Defining events There are only two ways to define events: (1) component signatures bind *event variables* like G, and (2) users can write *event expressions* such as $G+n$ where n is a constant. Events have a direct relationship to clock: if G occurs at clock cycle i, then $G+n$ occurs at clock cycle $i+n$.[2] This relationship with clock is crucial since it allows Filament to represent timing properties of components defined in clock-based HDLs. Adding event variables (G_0+G_1) is disallowed since events correspond to particular clock cycles, and it is meaningless to add them together.

[2]All event variables operate in the same clock domain, but this limitation can be removed in the future.

```
comp AddMult<G:2>(
  go: interface[G],
  a: [G, G+1] 32,
  b: [G, G+1] 32,
  c: [G+1, G+2] 32
) -> (out: [G+2, G+3] 32) { commands }
```

(a) Component's signature in Filament



(b) Pipelined use waveform

Figure 7.5: Signature and waveform diagram. The component allows pipelined execution or reuse after two cycles allowing overlapped execution. Shaded regions represent unknown values.

Timeline interpretation of events In order to capture potential resource conflicts from pipelined execution, Filament interprets events as a set of possible timelines. A timeline for an event G with a delay n is any infinite sequence of 1 cycle long clock pulses such that each pulse is at least n cycles apart. Figure 2a shows a set of valid timelines for an event with delay 2. By imbuing events with a timeline interpretation, Filament can reason about *repeated execution* and consider how pipelined executions may affect each other. By reasoning about such properties, we can define and enforce safety properties for pipelined execution of hardware. Furthermore, the timeline interpretation has a direct relationship to hardware: the delay of an event represents how many cycles a user must wait before providing a new set of inputs. This is usually referred to as the *initiation interval* of a pipeline by hardware designers (§7.3.3).

7.2.2 Components

Filament programs are organized in terms of *components* which describe timing behavior in their signatures and their circuit using a set of commands. Figure 7.5 shows the signature of a component in Filament (Figure 7.5a) and a waveform diagram visualizing two sets of inputs being processed in parallel (Figure 7.5b).

The component is parameterized using the event G with a delay of 2 which means that pipelined use can begin two cycles after the previous use.

Interface ports Hardware components typically have *control ports* which signal when values on *data ports* are valid and that the computation should be performed. Values on control ports are always considered semantically valid while values on data ports are only valid when the corresponding control port is high. Filament distinguishes control ports by defining them as *interface ports*. Interface ports are 1-bit ports that are associated with a particular event. When an interface port is set to 1, it signals to the component that the corresponding event has occurred. For example, setting `go` to 1 on an `AddMult` instance (Figure 7.5a) makes the module start processing the inputs. The availability intervals of all ports that use an event are relative to when the corresponding event's interface port is set to 1. If an event does not have an interface port, then the module can assume that the event triggers every n cycles where n is the event's delay.

Availability intervals The input and output ports of the component describe their availability in terms of the events bound by a component. For `AddMult` (Figure 7.5a), all ports use the event G. Availability intervals are *half-open*: for example, the input port `a` is available during $[G, G+1)$ which means it available during the first cycle when the component is invoked. Inside the body of a component, an input's availability interval represents a *guarantee* while an output's availability requires a *requirement* that the body must fulfill. When using a component, this is reversed: inputs have requirements that must be fulfilled by the user while outputs have guarantees.

7.2.3 Instances

All computations in a hardware design must be explicitly mapped onto physical circuits. Filament's `new` keyword allows instantiation of subcomponents. The following program instantiates two instances of the `Add` component named `A0` and `A1` that can be used independently. The instantiations do not provide bindings for the `Add`'s event T; *invocations* are responsible for providing those and scheduling the execution of an instance.

```
comp Add<T: 1>(left: [T, T+1] 32, @[T, T+1] right) -> (o: [T, T+1] 32);
comp AddTwo<G: 1>(...) { A0 := new Add; A1 := new Add; ... }
```

7.2.4 Invocations

```
F := new Reg; // FSM
F.in = F.out == 0 ? 1 : 0;
M := new Mult; A := new Add;
M.right = F.out == 0 ? r : M.out
M.left = F.out == 1 ? l : M.out
```

Resource reuse in hardware designs is *time-multiplexed*, i.e., different uses of the same resources are scheduled to occur at different times. This is done by building a finite state machine (FSM) using a register and using the output of the register to select which inputs to use. The example program computes $(l \times r)^2$ using a single multiplier using the FSM `F` to forward the inputs l and r into the multiplier in the first cycle and the output of the multiplier in the second cycle. However, the assignment to `M.left` incorrectly forwards the value from `M.out` in the first cycle. Mistakes in the control logic for the FSM do not lead to any visible errors; this error will lead to the data getting silently corrupted and propagating into other parts of the system.

```
comp Square<T:1>(left: [T, T+1] 32, @[T, T+1] right: 32) -> (
  out: [T+1, T+2] 32) {
```

```
M := new Mult;
m0 := M<G>(l, r);
m1 := M<G+1>(m0.out, m0.out) }
```

In contrast, every use of an instance in Filament must be explicitly named and scheduled through an invocation. The first invocation of the multiplier `M` is scheduled using the event G, uses the inputs l and r, and is named `m0`. The second invocation, scheduled one cycle later at $G + 1$, can then use `m0.out` to refer to the output of the first execution and pass it into the multiplier as an input. Because the second invocation is scheduled one cycle later, the input ports have a different requirement: the inputs must be available in the interval $[G + 1, G + 2)$ as opposed to $[G, G + 1)$ in the first invocation. This allows Filament to check that `m0.out` is semantically valid when it is used as an input to `m1` and that the two uses of the multiplier are scheduled to occur at different times, allowing the compiler to generate correct FSMs to schedule instance reuse. Each invocation only provides inputs for the data ports and elides inputs for the interface ports. During compilation, Filament's compiler automatically infers assignments for the input ports and generates efficient, pipelined FSMs to schedule the invocations (§7.4).

7.2.5 Connections

Filament programs allow ports to be connected and requires that the source is semantically valid for at least as long as the destination. Connections are physically implemented as continuously active wires connecting two ports in the circuit.

```
comp Add<G:1>(source: [G, G+3] 32) -> (dest: [G, G+1] 32) {
  dest = source; }
```

7.2.6 Interfacing with External Components

Filament's `extern` keyword allows the user to provide type-safe wrappers for black box modules by specifying a type signature without a body. Filament's standard library, which provides signatures for components like multipliers and registers, is defined using `extern` components.

Phantom events Phantom events allow Filament to model the behavior of components like adders which are *continuously* active and do not take an explicit enable signal. In the following signature, the event G is a phantom event because there is no corresponding interface port for it in the signature. Section 7.4.4 describes how user-level components can use phantom events.

```
extern comp Add<G: 1>(l: [G, G+1] 32, r: [G, G+1] 32) -> (
  o: [G, G+1] 32))
```

Ordering constraints In order to capture the full expressivity of external components, Filament allows defining ordering constraints between events. For example, combinational components can provide a valid output for more than one cycle if the inputs are provided for multiple cycles. Therefore, a more precise interface of a combinational adder is:

```
comp Add<G: L-G, L: 1>(l: @[G, L] 32, r: [G, L] 32) -> (o: [G, L] 32)
    where L > G
```

The events G and L mark the start and end for the input and output availability intervals. In order to ensure that the interval $[G, L]$ is well-formed, the signature requires $L > G$. The component guarantees that the output is provided for as long as the inputs are provided.

```
A := new Add;
// delay = (G+3)-G = 3
```

```
a0 := A<G, G+3>(x, y);
```

Parametric delays The new signature of adder additionally specifies a *parametric delay* of $L - G$ cycles to signal that the adder may not be reused while it is processing a set of inputs. In order to generate *static pipelines* which have input-independent timing behavior, Filament requires all such expressions to evaluate to a constant value. Like the example, an invocation of `Add` must provide some binding of the form $G = T + i$ and $L = T + k$ such that $k > i$, ensuring that the delay for the corresponding invocation is a compile-time constant $k - i$ and the ordering constraint $L > G$ is satisfied.

The signature of registers in Filament allows them to provide the output for as long as needed, similar to an adder. However, because a register is a state element, it only requires its input for one cycle. Furthermore, the delay signals that the register can accept a new write during the last cycle when the output is available.

```
comp Register<G: L-(G+1), L: 1>(
  @interface[G] go: 1, in: [G, G+1] 32) -> (out: [G+1, L] 32) where L >
      G+1;
```

7.3 Type System

Filament's type system enforces two fundamental restrictions of hardware design:

1. All reads only use *semantically* valid values. A port or wire will always have a value on it. Filament's availability intervals mark when the values are semantically valid.
2. Writes do not conflict. This is a corollary of the property that uses of a resource must not conflict because use of a resource is represented through

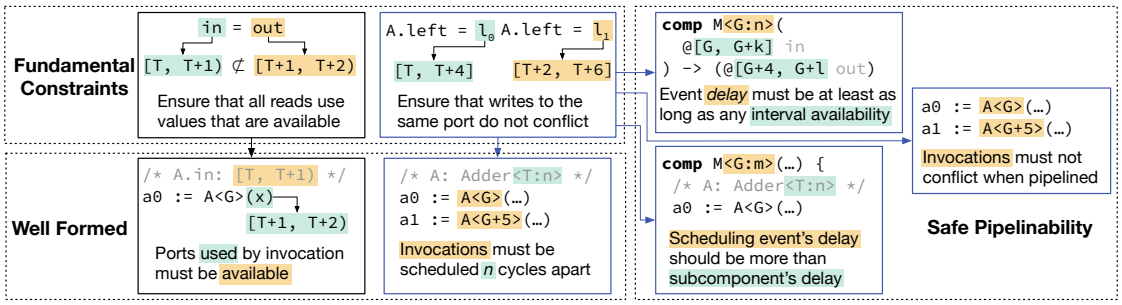


Figure 7.6: Overview of the Filament type system. The fundamental constraints of hardware design imply other constraints. Well-formedness ensures that *one execution* of a component is correct. Safe pipelining ensures that *pipelined executions* of the component are correct.

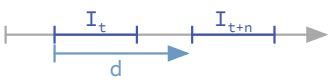
a write.

Filament ensures these properties using two checking phases: well-formedness checking, which ensures that a *single execution* of a component is correct, and *safe pipelining*, which ensures that *pipelined executions* of a component are correct.

7.3.1 Delay Well-Formedness

The delay of an event encapsulates all possible conflicts between parts of the pipeline scheduled using it. Filament requires that the delay of an event is at least as long as each interval that mentions it which ensures that instance reuse does not create conflicts between its input and output ports.

The proof is straightforward: for two invocations at time t and $t + n$ such that $n \geq d$ where d is the delay,



let I_t and I_{t+n} be the availability intervals of the input i. Since we know that the start times of the intervals are at least d cycles apart ($I_{t+n} - I_t \geq d$), and that length of the intervals is bounded by d ($|I_t| \leq d$) we can conclude that they do not overlap.

7.3.2 Well-Formedness

Valid reads In order to ensure this property, Filament needs to make sure that port values are only read when they are semantically valid. Signals are used in two places:

1. *Connections* (§7.2.5) forward a value from one port to another. Filament ensures that the availability of the output port is at least as long as the requirement of the input port.
2. *Invocations* (§7.2.4) schedule the use of a component instance using a set of events. Checking the validity of an invocation boils down to two steps: the requirements of the instance's input ports can be computed by binding the event variables in its signature to the invocation's event. Next, each argument essentially represents a connection between the instance's input and the argument and is checked using the criteria for connections.

```
comp Mult<T:3>(...);
comp main<G:10>() {
  M := new Mult;
  // busy b/w [G, G+3]
  a0 := M<G>(a, b);
  // busy b/w [G+1, G+4]
  a1 := M<G+1>(a0.out, b);
```

Conflict-free If an invocation schedules an instance with delay d using the event G, the instance may not be reused between $[G, G + d)$. This both ensures that the there are no conflicts between input and output ports (§7.3.1) and that none of the subcomponents conflict. The latter property holds because safe pipelining constraints ensure that a valid delay can correctly encapsulate all possible conflicts between subcomponents (§7.3.4). In the example program, the two invocations of `M` overlap causing Filament to reject this program.

7.3.3 Initiation Intervals

Pipelining is an important optimization since it allows a module to process multiple inputs in parallel. For example, a multiplier with a three cycle latency, but an *initiation interval* of one cycle takes three cycles to compute an output but can accept new inputs every cycle. In Filament, the delay of an event corresponds to initiation interval. While hardware designers talk about initiation intervals of a component, Filament generalizes it by allowing a component to have multiple events. In this case, each event specifies the initiation interval of some part of the internal pipeline. Filament ensures that the delay of a module describes a valid initiation interval, defined as follows:

Definition 7.3.1 (Initiation Interval)**.** *Let $P(t)$ be the execution of pipeline P at time t. $P(t_0) \perp P(t_1)$ states that the pipeline executions of P at t_0 and t_1 do not have resource conflicts. Then I is a valid initiation interval of pipeline P if and only if*

$$\forall n \geq 0 \; P(t) \perp P(t + I + n)$$

This definition requires that the pipeline is able to accept new inputs after *any* amount of time after the initiation interval. There might be other delays smaller than the initiation interval which allow the pipeline to accept new inputs in a small window of time before becoming invalid again. This would correspond to the following definition of an initiation interval I:

$$\forall k \neq 0 \; P(t) \perp P(t + k \times I)$$

Filament uses the first definition because delays are also used to check the well-formedness constraints of a component. If we used the second definition, the

well-formedness constraint would require that if an instance is scheduled at time t, it may only be scheduled again at other times $k * t$ which we think is less compositional. Regardless, this is not a fundamental limitation since both definitions can be encoded and enforced.

7.3.4 Safe Pipelining

While well-formedness ensures that one execution of a module is correct, i.e., all reads use valid values and there are no conflicts, safe pipelining must ensure that *pipelined executions* of the component do not create any additional conflicts. Checking that pipelined executions do not conflict is very similar to checking that invocations of the same instance do not conflict. This is because pipelined execution is exactly the same—an instance being reused after a period of time. Filament must show that for an invocation scheduled using event G, another invocation scheduled at any time after $G + d$ (where d is the delay) does not conflict with the first invocation. The following checks are sufficient to prove this.

Triggering Subcomponents Filament requires that when an event is used to invoke a subcomponent, the event's delay must be at least as long as the delay of the subcomponent's event.

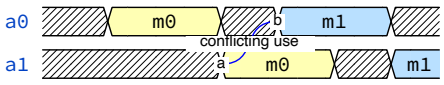
```
comp Mult<G:3>(@interface[G] go: 1, ...)
comp main<T:1>(@interface[T] go: 1, ...) {
  M := new Mult; m0 := M<T+2>(...) }
```

The event $T+2$ is used to schedule the invocation of instance M which has a delay of 3. However, $T + 2$ has a delay of 1, same as T. This is problematic because `main` may trigger every cycle while `M` can only support computations every 3 cycles. Filament therefore rejects this program.

Reusing Instances Previous checks already ensure that: (1) shared invocations do not conflict during one execution of the pipeline, and (2) pipelined execution of an invocation does not conflict with itself. However, we also need to ensure that pipelined invocations of a shared instance do not conflict with each other.

```
comp Mult<G: 3>(...)
comp main<T: 3>(...)
    {
  M := new Mult;
  m0 := M<T+2>(...);
  m1 := M<T+10>(...);
```

The example program will pass all our previous checks but is erroneous: executing the pipeline at time T and $T + 10$ will cause the `m1` from the time T execution to conflict with `m0` from the time $T + 10$ execution. Because Filament's definition of initiation interval allows re-execution at *any time* in the future, we must require that all invocations of a shared instance complete before the pipelined execution begins. The following is sufficient to ensure this: the delay must be greater than the number of cycles between the start of the earliest invocation and the end of the last invocation of a shared instance.




```
comp Dyn<G: ??, L: ??>(..) {
  M := new Mult;
  a0 := M<G>(a, b);
  a1 := M<L>(a0.out, b); }
```

Dynamic Reuse Since Filament components can be parameterized by multiple events, it is possible to invoke an instance using two different events. In the example program, the type-checker would have to prove that the intervals $[G, G + 3)$ and $[L, L + 3)$ do not overlap to enforce conflict freedom. The constraint $L \geq G + 3$ is sufficient to prove this. However, there is no way to statically pipeline this module: the delay of G is *dynamic*, it depends on exactly which cycle L is provided which cannot be known a priori. There is no compile-time constant value that can express the delays for both events. This is because delays describe the timeline for a

single event whereas dynamic modules require relating multiple events. Filament's solution is to disallow ordering constraints between events in user-level components which disallows the example program. External components (§7.2.6) can still use ordering constraints, but such constraints can only be satisfied using the natural order defined on $G + n$ events. This means in a well-typed program:

1. All delays evaluate to compile-time constants.
2. Invocations of a shared component all use the same event.

These constraints allow the compiler to generate efficient, statically timed pipelines from well-typed programs. Extending Filament with safe dynamic pipelines is an avenue for future work.

7.4 Compilation

Figure 7.7 shows an overview of the compilation flow. The primary goal of Filament's compilation pipeline is to transform the abstract schedules of invocations into explicit, pipelined control logic. The compiler first lowers programs into *Low Filament* which is an untyped extension of the Filament language that explicitly uses pipelined finite state machines (FSMs) to coordinate the execution of a module. Next, the compiler translates the program into the Calyx intermediate language [116] which performs generic optimizations and generates circuits.

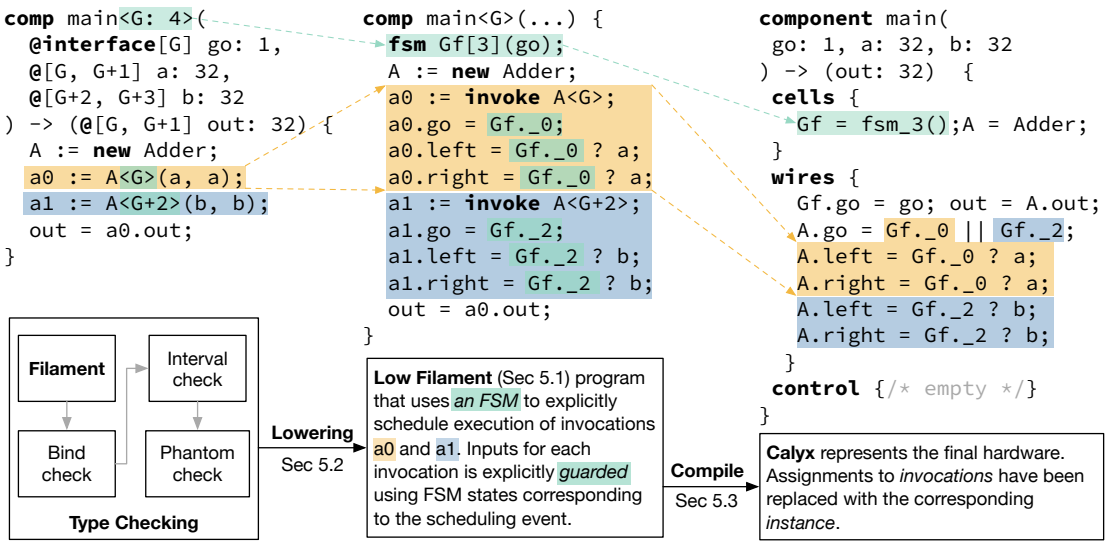


Figure 7.7: Compilation Flow. Filament programs are type checked (§7.3) and lowered to *Low Filament* (§7.4.1) programs. Lowering (§7.4.2) instantiates explicit FSMs to schedule invocation. Finally, Low filament programs are compiled to Calyx [116] which optimizes the design and generates hardware circuits.

7.4.1 Low Filament

Low Filament is an untyped version of Filament that introduces new constructs to explicitly represent the pipelined execution of a module.

Explicit Invocations Low Filament requires all ports corresponding to an invocation to be explicitly assigned. This includes interface ports, which high-level Filament manages implicitly.

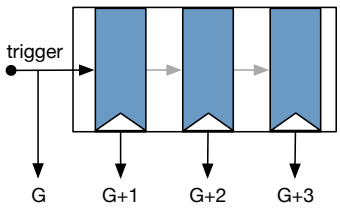
Guarded Assignment Filament uses *guarded assignments* to express multiplexing of signals and correspond directly to guarded assignments in Calyx [116]. The assignment only forwards the value from `out` when the guard is active. Otherwise, the value forwarded to `in` is undefined. Calyx's well-formedness condition requires that only one of the guards is active at

```
in = g1 ? out;
in = g2 ? out;
```

a time for any given source port.

Finite state machines Low Filament also provides the `fsm` construct to explicitly instantiate a pipeline FSM. This defines the FSM `F` with n states and a single input port `trigger` which triggers its execution. This generates a shift-register of size n with ports: `F._0`, …, `F._{n-1}`. If *trigger* is set to 1 at event G, the port `F._i` will become active at event $G + i$.

```
fsm F[n](trigger);
```



7.4.2 Generating Explicit Schedules

The compilation from Filament to Low Filament ensures that all high-level invocations have been compiled into explicit invocations. Figure 7.7 shows the compilation process for a program that uses an adder (`A`) through two invocations (`a0` and `a1`).

FSM Generation The compiler instantiates an FSM for each event parameterizing the module. The example program uses event G to schedule the invocations. The compiler walks over all expressions $G + i$ in the program to compute the number of stages for the pipelined FSM. While the original program does not explicitly mention the event $G + 3$, it is implied by the output port `a1.out` which is active in the interval $[G + 2, G + 3]$. The compiler instantiates the FSM `Gf` with 3 states triggered by the `go` signal. Note that the delay of the FSM *does not* affect the generation of the FSM.

Triggering Interface ports The compiler then lowers the invocations by generating explicit assignments to the adder's interface port `go`. The first invocation, scheduled at G, uses the port `Gf._0` to trigger the invocation while the second invocation, scheduled at $G + 2$, uses the port `Gf._2`.

Guard Synthesis In order to ensure that assignments from the two invocations to the data ports `left` and `right` do not conflict, the compiler synthesizes guards for the assignments. If the input port of an invocation require inputs during the interval $[G + s, G + e]$, the compiler generates the guard `Gf._s || ... || Gf._e` for the guard. Since the program is well-typed, the guard expressions for each invocation are guaranteed to not conflict (§7.3).

7.4.3 Lowering to Calyx

Low Filament is intentionally designed to be close to Calyx, so compilation is straightforward. For each FSM size n, we generate a Calyx component and instantiate it for the corresponding Filament component. The FSM is simply a sequence of registers connected together. Since assignments to all ports are explicit in Low Filament, we can simply compile the invocations by replacing them with the corresponding instance name. In the example program, assignments to both `a0.left` and `a1.left` are compiled to assignments to `A.left`. Since Filament guarantees that the generated guards are disjoint, we can be sure that Calyx will generate correct FSMs.

7.4.4 Optimizing Continuous Pipelines

Continuous pipelines do not make use of a signal to indicate when their inputs are valid and instead, they continuously process inputs. We can express such pipelines in Filament using *phantom events* (§7.2.6). Phantom events do not have a corresponding interface port and therefore cannot be used to trigger invocations. Filament ensures that a phantom event is used correctly through its phantom check analysis which ensures:

Definition 7.4.1 (Phantom Check)**.** *A phantom event G is used correctly if:*

1. *It is not used to share any instances.*
2. *It is only used to invoke subcomponents that use phantom events.*

First, resource sharing is disallowed because any pipeline that shares an instance must use some signal to trigger an internal FSM and track which use of the instance is currently active. Second, a phantom event is only available at the type-level and cannot be reified since there is no interface port. Therefore, only components that use phantom events can be invoked with a phantom event.

Filament defines two state primitives: a *register* and a *delay* component.

```
comp Register<G: L-(G+1), L: 1>(
  @interface[G] en: 1, in: [G, G+1] 32
) -> (out: [G+1, L] 32) where L > G+1;
```

```
comp Delay<G: 1>(
  in: [G, G+1] 32
) -> (out: [G+1, G+2] 32);
```

As the type signatures denote, the difference is that a register can hold onto a value for an arbitrary amount of time while a delay can only hold onto a value for a single cycle. The `Delay` component accepts inputs every cycle and can therefore provide the output for one cycle. In contrast, the register can use the `en` signal to hold onto a value for an arbitrary amount of time.

$$
\begin{aligned}
&x \in \mathit{vars} \quad t \in \mathit{events} \quad p, q \in \mathit{ports} \\
&T ::= t \mid T + n \quad \pi ::= [T_1, T_2] \\
&M ::= \\
&\quad \textbf{def}\, C\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)\{c\} \\
&c ::= c_1 \bullet c_2 \mid p_d = p_s \mid x := \textbf{new}\ C \\
&\quad \mid x := \textbf{inv}\ x\langle T\rangle(p_1, \ldots, p_j) \\
&\tau ::= \forall\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)
\end{aligned}
$$

(a) Abstract syntax

$$
\begin{aligned}
&[\![c]\!] : \mathcal{L} \to \mathcal{L} \qquad \mathcal{L} : \mathcal{T} \to \mathcal{R} \times \mathcal{W} \\
&[\![p_d = p_s]\!]\,(L) = \mathrm{map}(\lambda(R, W).\, \mathrm{if}\, p_s \in W \\
&\qquad \mathrm{then}\,(R\{p_s/p_d\}, W)\,\mathrm{else}\,(R, W), L) \\
&[\![c_1 \bullet c_2]\!]\,(L) = [\![c_1]\!]\,(L) \cup [\![c_2]\!]\,(L)
\end{aligned}
$$

(b) Log-transformer semantics

$$
\frac{\begin{array}{c}\Delta, \Lambda_1, \Gamma \vdash c_1 \dashv \Lambda_1', \Gamma_1 \\ \Delta, \Lambda_2, \Gamma \vdash c_2 \dashv \Lambda_2', \Gamma_2\end{array}}{\Delta, \Lambda_1 * \Lambda_2, \Gamma \vdash c_1 \bullet c_2 \dashv \Lambda_1' * \Lambda_2', \Gamma_1 \cup \Gamma_2}\ \textsc{CheckComp}
$$

(c) Composition judgement

Figure 7.8: Formal semantics of Filament where command is defined as a *log-transformer*. Typing judgements track the active timeline of an instance and ensure they are used in a disjoint manner.

Compilation The compiler does not instantiate FSMs or synthesize guards for invocations triggered using phantom events. Since Phantom Check ensures that all subcomponents themselves do not have an interface port, the compiler does not have to generate assignments for them. Filament generated code for continuous pipelines matches expert-written code.

7.5 Formalization

Figure 7.8a presents a simplified syntax for Filament: all components can be parameterized using exactly one event and cannot specify any ordering constraints between events. Since Filament disallows any form of event interaction in user-level components, multi-event user-level components are not fundamentally more expressive. Multi-event external components are more expressive but not supported in

our formalism. A Filament program ($\mathcal{P}$) is a sequence of components which define a signature and a body in terms of commands: composition, connection, instantiation, and invocation.

7.5.1 Semantics

Figure 7.8b presents Filament's semantics which is defined as functions over logs ($\mathcal{L}$). A log maps events ($\mathcal{T}$) to a set of ports that are read from ($\mathcal{R}$) and a multiset of ports that are written to ($\mathcal{W}$). Intuitively, a log captures all the reads and writes performed during every cycle of a component's execution. We track the multiset of writes to capture conflicts—if there are multiple writes to the same port in the same cycle, then the program has a resource conflict.

Concrete logs are generated by the semantics of component definitions while commands simply transform them. For example, a port connection forwards the value from the source port p_s to the destination port p_d. We model this by substituting all occurrences of p_d to p_s in the read set $\mathcal{R}$ when p_s is defined in the write-set $\mathcal{W}$ and mapping it over all defined events in the log. Composition reflects the parallel nature of hardware—it simply unions the two logs together. Write conflicts can appear due to composition. The semantics of a program is the log generated by executing a distinguished main component with the empty log. We formalize the well-formedness (§7.3.2) and safe pipelining (§7.3.4) constraints of the type system using this semantics.

Definition 7.5.1 (Well-Formedness)**.** *A component M is well-formed if and only if its log is well-formed. A log L is well-formed if and only if, for each event:*

- *There are no conflicting writes: $W_s = W$ where W_s is the deduplicated set of*

writes.

- *Reads are a subset of writes:* $R \subseteq W_s$

Definition 7.5.2 (Safe Pipelining)**.** *If a component* M *has an event* T *with delay* d*, and* $[\![M]\!]_G$ *represents its log where* T *is replaced with the event* G*, then* M *is safely pipelined if and only if all logs* L_n *are well-formed:* $L_n = \forall n \geq d \ [\![M]\!]_T \cup [\![M]\!]_{T+n}$

7.5.2 Type System

Filament implements a type system inspired by separation logic [133] to enforce the well-formedness and safe pipelining constraints. Our presentation focuses on the specific typing judgement that ensures that there are no conflicting uses of an instance. Appendix B.3 provides the full type system. At a high level, our typing judgement for composition (Figure 7.8c) mirrors the parallel composition rule used in concurrent separation logic [23]—the two commands are checked under two disjoint resource contexts. Our insight is adapting the definition of separating split to timelines of instances and ensuring that instance reuse does not conflict. The typing judgements have the form: $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$. Γ is the standard type environment, Δ tracks each event's delay, and Λ is the *resource context*.

Resource contexts and separating split Λ is the resource context and tracks the availability of each instance and port in the form of an interval (π). After instantiation, each instance is available in the interval $[0, \infty)$. The invocation rule (not shown) checks that, for an instance's event with a delay d, the instance is available in the interval $[G, G+d)$ where G is the scheduling event. The composition rule (Figure 7.8c) splits the resource context before checking the two commands:

$$\Lambda = \Lambda_1 * \Lambda_2 \text{ iff } \forall (x : \pi) \in \Lambda \Rightarrow$$
$$\exists \pi_1, \pi_2 . (x : \pi_1) \in \Lambda_1 \wedge (x : \pi_2) \in \Lambda_2 \wedge \pi_1 \cap \pi_2 = \emptyset \wedge \pi_1 \cup \pi_2 = \pi$$

A valid split is one where the resulting contexts have disjoint intervals for each instance and the union of the intervals is the original interval. By using this definition of split, Filament ensures that invocations reuse instances in a non-conflicting manner. Appendix B.4 presents the remaining type judgements that encode constraints to enforce well-formedness and safe pipelining and proves the following type soundness theorem:

Theorem 7.5.3. *If* $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$ *then* $[\![c]\!]$ *is well-formed (Definition 7.5.1).*

7.6 Evaluation

We evaluate Filament's ability to efficiently express a number of accelerator designs and to express the interfaces generated by state-of-the-art accelerator generators. Our evaluation answers the following questions:

1. Can Filament express the interfaces generated by state-of-the-art accelerator generators and integrate with existing tools?
2. Can Filament be used to generate efficient accelerators?

Implementation The Filament compiler is implemented using a pass-based compiler in 5426 lines of Rust, 341 lines of Verilog for the standard library primitives, and the latest version of the Calyx compiler [116] to generate Verilog. All benchmarks compile in under a second.

7.6.1 Expressivity Evaluation

To demonstrate the expressivity of Filament, we focus on giving type signatures to designs generated by Aetherling [52].

Aetherling's space-time types Aetherling [52] is a functional, dataflow DSL that generates statically-scheduled, streaming accelerators for image processing tasks. Aetherling's "space-time" types enable users to express the shape of the data stream as a sequence of valid and invalid signals. For example, the type `TSeq 1 1` denotes that there will be a stream with one valid element followed by one invalid element. Nesting these types allows users to express more complex shapes: `TSeq 3 0 (TSeq 1 1)` denotes that there will be three valid elements, with no invalid values, each of which has a shape described by `TSeq 1 1`. In our case study, we import 14 designs implementing two kernels: conv2d and sharpen. Aetherling's evaluation studies 7 design points for each kernel with different resource-throughput trade-offs. Filament can express the interface types for all designs and, in the process, finds several bugs in the generated interfaces.

Cycle accurate harness We implemented a generic, *cycle-accurate* harness to test Filament programs. At a high-level it:

1. Provides the inputs for exactly the cycles specified in a component's interface.
2. Pipelines the execution of the component using event delays.
3. Captures the value of output ports in the intervals provided in the signature.

Throughput	Reported	Actual
16	7	7
8	6	6
4	6	6
2	6	6
1	7	7
1/3	**10**	12
1/9	**16**	21

(a) Reported latencies for conv2d

Throughput	Reported	Actual
16	7	7
8	7	7
4	7	7
2	7	7
1	8	8
1/3	**11**	13
1/9	**17**	20

(b) Reported latencies for sharpen

Table 7.1: Latencies of Aetherling Designs. Highlighted latencies are reported incorrectly by Aetherling.

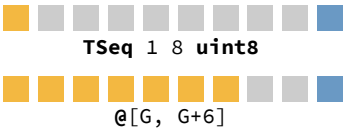
The harness extracts the availability intervals and the event delays using a simple command-line flag provided to the compiler and executes the design using the cocotb Python library. The design of this generic harness is reliant on a Filament-like system to document the timing behavior of modules; without Filament, a user would have to manually extract this information from the Verilog code.

Methodology We compile each Aetherling design to Verilog and use Aetherling's command line interface to extract the design's latency information. Each benchmark has five fully-utilized designs, which can accept new inputs every cycle, and two underutilized designs which produce 1/3 and 1/9 pixels per clock cycle and accept new inputs every 3 and 9 cycles. We give each design a type signature and validate its outputs. For designs with mismatched outputs, we change the latency till we get the right answer.

Latency Table 7.1 reports the latencies as provided by Aetherling's command line interface and those that we found to generate correct outputs with Filament's cycle accurate test harness. Of the 14 designs, Aetherling reports incorrect latencies for 5 designs.

Underutilized designs Aetherling explores the utility of *underutilized* designs which produce less than one pixel per clock cycle. Aetherling's compiler optimizes such designs by sharing compute resources. An Aetherling design that produces 1/9 pixels per clock has the type `TSeq 1 8 uint8` which states that there will be 1 valid datum followed by 8 invalid ones. The type indicates that the design generated by Aetherling should only use its input in the first cycle since the data provided in the next cycles is invalid. However, this interface is incorrect.

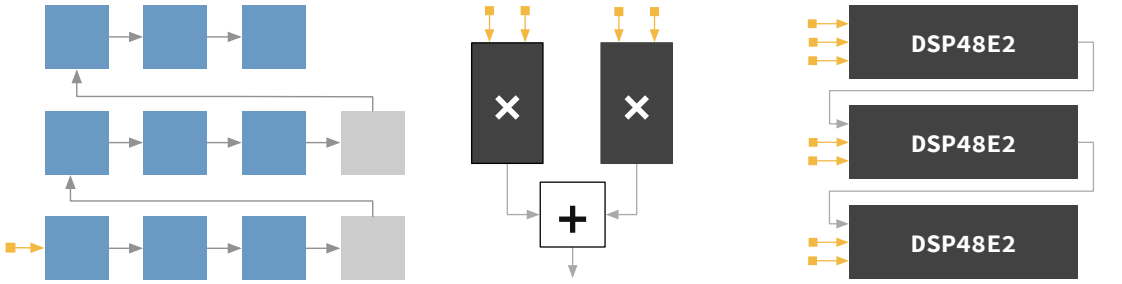
```
comp Conv2d<G: 9>(
  @[G, G+6] I: 8,
) -> (O: [G+21, G+22] 8);
```



The Filament type, which reflects the actual interface needed to correctly execute the module, requires the design to hold its input signal for six cycles, i.e., the data element must be valid for six cycles instead of just one; the Aetherling implementation breaks its own interface. The Aetherling test harness does not catch this bug because it always asserts all inputs for 9 cycles. In contrast, Filament's test harness only asserts the input signal for as long as the corresponding availability interval specifies. Finally, the delay for the phantom event G encodes the fact the design can process a new input every 9 cycles. This illustrates the subtlety of specifying time-sensitive interfaces which accurately describe signal availability and pipelining.

7.6.2 Accelerator Design with Filament

We study Filament's efficacy in generating efficient designs and reusing components generated from other languages by implementing a two-dimensional convolution in Filament. We build two Filament-based designs and compare them to the



(a) Line buffer in the `Stencil` component

(b) Base design uses pipelined 3-cycle multiplier

(c) Reticle-based design uses DSP cascade

Figure 7.9: Components used in the design of Filament-based conv2d convolution. The stencil component provides the last three inputs and is either connected to the naive multiplier or a Reticle-generated DSP cascade.

Aetherling-generated conv design.

Architecture Our implementation is directly inspired by the structure of the Aetherling implementation of conv2d that outputs 1 pixel per clock cycle. The design uses a 3×3 filter over a 4×4 matrix. The `Stencil` module (Figure 7.9a) implements a line buffer to save the last 11 values and outputs 9 values corresponding to the filter start index. The `Conv2d` kernel takes 9 values as inputs and produces an output corresponding to the result of the convolution.

```
comp Prev[SAFE]<G: 1>(
  @interface[G] en: 1,
  in: [G, G+1] 32,
) -> (out: [G, G+1] 32);
```

Stream primitives in Filament To implement line buffers, we implement a new `Prev` component which outputs the last value stored in it.[3] The Verilog implementation of `Prev` is simply a register but Filament gives it a different type signature—it allows access to the output in the same cycle when the input is provided which corresponds to reading the previous value in the register. The

[3] `prev` is a common operator in dataflow and functional reactive languages.

component uses a compile-time parameter `SAFE` to indicate whether the first read produces an undefined value. We also define a `ContPrev` component which is similar to a `Prev` component but uses a phantom event and can therefore be used in continuous pipelines (§7.4.4). The stencil component (Figure 7.9a) is implemented as a sequence of `Prev` components.

Design 1: Pipelined multipliers The base `Conv2D` kernel uses fully pipelined multipliers with a three cycle latency and combinational adders. The multipliers do not have any associated Verilog implementation—they are implemented using Xilinx's LogiCORE multiplier generator. However, Filament makes it easy to interface with them by providing a type-safe extern wrapper (§7.2.6).

```
comp Tdot<G: 1>(
  clk: 1, reset: 1,
  a0: [G, G+1] 8,
  b0: [G, G+1] 8,
  a1: [G+1, G+2] 8,
  b1: [G+1, G+2] 8,
  a2: [G+2, G+3] 8,
  b2: [G+2, G+3] 8,
  c: [G+2, G+3] 8,
) -> (y: [G+5, G+6] 8)
```

Design 2: Integrating with Reticle Our second design uses a dot-product unit generated using Reticle [154], a low-level language for programming FPGAs. Figure 7.9c shows the architecture Reticle generates to make use of *DSP cascading* which efficiently utilizes resources present on an FPGA. DSP cascading explicitly instantiates low-level FPGA primitives and connects them together to implement the computation: $y = c + \Sigma_{i=0}^{3} a_i \times b_i$. Unlike standard compilation flows which rely on the synthesis tool to infer DSP usage from *behavioral* descriptions, Reticle generates *structural* descriptions that predictably map onto DSPs. We provide a type signature for the Reticle design which indicates that the inputs must be pro-

Name	LUTs	DSPs	Registers	Freq. (MHz)
Aetherling	104	10	78	769.2
Filament	128	**9**	**11**	**833.3**
Filament Reticle	**14**	**9**	20	645.1

Table 7.2: Resource usage and frequency of conv2d designs. Best values highlighted.

vided in a staggered manner. Note that this is not implementation details leaking through—a DSP cascade that starts a new computation every cycle needs to either register all its inputs or provide them in a staggered manner.

Evaluation methodology We validate the correctness of all the designs using our timing-accurate test harness and compare the area and latency of the designs. For each design, we increase the target frequency till we reach worst negative slack of less than $0.1ns$ and synthesize them using Vivado v2020.2. Each design has a throughput of 1 pixel per clock cycle.

Summary Table 7.2 shows the results of the comparison: the Filament design can be synthesized at a higher frequency and uses fewer resources than the Aetherling design. This is because Filament can safely and directly use low-level implementation modules which can be directly compiled into a safe and efficient design. In contrast, the Aetherling compiler has to generate extra logic when bridging the gap between its high-level language and low-level circuits. The Reticle-based design uses an order of magnitude fewer logic resources than the base Filament design or the Aetherling design. This is because unlike Aetherling, Reticle generates low-level *structural* Verilog which can predictably map onto DSP resources. This demonstrates the utility of Filament as both an integration and design language—designs in Filament can use low-level hardware modules safely and compose complex modules generated from other languages. It also reveals another use case for

Filament: instead of directly generating Verilog, Aetherling-like languages can generate Filament programs and enable performance engineers to optimize the designs further and remove abstraction overheads.

CHAPTER 8

CORRECT AND COMPOSITIONAL HARDWARE GENERATORS

Filament provides a new foundation for reasoning about modular and efficient composition of hardware designs. However, it is limited to describing the composition of individual circuits. In realistic designs, users want to specify an entire *family of circuits*. For example, when implementing a dot-product module, the user can decide to allocate one multiplier and use it over multiple cycles to perform the computation, leading to a resource-efficient but high-latency design, or allocate multiple multipliers to compute the result faster but yield a more resource-expensive implementation. These circuits are all related to each other because they implement the same computation—a dot-product—but express different performance characteristics.

Existing tools provide three different mechanisms to express such families of circuits:

- **Parameterization**: hardware description languages (HDLs) like SystemVerilog [11] and Bluespec [119] provide compile-time constructs like for-loops and conditionals [11, §27] which, like software metaprogramming [49, 54, 134], can generate code at compile time.
- **Metaprogramming**: Embedded hardware description languages (eHDLs) like Chisel [13] and PyMTL [101] are embedded in software languages like Scala and Python. In eHDLs, executing the host language program generates the circuit allowing users to utilize host language constructs for metaprogramming.
- **Custom Tools**: Depending on the complexity of the circuit and the optimization goals, users might develop customized tools like FloPoCo [46]. Similar

to the Systolic Array generator in Calyx (§6.2), such tools are implemented as arbitrary software programs and intended to generate specific hardware blocks.

We term such tools *hardware generators*. Hardware generators are useful for producing specific hardware blocks which are utilized in the context of a specific, general-purpose design. Hardware generators are powerful because they are *reusable* and *flexible*. Flexibility means that they can be used to explore performance trade-offs for a given computation. The burden of correctness is also moved away from the core design. Once a designer of a hardware generator proves the implementation correct, the main design can use any generated design without worrying about its correctness. Flexibility enables reuse: different designs have different requirements and might want to trade off performance for resource consumption.

However, this flexibility of hardware generators complicates their integration into large-scale designs. Users have two choices for integration:

Link specific instances Because hardware generators express an entire design space of circuits, a designer might instantiate a particular point in the design space and integrate it. This is straightforward because any specific design point will have particular interfaces and timing behaviors which can be easily hard-coded into the surrounding glue code. However, this loses out on the power of generators: instead of exploring trade-offs in the context of a complete implementation, the user must commit to a specific design point and then build the rest of the circuit around it. In order to explore a new design point, the surrounding circuit must be reconstructed. This is by far the most common approach to integration.

Link generators The other choice is to link together generators. There are two challenges to this. First, it is not clear what the interface of a generator is: the latency of a dot-product implementation from a generator is dependent on the number of multipliers available. However, existing HDLs do not provide an easy way to capture this information. Second, integrating generators amounts to composing entire design spaces of circuits which exponentially increases the complexity and complicates reasoning about correctness.

This chapter demonstrates an approach that allows us to retain the flexibility of generators while automatically proving the correctness of their composition. The key insight is that *parameters can influence timing behaviors* and therefore must be modeled in the interface of generators. We extend Filament's type system to express and reason about parametric designs. In developing this extension, we exploit the symmetry between universal and existential types to formally characterize *bottom-up parameterization*. Bottom-up parameterization allows us to provide interfaces to modules generated by custom tools and allows Filament to integrate them flexibly. In doing so, we enable Filament to become the ultimate integration tool capable of integrating not just individual hardware modules but entire ecosystems of tools.

8.1 Motivating Example

We motivate the use of parameterized designs and generators by implementing a pointwise multiplication module that processes two, 4-element vectors.

8.1.1 Initial Implementation

We start by implementing a specific implementation of the dot-product module. The computational specification of pointwise multiplication is straightforward:

```
let a, b, o: int[4];
for (let i = 0 .. 4) { o[i] = a[i] * b[i] }
```

When implementing this computation in hardware, we need to make several decisions. First, how many multipliers should be allocated; allocating more multipliers will allow for more parallel computation and reduce its latency but require more resources. Second, how can we *safely reuse* the allocated multipliers over time.

A given multiplier will have a latency—the number of cycles it takes to produce an output—and an *initiation interval*—number of cycles before it can accept new inputs. A pipelined multiplier can accept new inputs before it is finished processing the previous set of inputs. Filament allows us to specify and check for the second set of constraints:

```
comp Mul<'G:1>(
  l: ['G,'G+1] 32, r: ['G,'G+1] 32) -> (out: ['G+3,'G+4] 32);
```

The latency of the module is encoded using the output `out`'s *availability interval* and event `G`'s delay encodes the initiation interval (§7.2).

Next, we allocate two multipliers to process the inputs. We extend Filament ports to be able to accept and produce array values as arguments (§8.2.3)

```
comp PointMul<'G:2>(a[4]: ['G,'G+2] 32, b[4]: ['G,'G+2] 32) -> (
  o[4]: ['G+4, 'G+5] 32
) {
  M0, M1 := new Mul;
  // First set of computations
  o0 := M0<'G>(a[0], b[0]);
  o1 := M1<'G>(a[1], b[1]);
  d0 := new Delay<'G+3>(o0.out);
  d1 := new Delay<'G+3>(o1.out);
  // Second set of computations
```

```
  o2 := M0<'G+1>(a[2], b[2]);
  o3 := M1<'G+1>(a[3], b[3]);
  // All output values appear at 'G+4
  o[0] = o0; o[1] = o1; o[2] = o2; o[3] = o3;
}
```

In order to process four elements using two multipliers, the module runs two computations in parallel at a time. The first two computations occur over the first two indices of the arrays, start at `'G`, and produce outputs available at `['G+3,G+4]`. Because the multiplier is fully-pipelined, the next computations can be started at `'G+1` and produce the output at `['G+4,'G+5]`. In order to ensure all outputs are available in the same cycle, we use a `Delay` component to delay the outputs from the first computations by one cycle. The total latency of the module is four cycles. Because the multipliers are reused in the component, the initiation interval (II) of the component limits to accepting new inputs every other cycle (II=2).

The latency and initiation intervals are both a functions the number of multipliers: had we allocated four multipliers, the component could accept new inputs every cycle (II=1) and have a latency of three cycles. On the other hand, allocating only one multiplier would result in an implementation with II=4 and latency of seven. Note, however, that these implementations express a resource-performance trade-off: allocating more multipliers increases the resource requirements of the component and different users might want to pick different points in the design space. Pre-committing to one implementation limits the possibility of exploring a *design space* of implementations. Instead, we would like to design a component that can express all of these trade-offs in one implementation.

8.1.2 Parameterized Design

Parameterization allows users to implement *generators*, each of which expresses a family of circuits. The most common approach in HDLs is to provide metaprogramming capabilities: language constructs which can be used to generate code at compile time. For example, SystemVerilog [11] provides compile-time `for` loops which can generate code based on parameter values:

```
parameter N = 4;
input l[N-1:0], r[N-1:0];
output o[N-1:0];
for (i = 0; i < N; i++) begin
  OrGate o(l[i], r[i], o[i]);
end
```

Which would generate the following circuit:

```
OrGate o0(l[0], r[0], o[0]);
OrGate o1(l[1], r[1], o[1]);
OrGate o2(l[2], r[2], o[2]);
OrGate o3(l[3], r[3], o[3]);
```

Parameterized designs can enable powerful design-space exploration capabilties in both complex processor designs [168], domain-specific architectures [61], and custom accelerators [138]. We extend Filament to support parameterized design and attempt to build a pointwise multiplication unit that takes the number of multipliers as a parameter:

```
comp PointMul[N]<'G:..>(a[4]: .., b[4]: ..) -> (o[4]: ..) {
  for (let i = 0 .. N) {
    M := new Mul;
    for (let j = 0 .. 4/N) {
      m := M<'G>(a[i+N*j], b[i+N*j]);
      o[i+M*j] = m.out
}}i}
```

We define the parameter `N` for the `PointMul` component and generate `N` multipliers which each process $N \div 4$ elements of the array. We leave out the delay of the even and the availability intervals for ports. Testing this design with $N = 4$, we will

get the right results but when setting $N = 2$, we see the problem: all invocations (§7.2.4) of the multiplier are scheduled at the same time; we are attempting to send in all the $N \div 4$ inputs at the same time! Unsurprisingly, parameterization makes testing exponentially harder: each implementation has a large space of design space of point each of which might express different reuse policies and timing behaviors. To fix the implementation, we have to change when the invocations are scheduled so that they respect the initiation interval of the multiplier:

```
for (let j = 0 .. 4/N) {
  m := M<'G+i>(a[i+N*j], b[i+N*j]);
  d := new Delay<'G+3+j>(m.out);
  o[i+M*j] = m.out
}
```

We also delay the outputs from the multiplier so that they arrive at the same time. Given this, we can state the interface for the module:

```
comp PointMul[N]<'G:4/N>(
  a[4]: ['G,'G+4/N] 32,
  b[4]: ['G,'G+4/N] 32
) -> (
  o[4]: ['G+2+4/N,'G+3+4/N] 32
) where 4 % N == 0
```

The latency of the module is $L_{Mul} + \frac{4}{N} + 1$ where L_{Mul} is the latency of the multiplier (three in this case). The II of the module is limited by the reused multiplier: each of the N multipliers are reused over $\frac{4}{N}$ cycles and therefore the whole module must wait for that many cycles before accepting new inputs. Finally, we also require that N perfectly divides four to ensure that the inputs can be processed evenly.

Discussion The example demonstrates the challenges of parameterized design. (1) testing is insufficient to establish the correctness of a parameterized implementation, (2) reusing pipelined resources becomes increasingly challenging, and (3) capturing the interfaces of parameterized designs requires tracking the influence of

parameters on timing behavior of the module. We delve on the importance of (3) in the next section that demonstrate that correct and efficient parameterized design fundamentally needs to address the influence of parameters on timing behaviors.

8.1.3 Integrating with External Generators

As our final extension, we would like our pointwise multiplication kernel to operate over floating-point values instead of integers. We could choose to implement our own floating-point multiplier but efficient design requires picking between many possible implementation choices [60]. Instead, we can use floating-point core generator like FloPoCo [46].

FloPoCo takes the bitwidths of inputs and outputs as well as *target frequency* and *target device* parameters and generates synthesizable VHDL [79] code for the computation:

```
flopoco FPMult wE=8 wF=23 frequency=300 target=Virtex5
```

This command above will generate a 32-bit floating-point multiplier with 8 exponent bits (`wE=8`) and 8-mantissa bits (`wF=23`). Unlike an HDL generator, FloPoCo is implemented as a tool in C++ and uses a customized internal representation to automatically pipeline and optimize a floating-point operation for a target device and frequency. However, different target frequencies and devices require different amounts of pipelining: a target frequency of 100MHz might require one pipeline stage while a frequency of 500MHz might require eight. Once again: we have the choice of generating a specific instance and gluing it to the rest of the design, thereby limiting our ability to explore new designs, or integrating all possible multiplier circuit that can be generated.

We can frame the latter integration as connecting a parameterized design whose implementation is decided upon by the FloPoCo tool. Given this, we can write the following interface:

```
comp FPMult[wE, wF]<G:??>(l: [G, G+1] W, r: [G, G+1] W) -> (
  out: [G+??, G+??] W
) where W = wE + wF + 1
```

To complete the interface, we need to specify the initiation interval and the latency of the generated circuit. FloPoCo always generates fully pipelined designs [46], which means the delay for event G is one. However, the latency is a function of the input bitwidth as well as the target frequency and target board:

$$L = f(wE, wF, freq, board)$$

Worse still, this function has no closed-form representation: it is entirely decided upon by the FloPoCo's internal heuristics and might change from version to version!

This is an instance of a general problem: certain components have complex dependencies between parameters and timing behaviors that cannot be easily express or captured at compile-time. For example, a component might switch between a combinational multiplier or a pipelined one based on a parameter value. Similarly, when using open-source libraries like HardFloat [72] which implement purely combinational designs and expect users to add registers and optimize them at synthesis time using register retiming tools [98]. In all these cases, the component's latency does not have an expressible relationship between input parameters and timing behaviors.

Output parameters Input parameters correspond to *universally-quantified* types: they parameterize a module using abstract type variables. Existential types are the

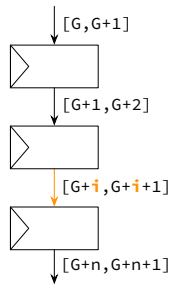
dual of universally-quantified types and are used to hide implementation details while forcing clients to conform to the right interface. We exploit this duality and define *output parameters* as the existential dual to input parameters. Parafil uses output parameters to specify the interfaces of modules generated by external tools:

```
comp FPMult[wE, wF]<G:1>(l: [G, G+1] W, r: [G, G+1] W) -> (
  out: [G+L, G+L] W
) with {
  some L
} where W = wE + wF + 1
```

The new interface uses the existentially-quantified parameter `L` to express the latency of the module. We to not add parameters for the target frequency or target board; they are controlled externally by the user. Because the value of this parameter is not known, the client code must be abstract with respect to it:

```
// Perform a*b+c where a, b, c: ['G, 'G+1] 32
M := new FPMul[23, 8];
A := new Add[32];
mul := M<'G>(a, b);
delay := new Register<'G,'G+M::L>(c); // delay c by L cycles
add := Add<'G+M::L>(mul.out, delay.out);
```

The above implementation accesses latency of the multiplier using the syntax `M::L`. The multiplication occurs in the first cycle. However, since the output from the multiplier L cycles later, the input `c` is delayed using a register. The addition is performed in cycle `L` and the output is available in the same cycle. Parafil's type checker will ensure that the implementation is correct *no matter* what the value of `L` is. This means that the design can be proved correct once and used with any implementation generated by FloPoCo, enabling confident design space exploration.



(a) A shift register with availability intervals for each wire.

```
module Shift #(parameter
    N = 0) (
  input clk, input
      [31:0] in,
  output [31:0] out);
reg [31:0] r[N-1:0];
generate for (i=0; i < N
    ; i++)
  always @(posedge clk)
    if(i == 0) r[0] <=
        in;
    else sr[i+1] <= r[i
        ];
endgenerate
assign out = sr[N];
endmodule
```

(b) Parameterized shift register in Verilog.

```
comp Shift[N]<'G:1>(
  in: [G, G+1] 32
) -> (out: [G+N, G+N+1] 32)
    where N > 0 {
  bundle w[N+1]: for<i> [G+i,
      G+i+1] 32;
  w[0] = in;
  for k in 0..N {
    R := new Reg[32]; r := R
        <'G+k>(w[i]);
    w[i+1] = r.out;
  }
  out = w[N];
}
```

(c) parameterized shift register in Parafil.

Figure 8.1: Shift register implementations in Verilog and Parafil. Parafil implementation provides a timeline type to each value of the bundle which allows it enforce Filament's type safety guarantee.

8.1.4 Summary

Parameteric design and integration is a key component in the development of reusable components in a hardware ecosystem. It enables implementations to express trade-offs that are key to making components reusable in different contexts. However, a key challenge is correct integration without the loss of flexibility: parameters fundamentally influence the timing behavior of modules and require users to reason about them implicitly. Parafil explicates that relationship and enables correct composition. It then goes a step further and enables us to express the interfaces of modules generated from external tools like FloPoCo and supercharges design space exploration by ensuring correct integration of all possible design points.

8.2 The Parafil Language

Filament's type system reasons about the correctness of particular instances. Parafil extends Filament with several new operators to support parameterized design and extends the type system to be able to statically reason about correctness of *all possible instances*.

8.2.1 Parameters

Parafil adds parameter expressions to user-level Filament components. Any location that previously contained a concrete number can now instead use a parameter expression including availability intervals and delays. This allows Parafil to express parameter-dependent timing, pipelining, and resource reuse. Like Filament, Parafil is limited to expressing design with input-independent timing behavior.

8.2.2 Parametric Signatures

A shift register (Figure 8.1) is a sequence of registers that delay a signal by N cycles where N is an input parameter. The Parafil interface (Figure 8.1c) expresses this as follows:

```
comp Shift[N]<'G:1>(in: ['G, 'G+1] 32) -> (
  out: ['G+N, 'G+N+1] 32) where N > 0 { ... }
```

The signature introduces the parameter `N` and uses it in the availability interval for the `out` port. It also describes the timing behavior of the module precisely: the shift register delays the input signal by N cycles and can process new inputs every cycle (since `'G:1` specifies the delay of `'G` as 1). In contrast, the signature in the Verilog implementation (Figure 8.1b) only captures the bitwidths of the inputs and outputs. The Parafil signature also uses a `where` clause to ensure that $N > 0$. This

constraint is checked at compile-time: the module that instantiates `Shift` must statically prove that the argument, which can itself be a parameter expression, is greater than 0. The implementation of `Shift` assumes that this fact is true and uses it to discharge its own proofs.

8.2.3 Bundles

Bundles are multi-dimensional arrays where the availability interval a particular index *depends* upon the index.

```
bundle w[N+1]: for<i> ['G+i, 'G+i+1] 32;
```

The bundle `w` has $N+1$ elements and the i^{th} element has the availability $[G+i, G+i+1)$; `w[0]` has availability $[G, G+1)$, `w[2]` has availability $[G+2, G+3)$ and so on. Availability intervals for bundles are therefore a form of dependent types.

Bundles can be accessed using array-access syntax (`w[0]`) to get a particular index, or range-access (`w[4..N]`) to get a slice. Both syntaxes can be used on the left side of an assignment and at use locations.

For our shift register implementation, we use a bundle to track the availability of the output from each register. Figure 8.1a visualizes this: the input is available in the first cycle, the next wire holds a value in the second cycle, and so on. Bundles also allow programs to forward-declare values and aid resource reuse in parametric programs (§8.2.5).

8.2.4 Compile-time Constructs

Parafil adds several compile-time constructs to Filament.

Loops. The implementation of the shift register instantiates N registers using a `for` loop:

```
for i in 0..N {
  R := new Reg[32]; r := R<'G+i>(w[i]);
  w[i+1] = r.out }
```

The loop body instantiates and invokes a register at time `G+i`, which is the i^{th} cycle of the shift register's execution, uses the input `w[i]`, and connects the output of the register to `w[i+1]`.

Remaining constructs have standard semantics:

- `if-else`: Branch on parameter expressions.
- `let`-bindings: Name parameter expressions.
- `assume`: Asserted *trusted* facts that the type checker's SMT backend cannot automatically prove.
- Recursive instantiation to express modules such as reduction trees, which naturally decompose problems into smaller versions of themselves.

8.2.5 Reusing Instances

We lift Filament's reasoning about inter- and intra-iteration resource conflicts to Parafil. Filament's implementation for these checks walks over the entire component, collects all invocations associated with an instance, and computes the appro-

priate constraints. This approach is infeasible in Parafil since a parametric program can *generate* an unknown number of invocations for an instance. We redesign the check to be local using *instance availability intervals:*

```
A := new Add[32] in ['G, 'G+K];
```

Each instance specifies an availability interval which denotes when the instance can be invoked. Given this information, the Parafil type checker ensures:

- For each invocation's scheduling event E with delay d, $[E, E+d)$ is contained in the availability interval.
- Each pair of invocations is separated by at least d.

Finally, the type-checker ensures that the *delay* of the containing component's event is greater than the length of the instance availability interval. This is because if an instance is used for K cycles, then the component cannot process new inputs for at least K cycles. We include a simple pass to infer instance availability intervals for non-parametric programs for backward compatibility with Filament; extending this inference to parametric programs is future work.

8.3 Bottom-up parameterization

Input parameters, such as the bitwidth of an adder, allows the instantiating module to control how a child module is generated, enabling top-down parameterization. Parafil's *output parameters*, enables *bottom-up* parameterization: during elaboration (§8.4.3), a child module can return parameters which influence the elaboration of other modules in the parent. Parafil's key insight is that output parameters correspond to *existential types*: they hide the implementation details of a module

```
comp FPMult[E,W]<'G:1>
( l: ['G,'G+1] W,
  r: ['G,'G+1] W
) -> (
  o: ['G+3,'G+4] W)
```

(a) Concrete interface.

```
comp FPMult[E,W]<'G:1>
( l: ['G,'G+1] W,
  r: ['G,'G+1] W
) -> (o: ['G+L, 'G+L+1] W
) with { some L; }
```

(b) Abstract interface.

Figure 8.2: Parafil's output parameters abstract details such as latency and the type system ensure correct composition.

and force a parent to be abstract with respect to those details. Using this, Parafil seamless incorporates output parameters into its type system and provides strong composition guarantees.

8.3.1 Interfaces for Hardware Generators

Interfaces for modules generated by FloPoCo can be captured using input parameters: the bitwidth of the exponent and the mantissa. However, the target frequency configuration, which decides the pipelining and latency of the resulting module, is not easily encapsulated. Figure 8.2a shows the current of integrating such generated blocks: users pick a specific latency and write HDL glue code with respect to it.

8.3.2 Stable Interfaces for Generator Composition

Figure 8.2b shows the Parafil approach: using an output parameter `L` (defined using the syntax `some L`), the interface abstracts the latency of the module. The output is available `L` cycles after the input is provided but gives no information about its actual value, which is decided by FloPoCo.

In order to schedule computations that make use of outputs from such modules, programs need to use the output parameter access syntax (`M::L`). The Parafil program computes $a \times b + c$ using the `FPMult` module. The invocation for the adder, `A`, is scheduled at `'G+M::L` which depends on the value of `M::L`. Parafil's type-checker treats `M::L` as abstract and proves that the resulting computation will be correct for any value of `M::L`.

```
M := new FPMult[8, 32];
m := M<'G>(a, b);
A := new Add[8, 32];
a := A<'G+M::L>(m.o, c)
```

A common pattern when using output parameters is *threading them up*. For example, the module with the computation itself has an abstract latency which depends upon the value of `M::L`. Parafil components specify this using output parameter assignments:

```
comp Compute<'G:1>(...) with { some L; } {
  ...; L <- M::L } // Output param binding
```

Since the adder is combinational, the abstract latency `L` for the component is `M::L+0`. However, this dependency between the two latencies is not exposed in the signature; to a user of this component, the latency `L` is completely abstract. This abstraction be used for specializing the timing behavior of an implementation based on input parameters:

```
comp SmartMul[W]<'G:1>(...) with { some L } {
 if W < 4      { M := CombMult[W]; ..; L <- 0; }
 else if W < 9 { M := FastMult[W]; ..; L <- 2; }
 else          { M := SlowMult[W]; ..; L <- 4; }}
```

8.4 The Parafil Compiler

The Parafil compiler type-checks parametric programs and eliminates all compile-time abstractions used by the Parafil compiler to generate an unparameterized

Filament program.

8.4.1 Type Checking

The Parafil compiler lifts Filament's guarantees to parameterized programs through *symbolic reasoning*. It encodes Filament's typing constraints (§7.3) as an SMT formula and discharges them using a solver.

Encoding is defined recursively over Parafil statements. It takes a Parafil program (C_p), the current *path condition* (pc), and generates an SMT formulas ($\mathbb{P}$):

$$encode : C_p \to \text{pc} \to \mathbb{P}$$

Path conditions. Path conditions define the current set of known facts at a program point and are updated when entering conditionals and loops:

$$encode(\texttt{if c \{ t \} else \{ f \}, pc}) \triangleq$$
$$encode(\texttt{t}, pc \wedge c) \wedge encode(\texttt{f}, pc \wedge \neg c)$$

Constraints. During type checking, the encoding function generates assertions for each program statement.

$$encode(\texttt{b0[0..N] = b1[4..P]}, pc) \triangleq$$
$$pc \implies N = (P - 4) \wedge live(b_0, 0, N) \subseteq live(b_1, 4, P)$$

For example, for a bundle connection, the constraints to ensure that (1) bundle sizes match, and (2) that they have the correct availability intervals. Parameters are treated as unbounded integers which ensures that Parafil's guarantees extend

to all possible designs. Finally, constraints are guarded by the path condition. Once constructed, the negation of the query is asserted and if the SMT solver returns UNSAT, the original query is valid.

Output parameters. Within the defining component, constraints on output parameters are treated as assertions and each parameter assignment must satisfy them. When a component is instantiated, constraints on the output parameters are treated as *assumptions* and the parent module uses them to discharge proofs.

8.4.2 Partial Evaluation

Compiling Parafil components to Filament requires partially evaluating them with respect to parameters. The partial evaluator: (1) substitutes concrete values for parameters, (2) evaluates compile-time control flow such as `for` and `if` with the resulting concrete expressions, (3) returns bindings for output parameters computed during evaluation. The result of evaluation is a Filament component and concrete bindings for output parameters; all input parameters, control-flow, and recursion is eliminated.

The evaluator takes a Parafil program (C_p) a binding of parameters ($s : \text{Var} \rightharpoonup \mathbb{N}$) and returns a concrete Filament program (C_f) and bindings for the output parameters:

$$eval : C_p \to s \to (C_f, s)$$

Parameter expressions. Evaluating parameter expressions requires *substitution* which is recursively defined on the grammar of parameter expressions:

$$subst : p \to s \to \mathbb{N}$$

$$subst(x, s) = s(x)$$

$$subst(e_1 \text{ op } e_2) = apply(\text{op}, subst(e_1, s), subst(e_2, s))$$

$$subst(f(e_1, \ldots, e_n)) = apply(\text{f}, subst(e_1, s), \ldots, subst(e_n, s))$$

The *apply* evaluates functions and operations using concrete values. After substitution, all parameter expressions are reduced to concrete values.

Control flow. Control flow operators are given standard semantics using a recursive definition of *eval*:

$$eval(\texttt{if e \{ t \} else \{ f \}}, s) \triangleq$$

$$\text{if } subst(e, s) \text{ then } eval(t, s) \text{ else } eval(f, s)$$

Output parameter bindings. Parameter assignments update the output bindings:

$$eval(\texttt{p <- e}, s) \triangleq \textbf{empty}, \{p \mapsto subst(e, s)\}$$

The evaluator additionally ensures that all output parameters have exactly one assignment.

8.4.3 Elaboration

Figure 8.3 shows the process of elaboration for component `MulAdd` that computes $l \times r + c$. The inputs (`l`, `r`, `r`) are provided in the first cycle and the output

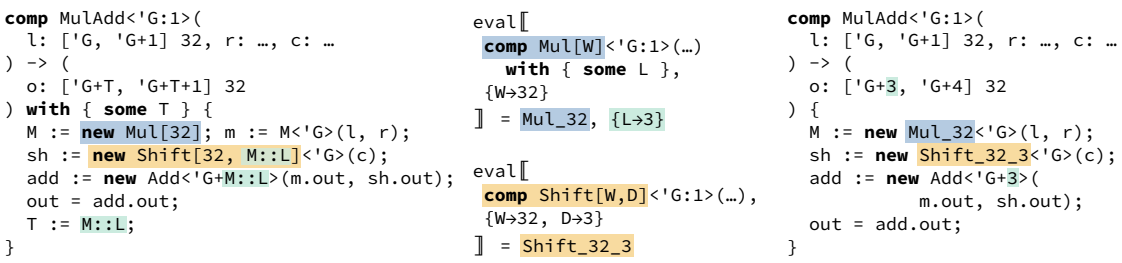


Figure 8.3: Parafil's elaboration pass compiles parametric programs into Filament programs. After elaboration, the Filament compiler's backend lowers the program to synthesizable hardware.

is available on cycle `T`, which is abstracted using an output parameter. This is because the implementation uses a multiplier component with an abstract latency `L`. The adder is scheduled using the multiplier's output parameter `M::L` and we use a shift register to delay the input `c`.

The process of compilation, or *elaboration*, start with `MulAdd` and recursively expands each instance. In Figure 8.3, the pass first encounters the instance `Mul[32]` and partially evaluates the definition of `Mul` with the binding $\{W \mapsto 32\}$ (Figure 8.3 center) to return a new, unparameterized Filament component `Mul_32` along with the binding $\{\texttt{M::L} \mapsto 3\}$. This binding for `M::L` added to `MulAdd`'s binding. The pass then evaluates `Shift[32, M::L]` using the concrete value for `M::L` and calling *eval* again. The resulting component (Figure 8.3 right) does not have any parameters and is a valid Filament program.

External components. External component and cannot be specialized by Parafil's partial evaluator; their definition exists only in Verilog and Parafil only has access to their interface. The elaboration pass substitutes concrete values for all parameters to such modules and passes them to the Filament compiler which already knows how to handle them.

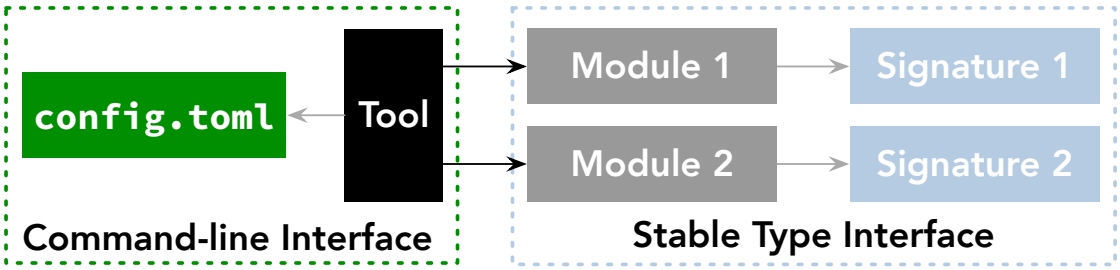


Figure 8.4: The Parafil `gen` framework. Tools define stable interfaces for the modules, used by the type checker, and a command-line interface, used by the elaboration pass.

Order of evaluation. Since components can define instances in any order, the elaboration pass first topologically sorts the instance list based on output parameter definitions and uses. If no topological sort exists, then component is invalid and cannot be elaborated.

8.4.4 Bundle Elimination

After elaboration, all Parafil programs have explicit assignments to bundle locations:

```
comp Foo<'G:1>(in[2]: for<i> ['G+i,'G+i+1] 32) {
  bundle A[2]: for<i> ['G, G+1] 32;
  A[0] = in[0]; out = A[0]; }
```

Bundle elimination inlines writes to a particular index into all of its uses. For bundles used in the signature, it instantiates explicit ports:

```
comp Foo<'G:1>(in0: ['G, 'G+1] 32),
 in1: ['G+1, 'G+2] 32) -> (...) { out = in0 }
```

```
comp FPExp[E, M]<'G:1>(
  X: ['G, 'G+1] W,
  Y: ['G, 'G+1] W,
) -> (
  R: ['G+L, 'G+L+1] W
) with {
  let W = E+M+3;
  some L;
}
```

(a) Type signature.

```
path = "flopoco"
# Module definition
[modules.FPExp]
name = "FPExp"
parameters = ["E", "M"]
cli = "FPExp ${M} ${E}"
name = "FPE${E}_${M}"
# Extract output
outputs.L = "depth"
```

(b) Command-line interface.

Figure 8.5: Interface for the `FPExp` module generated by FloPoCo. The type signature uses an output parameter to abstract over the latency. The configuration file describes how to invoke the tool and extract an output parameter value.

8.5 Composing External Generators

Parafil provides an in-language mechanism to compose designs generated by external generators. This framework, called Parafil `gen`, allows black box tools to define *stable interfaces* for generate-able modules and a command-line interface to invoke the tool to generate particular instances. The signatures are used by the type checker (§8.4.1) to ensure correct composition with any generateable instance while the command-line interface is used by the elaboration pass (§8.4.3) to generate instances. This decoupling between checking and generation allows users to prove that a design is correctly composed once and explore the design space repeatedely.

8.5.1 Type Checking

Type signatures for modules generated by tools must account for all possible implementations. Figure 8.5a shows the interface for the `FPExp` module. Input parameters configure the floating-point representation while the output parameter

abstracts the latency of the design. Using this component in Parafil is seamless: users instantiate the component like any other and the type checker ensures that the composition is correct using the provided interface.

8.5.2 Elaboration

The elaboration pass is responsible for transforming a parameterized Parafil program into an unparameterized Filament program. For components generated by external generators, elaboration needs a mechanism to invoke the tool and generate a Verilog file. Parafil `gen` uses *tool interfaces* which specifies the information needed to interface with a generator through a command-line interface. Figure 8.5b overviews the interface for the `FPExp` module:

- **Command-line interface.** The `cli` string describes how to invoke the FloPoCo binary to generate an `FPExp` module. The template string uses interpolation to passes parameters as command-line arguments.
- **Name generation.** The `name` string describes how the tool generates the name of the final Verilog module given a set of parameters.
- **Bindings for output parameters.** The `output` dictionary maps output parameters to strings. Given `outputs.L = "depth"`, Parafil will execute the tool and will look for the line `depth = <n>` in the standard output stream, setting the binding of `L` to n.

Generating an external definition. After executing the tool, Parafil has the bindings for output parameters and the name and path to the Verilog module. To generate an unparameterized Filament wrapper, it partially evaluates the signature

of module (Figure 8.5a). For example, if the elaboration for `FPExp[16,4]` generates the binding $\{L \mapsto 3\}$, the final Filament signature will be:

```
ext comp FPE16_4<'G:1>(
  clk: 1, X: ['G, 'G+1] 23, Y: ['G, 'G+1] 23
) -> (R: ['G+3, 'G+4] 23);
```

8.6 Parameterized FFT

Fast Fourier transforms (FFTs) are widely-used signal processing algorithms. Hardware generators for FFTs are widely studied [107, 108] because of the plethora of use cases and many implementation choices: radix size, circuit reuse, and the dataflow [45, 122]. We implement several FFT modules using Parafil:

- A Parafil implementation with parametric reuse.
- A second Parafil implementation that uses FloPoCo [46] to generate floating-point modules.
- An implementation that uses XLS [66] to generate butterfly modules and reuses them in Parafil code.

8.6.1 FFT Building Blocks

We use the Pease dataflow [122] which provides a regular structure for both the butterfly and permutation stages. We design an *iterative* implementation that shares subcircuits between stages and a *streaming* implementation that provides higher throughput at the cost of higher resource usage. Our implementation uses Parafil's

```
comp Butterfly[W, E]<'G: II>(
  in0[2]: ['G, 'G+II] W,
  in1[2]: ['G, 'G+II] W,
  twid[2]: ['G, 'G+II] W
) -> (
  out0[2]: ['G+L, 'G+L+1] W,
  out1[2]: ['G+L, 'G+L+1] W
) with {
  some II, L where L >= II > 0;
}
```

(a) Signature of the butterfly component with abstract latency and initiation interval.

```
comp Perm[Stages, W]<'G: 1>(
 inp[P][2]: ['G, 'G+1] W
) -> (
 out[P][2]: ['G, 'G+1] W
) with { let P = pow2(Stages) }
 where Stages > 0, W > 0 {
 for i in 0..P/2 {
  out{i}{..} = inp{i*2}{..};
  out{i+P/2}{..} =
      inp{i*2+1}{..}; }}
```

(b) Permutation component implemented using bundles.

```
comp BitRev[Stages, W]<'G: 1>(
 inp[P][2]: ['G, 'G+1] W
) -> (
 out[P][2]: ['G, 'G+1] W
) with { let P = pow2(Stages) }
 where Stages > 0, W > 0 {
 for j in 0..P {
  let br = bit_rev(j, Stages);
  out{j}{0..2} = inp{br}{0..2};
}}
```

(c) Bit reversal implemented using the `bit_rev` parameter function.

Figure 8.6: Building blocks for the Pease FFT. Complex numbers are represented as two element bundles.

output parameters (§8.3) to abstract modules' latencies, which lets us seamlessly replace their implementations.

Given N complex numbers as inputs, the Pease dataflow applies a bit-reversal followed by a series of stages, each consisting of $N/2$ parallel butterflies. It then permutes the output from each stage using the stride permutation to produce the next stage's input.

Bit reversal permutation. The Pease FFT requires the inputs to be bit-reversed before they are passed into the first butterfly stage. Since bit-reversal is a simple operation, the component (Figure 8.6c) has a combinational implementation.

Butterfly. The butterfly component, given complex inputs a, b and the twiddle factor ω, computes:

$$(a, b, \omega) \mapsto (a + b\omega, a - b\omega)$$

Figure 8.6a shows the Parafil signature of the butterfly module. Output parameters model the latency (`L`) and initiation interval (`II`), the number of cycles before the module can start processing new inputs. This lets us transparently swap out combinational, pipelined, or externally generated implementations of the butterfly.

Stride permutation. The stride permutation stage connects the outputs from one butterfly stage to the inputs of the next. The computation is:

$$out(i) = \begin{cases} in(\frac{i}{2}) & i \bmod 2 = 0 \\ in(\frac{i-1+N}{2}) & \text{otherwise} \end{cases}$$

This permutation is static, so our implementation (Figure 8.6b) uses bundles to combinationally rewire inputs to outputs.

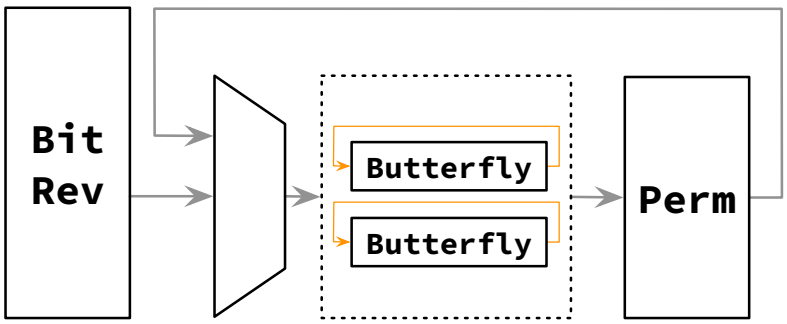


Figure 8.7: Iterative FFT that reuses butterfly components within and across stages.

8.6.2 Iterative FFT

Figure 8.7 overviews the parameterized iterative FFT. The bit-reversal stage transforms the inputs and passes it to the butterflies, which in turn forward their output to the permutation. The permutation component then sends the inputs back to the butterflies for the next stage. The design uses exactly one bit-reversal and one permutation component.

```
comp IterFFT[N, B=N/2]<'G:L>(
  in0[N][2]: ['G, 'G+1] 32) -> (
  out[N][2]: ['G+L, 'G+L+1] 32
) with { some L where L > 0 }
```

The design is parameterized by the number of inputs (`N`) and the number of butterfly components (`B`), with the default being $N/2$ to allow fully parallel computation for a stage of the FFT. If fewer butterflies are provided, the FFT component reuses them over time and instantiates registers to store the output for a stage till the output from all butterflies is computed. Finally, the latency and throughput of the iterative FFT is abstracted using the output parameter (`L`). This is because our implementation uses butterfly components with abstract latencies (Figure 8.6a) and that requires threading the abstract latency through (§8.3.2).

FloPoCo integration. The Parafil FFT implements the complex math using a 5 stage floating-point multiplier and a 5 stage floating-point adder. In order to explore area-throughput trade-offs, we abstract the interface for the floating-point modules and use FloPoCo [46] to generate implementations targeting different frequencies. Because of the `gen` framework, using FloPoCo generated components is seamless:

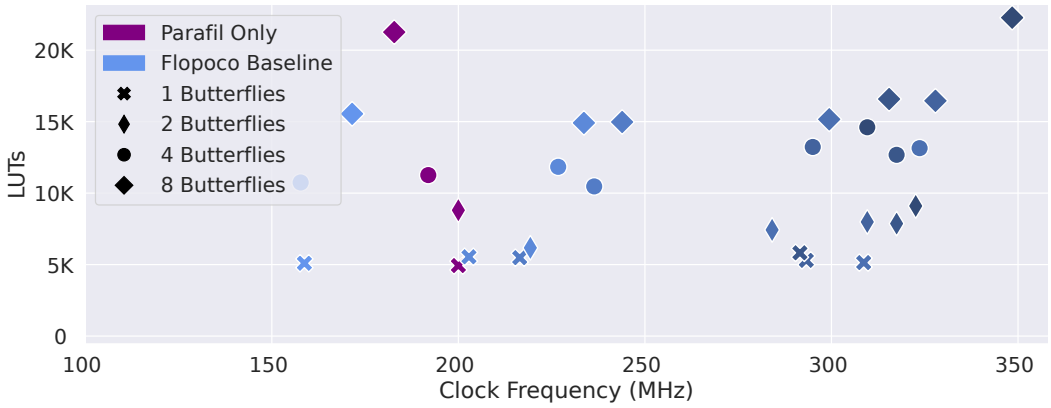
```
ac := new FPMult[E, M]<'G>(in{0}, twiddle{0});
bd := new FPMult[E, M]<'G>(in{0}, twiddle{1});
re := new FPSub[E, M]<'G+FM::L>(ac.R, bd.R);
```

The `FPMult` and `FPSub` are defined by FloPoCo and are automatically generated during elaboration (§8.5.2).

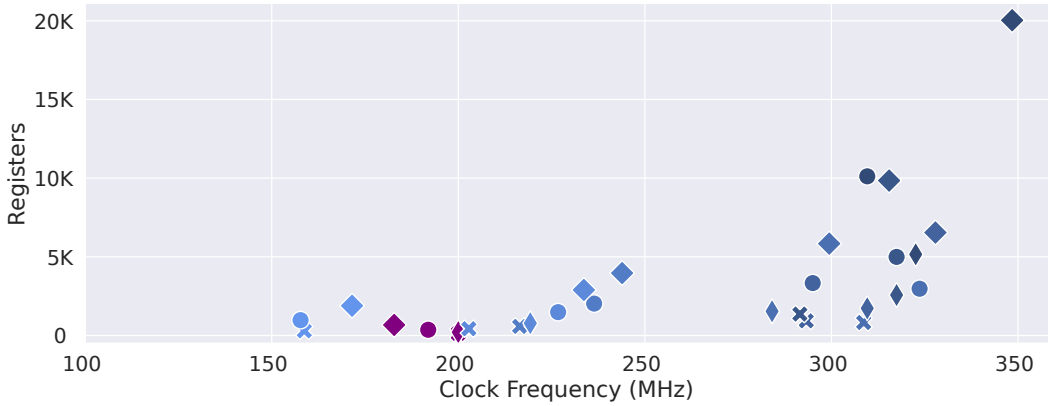
Evaluation. Our evaluation of the iterative FFT modules seeks to answer two questions:

1. Does the parameterized Parafil design offer meaningful area-throughput trade-offs?
2. Does using FloPoCo modules enable us to explore a larger space of trade-offs?

Figure 8.8 summarizes our results. We generate Parafil point by choosing the number of butterflies to be $\{1, 2, 4, 8\}$ and the FloPoCo-Parafil points by additionally sweeping the target frequency input for FloPoCo from 300–900MHz. The graphs report post place-and-route resource usage using Vivado 2020.2 against the maximum frequency for all points that can meet timing at at least 100MHz. The designs generated by the parameterized Parafil implementation offer area-frequency trade-off: reducing reuse improves frequency at the cost of resources. The FloPoCo-Parafil expresses a larger design space: darker points have a higher target frequency input



(a) Maximum frequency vs LUT usage.



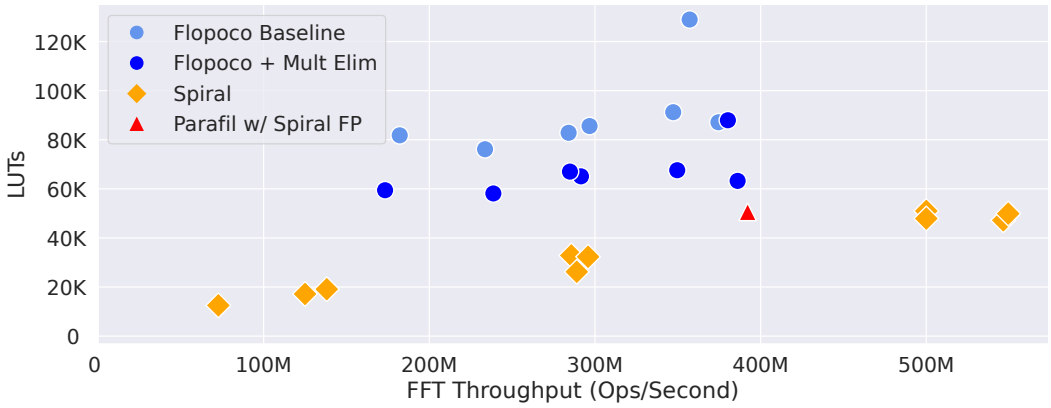
(b) Maximum frequency vs. register usage.

Figure 8.8: Iterative FFT. Darker points have deeper pipelines and shapes represent different amounts of reuse.

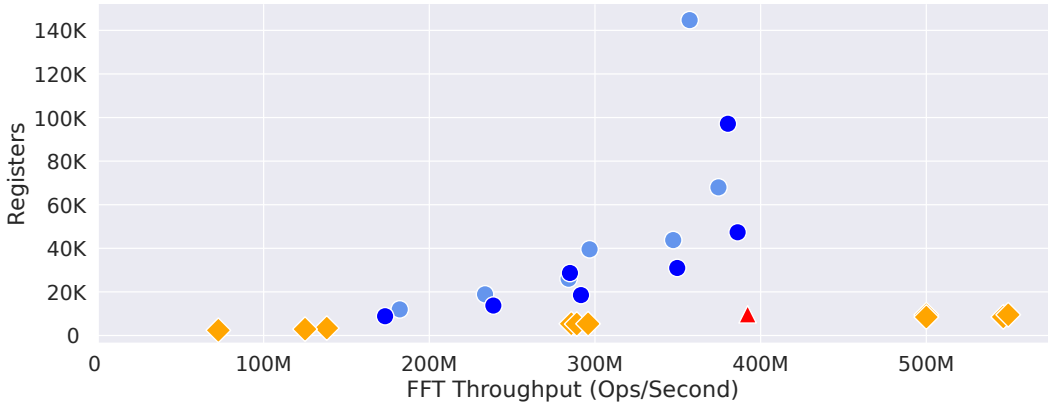
and are more pipelined. These designs can achieve a wider range of frequencies than the pure Parafil implementation.

8.6.3 Streaming FFT

We implement a streaming FFT as a special case of the iterative FFT that instantiates separate stages and does not reuse butterflies. We integrate FloPoCo and compare the generated design to FFTs generated by Spiral [107].



(a) Latency vs LUT usage.



(b) Latency vs. register usage.

Figure 8.9: Streaming FFTs in Parafil and Spiral compared to optimized designs.

Initial comparison. Figure 8.9 reports usage against throughput, computed as ratio of frequency over initiation interval. Throughput is an application-level metric that allows us to compare different FFT implementation. Between FloPoCo-Parafil designs and Spiral, the latter does strictly better because Spiral implements application-specific FFT optimizations that our implementation does not yet replicate.

Eliminating unit multiplies. Spiral opportunistically eliminate floating-point multiplier when the twiddle factors are constants (1 or $-i$). This changes the

latency of butterfly modules in some stages. Parafil's output parameters abstract the latency and make it easy to express this optimization:

```
if is_one(Twid) { // is 1
  .. // Forward inputs
  L <- 0; // Purely combinational
} else if is_i(Twid) { // is -i?
  .. // Negate and forward inputs
  L <- 0; // Purely combinational
} else { // floating-point multiplier needed
  M := new FPMul; ...; L <- M::L; // Multi-cycle
}
```

The output parameter `L` reflects the component's conditional latency: combinational (0) for unit factors and sequential (the same as the multiplier) otherwise. This abstraction over `L` is a unique feature of Parafil: clients do not *provide* this latency parameter to instantiate `OptMul`; instead, they *adapt* to the latency that the component reports. This abstraction supports the optimization efficiently while keeping implementation details hidden. Figure 8.9 summarizes the results: register usage goes down by 28.9%, LUT usage by 25.7%, and latency by 27.8%, bringing us closer to the Spiral designs.

Integrating Spiral. The Spiral implementation still outperforms the Parafil-FloPoCo integration because its modules are mapped onto DSP blocks more often. Figure 8.9 shows that if we replace some Parafil blocks with the Spiral ones, the resulting design is competitive. A better approach, however, is to simply define an interface for Spiral FFTs and import them in Parafil:

```
comp FFT[N]<'G: II>(
  X[W*2][N/W]: for<_, i> ['G+i, 'G+i+1] 32
) -> (
  Y[W*2][N/W]: for<_, i> ['G+L+i, 'G+L+i+1] 32
) with {
  some II, L where L >= II > 0;
  some W where N % W == 0; // streaming width
} where N > 0;
```

The interface for Spiral modules abstracts over the latency, initiation interval, and the *streaming width* (`W`) using output parameters. For $N \neq W$, where N is the number of points, this means that inputs are provided over multiple cycles enabling resource savings.

8.7 Enriching High-Level Design

Heterogeneous generator composition, enabled by Parafil abstractions, can enable new hardware design flows where high- and low-level programming models for hardware design can be combined. We prototype such a flow by combining Google's XLS toolchain with Parafil and show how it offers the best of both worlds: XLS enables rapid design of efficient, pipelined datapaths, and Parafil enables precise and correct resource reuse.

8.7.1 DSLX Language

XLS [66] represents Google's continued investment in high-level programming models for hardware after using high-level synthesis tools to design production chips [130]. XLS provides an expression-oriented functional language called DSLX to express computations which are automatically pipelined and compiled to synthesizable hardware.

DSLX programs look like standard imperative programs:

```
pub fn ALU<E:u32, M:u32>(
  left: Float<E, M>, right: Float<E, M>, op: u1
) -> Float<E, M> {
  if (op == 0) { float::add(l, r) }
  else { float::mul(l, r) }
```

```
}
```

This implements a ALU parameterized on the floating-point representation. Unlike HDL programs—which instantiate circuits and schedule computations—DSLX use function calls and control flow to express the design. However, this high-level representation has drawbacks: XLS cannot currently express *resource reuse* which means it cannot express designs like Parafil's iterative FFT (Figure 8.7).

8.7.2 Integrating with Parafil-gen

Unlike FloPoCo, XLS is a programming language and can therefore support arbitrary computation instead of a limited set of hardware blocks. We define a wrapper script that integrates XLS to Parafil `gen`'s block-based interface. The script does the following:

- Maps names of blocks to XLS template programs.
- Given an XLS block and a set of parameters, uses a preprocessor to replace the value of the parameters.
- Runs the XLS toolchain and returns a Verilog module.

For each module, we provide a type signature and an entry in the configuration file (§8.5):

```
pub fn A(
  x: Float<E, M>
) -> Float<E, M>;
```

```
comp A[E, M]<'G:1>(
  x: ['G, 'G+1] E+M+1
) -> (
  out: ['G, 'G+1] E+M+1
) with { some L }
```

The XLS signature above is invalid since `E` and `M` are unbound. The preprocessor in our XLS scripts replaces concrete values for them before invoking the

```
pub fn Butterfly(
  a_r: Float<E, M>,
  a_i: Float<E, M>,
  b_r: Float<E, M>,
  b_i: Float<E, M>,
  t_r: Float<E, M>,
  t_i: Float<E, M>
) -> (
  Float<E, M>,
  Float<E, M>,
  Float<E, M>,
  Float<E, M>)
```

(a) XLS signature.

```
comp Butterfly[E, M]<'G:1>(
  a_r: ['G, 'G+1] W,
  a_i: ['G, 'G+1] W,
  b_r: ['G, 'G+1] W,
  b_i: ['G, 'G+1] W,
  t_r: ['G, 'G+1] W,
  t_i: ['G, 'G+1] W
) -> (
  out: ['G+L, 'G+L+1] 4*W
) with {
  let W = E+M+1;
  some L where L >= 1; }
```

(b) Parafil signature.

Figure 8.10: Butterfly module in XLS.

XLS toolchain. The Parafil interface for this module has both input parameters—which provide bindings for `E` and `M`—and an output parameter—which abstracts the latency.

8.7.3 Iterative FFT with XLS

XLS does not support resource reuse and therefore cannot express an iterative FFT. Instead, we use XLS to design a butterfly module (§8.6.1) and use Parafil code to express the resource reuse. Figure 8.10a shows the XLS interface for the module: it takes real and imaginary parts of the inputs and the twiddle factor. Figure 8.10b explicates its timing behavior: all inputs are accepted in the first cycle and the output is available after `L` cycles, where `L` is determined by XLS based on the user-specified pipeline depth.

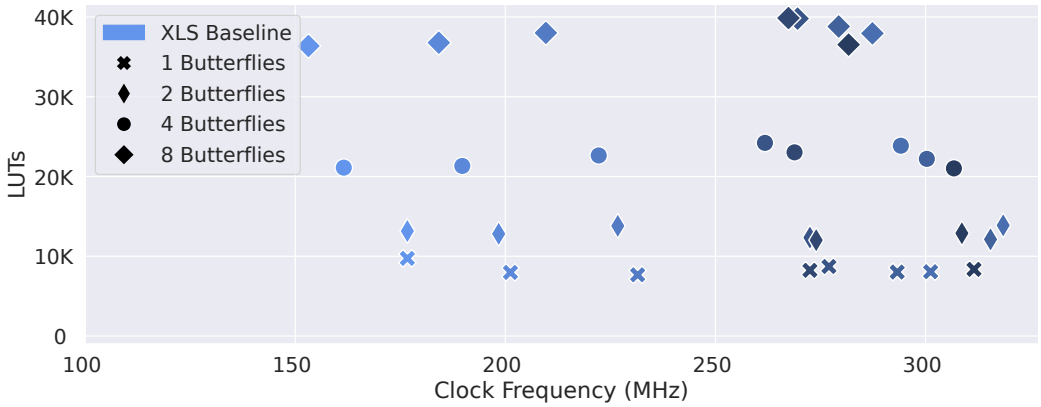
DSLX's expression-based representation makes it easy to express the butterfly computation:

```
let re = sub(mul(a_r, t_r), mul(b_i, t_i));
let img = ...;
// in0 + (w * in1)
```

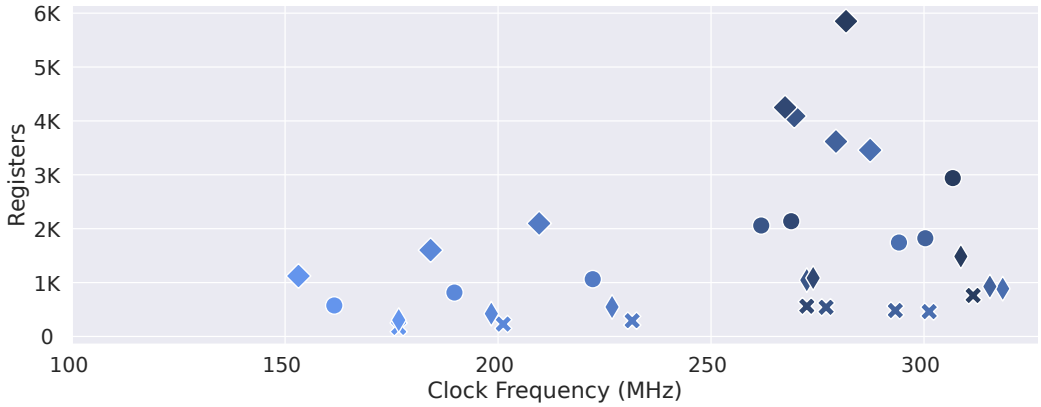
```
let o0_r = add(a_r, re); let o0_i = add(a_i, im);
// in0 - (w * in1)
let o1_r = sub(a_r, re); let o1_i = sub(a_i, im);
return (o0_r, o0_i, o1_r, o1_i)
```

Expressing reuse. Figure 8.10b does not match the interface of the butterfly module used in our iterative FFT (Figure 8.6a). We define a wrapper module that exposes the same interface as the iterative FFT's butterfly and explicitly forward signals to and from bundles. This allows us to *transparently* replace our butterfly module with XLS. Furthermore, it is *the only change* needed to express butterfly reuse. The iterative design is abstract with respect to the butterfly implementation and can therefore reuse any implementation, including the XLS implementation; our job is done.

Figure 8.11 summarizes the results of exploring the design space of the combined Parafil-XLS iterative FFT. We select between reuse factors in $\{1, 2, 4, 8\}$ and number of pipeline stages in XLS from $\{5, 10, \ldots, 45\}$. We report LUT and register usage against the maximum frequency and remove any design points that cannot meet timing at 100MHz. As the amount of reuse decreases and number of stages increase, designs achieve higher frequencies and require more resources.



(a) FFT Throughput vs LUT usage.



(b) FFT Throughput vs. register usage.

Figure 8.11: Parafil FFTs using XLS butterflies. Darker points have more pipeline stages and different shapes represent different amount of sharing.

CHAPTER 9

CONCLUSION

9.1 Retrospective

This dissertation examined programming models for hardware design at various levels of abstraction and demonstrated that *reasoning about time* can enable tools to provide modular and efficient abstractions. Since publication, each system has seen adoption and impact in different ways:

Calyx (§§ 4–6). Calyx demonstrated that intermediate languages can provide both scalable compiler analyses and precise hardware specification through its novel group and control abstractions. Group-like abstractions have become common in compiler intermediate languages (ILs) published since [103, 160, 163]. The system has gained significant traction, serving as the foundation for multiple frontend compilers targeting languages like Halide [69], TVM [30], and C++, as well as hardware generators [26]. Additionally, Calyx has been integrated into the LLVM CIRCT infrastructure [153] and enabled novel debugging [17] and profiling capabilities for hardware accelerators.

Filament (§§ 7 and 8). While contemporary projects attempted to provide reasoning about module latencies [103, 142], Filament was the first to provide holistic reasoning about a module's pipelining behavior and enabled modular composition of pipelined circuits. Filament's type system, being built on a simple structural HDL, was broadly applicable and has since influenced the design of other systems [66, 81].

Dahlia (§3). Dahlia demonstrated the predictability challenges of high-level synthesis (HLS) tools and rectified them using a novel substructural type system to reason about constraints of circuits within high-level languages. Subsequent systems added affine reasoning for circuits [142] and used Dahlia's notion of logical timesteps [164] in related problem domains.

9.2 Open Questions

The demand for specialized accelerators, coupled with customized compiler and system stacks, will keep growing in the next decade. Fundamental ideas in programming systems, such as the separation between architecture and programs, need to be re-examined to enable next-generation systems to effectively utilize accelerators. Several fundamental challenges in accelerator design and programming are fertile grounds for future research:

Abstractions for accelerator programming While accelerators offer tremendous potential, their utility hinges on effective programming interfaces. Modern programming models must elegantly expose critical low-level features like memory hierarchies [37, 90, 123] while enabling crucial optimizations such as computation-communication overlap [82]. The success of accelerator architectures fundamentally depends on their programmability.

Designing programmable accelerators Traditional high-level synthesis has primarily targeted fixed-function accelerators. However, real-world applications demand flexible, programmable solutions to justify the substantial investment in ac-

celerator development. The challenge lies in developing programming abstractions that can specify and generate programmable hardware accelerators. Previous research in ASIPs [15, 38, 47, 65] laid the groundwork for processor generation but struggled with complex architectural features. Recent advances in compiler synthesis [150] and integrated toolchains [51] show promise in limited domains, but generalizing these approaches remains an open challenge.

Rethinking the RTL abstraction The register transfer level (RTL) abstraction, while foundational to hardware design, warrants re-examination. As demonstrated by Filament (§7), elevating timing behavior to the source level can prevent entire categories of hardware bugs. Future hardware description languages (HDLs) must enable reasoning about critical properties like deadlock freedom [156], concurrent behavior [109], and security guarantees [165] without compromising performance. Drawing inspiration from Rust's success in systems programming, we need innovative language designs, type systems, and compilation strategies to revolutionize hardware development.

For accelerators to become ubiquitous, designers have to take a cross-stack view of the challenges, designing abstractions to program them and to optimize them down to the level of the last transistor. Researchers have the opportunity to provide a firm foundation for languages and machines that will come to define the next generation of computing.

BIBLIOGRAPHY

[1] Amaranth: A modern hardware definition language and toolchain based on python. URL https://github.com/amaranth-lang/amaranth.

[2] Design Compiler. URL https://archive.is/BW7Yj.

[3] Polybench/c 4.2.1. URL https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.

[4] SpinalHDL: Scala based HDL. URL https://github.com/SpinalHDL/SpinalHDL.

[5] Ali E. Abdallah and John Hawkins. Formal behavioural synthesis of Handel-C parallel hardware implementations from functional specifications. In *Hawaii International Conference on System Sciences (HICSS)*, 2003.

[6] F.E. Allen and J. Cocke. *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971.

[7] Amazon Web Services. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[8] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference*, 1967.

[9] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *USENIX Symposium on Networked System Design and Implementation (NSDI)*, 2020.

[11] IEEE Standards Association. Ieee standard for systemverilog–unified hardware design, specification, and verification language, 2018.

[12] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. CλaSH: Structural descriptions of synchronous hardware using Haskell. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010. doi: 10.1109/DSD.2010.21.

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

[14] Henry G. Baker. "Use-once" variables and linear objects: Storage management, reflection and multi-threading. *SIGPLAN Notices*, 1995.

[15] Mario R Barbacci. Instruction set processor specifications (isps): The notation and its applications. *IEEE Transactions on Computers*, 1981.

[16] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 2000.

[17] Griffin Berlstein, Rachit Nigam, Christophe Gyurgyik, and Adrian Sampson. Stepwise debugging for hardware accelerators. In *ACM International Con-*

ference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023. doi: 10.1145/3575693.3575717.

[18] J Bernardy, Mathieu Boespflug, Ryan Newton, Simon L. Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[19] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 1996.

[20] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[21] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020. doi: 10.1145/3385412.3385965.

[22] Robert K Brayton, Gary D Hachtel, and Alberto L Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 1990.

[23] Stephen Brookes. A semantics for concurrent separation logic. In *International Conference on Concurrency Theory*. Springer, 2004.

[24] 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.

[25] 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. doi: 10.1145/1950413.1950423.

[26] Benjamin Carleton. A numerics frontend for calyx. URL https://github.com/cucapra/calyx-nums.

[27] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of latency-insensitive design. 2001.

[28] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014.

[29] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. Allo: A programming model for composable accelerator design. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2024.

[30] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[31] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 2014.

[32] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 2016.

[33] 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. doi: 10.1145/3373087.3375297.

[34] 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. doi: 10.1145/3490422.3502362.

[35] 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. doi: 10.1109/TC.2023.3292590.

[36] Yaohan Chu. Introducing CDL. *Computer*, 1974.

[37] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011.

[38] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2003.

[39] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, 2015.

[40] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A Pythonic approach for rapid hardware prototyping and instrumentation. In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2017. doi: 10.23919/FPL.2017.8056860.

[41] J. Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*, 2006. doi: 10.1145/1146909.1147025.

[42] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *International SoC Conference*, 2006.

[43] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. 2011.

[44] Jason Cong and Jie Wang. Polysa: Polyhedral-based systolic array auto-compilation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018. doi: 10.1145/3240765.3240838.

[45] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 1965.

[46] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 2011.

[47] Hugo De Man, Jan Rabaey, Paul Six, and Luc Claesen. Cathedral-II: A silicon compiler for digital signal processing. *IEEE Design & Test of Computers*, 1986.

[48] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 1974.

[49] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

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

[51] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. PEak: A single source of truth for hardware design and verification. *arXiv preprint arXiv:2308.13106*, 2023.

[52] 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. doi: 10.1145/3385412.3385983.

[53] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankar-

alingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[54] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *Summit on Advances in Programming Languages*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015.

[55] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog hdl and its ancestors and descendants. *Proceedings of the ACM on Programming Languages*, (HOPL), 2020.

[56] Peter L Flake, Gerry Musgrave, and Mike Shorland. The hilo logic simulation language. 1975.

[57] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, 2006.

[58] 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.

[59] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[60] Sameh Galal, Ofer Shacham, John S Brunhaver II, Jing Pu, Artem Vassiliev, and Mark Horowitz. Fpu generator for design space exploration. In *IEEE Symposium on Computer Arithmetic*. IEEE, 2013.

[61] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[62] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Design Automation Conference (DAC)*. IEEE, 2021.

[63] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, April 1992.

[64] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022.

[65] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *International conference on Compilers, architecture and synthesis for embedded systems*, 2003.

[66] Google. XLS: Accelerated computing at google. URL https://google.github.io/xls/.

[67] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

[68] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

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

[70] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

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

[72] Berkeley hardfloat authors. Berkeley hardware floating-point units. URL https://github.com/ucb-bar/berkeley-hardfloat.

[73] 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. 2014. doi: 10.1145/2601097.2601174.

[74] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark

Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. 2016. doi: 10.1145/2897824.2925892.

[75] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *CommunicationsoftheACM*, 2019.

[76] Frederick J Hill. Introducing AHPL. *Computer*, 1974.

[77] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 2008.

[78] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[79] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, Jan 2009.

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

[81] Jane Street. HardCaml: Register transfer level hardware design in OCaml. URL https://github.com/janestreet/hardcaml.

[82] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

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

[84] 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.

[85] Hyegang Jun, Hanchen Ye, Hyunmin Jeong, and Deming Chen. Autoscaledse: A scalable design space exploration engine for high-level synthesis. *ACM Transactions on Reconfigurable Technology and Systems*, 2023.

[86] Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and

Rachit Nigam. Unifying static and dynamic intermediate languages for accelerator generators. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2024. doi: 10.1145/3689790.

[87] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)*, 2016.

[88] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2017.

[89] 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. doi: 10.1145/3192366.3192379.

[90] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. Stash: Have your scratchpad and cache it too. *ACM SIGARCH Computer Architecture News*, 2015.

[91] David C Ku and Giovanni De Micheli. *Hardware C: a language for hardware design*. Computer Systems Laboratory, Stanford University, 1988.

[92] Hsiang-Tsung Kung. Why systolic architectures? *IEEE Computer*, 1982. doi: 10.1109/MC.1982.1653825.

[93] Tadahiro Kuroda. CMOS design challenges to power wall. In *International Microprocesses and Nanotechnology Conference*. IEEE, 2001.

[94] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey. SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[95] 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.

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

[97] 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. doi: 10.1109/CGO51591.2021.9370308.

[98] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 1991.

[99] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the ACM SIGCOMM Conference*, 2016.

[100] Katie Lim, Matthew Giordano, Theano Stavrinos, Irene Zhang, Jacob Nelson, Baris Kasikci, and Thomas Anderson. Beehive: A flexible network stack for direct-attached accelerators. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024.

[101] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014. doi: 10.1109/MICRO.2014.50.

[102] 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.

[103] Kingshuk Majumder and Uday Bondhugula. Hir: An mlir-based intermediate representation for hardware accelerator description, 2021.

[104] Paolo Mantovani, Robert Margelli, Davide Giri, and Luca P Carloni. HL5: a 32-bit RISC-V processor designed with high-level synthesis. In *IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2020.

[105] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. In *High Integrity Language Technology (HILT)*, 2014.

[106] Carver Mead and Lynn Conway. Introduction to VLSI systems, 1980.

[107] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 2012. doi: 10.1145/2159542.2159547.

[108] VM Milovanović and ML Petrović. A highly parametrizable chisel HCL generator of single-path delay feedback FFT processors. In *International Conference on Microelectronics (MIEL)*. IEEE, 2019.

[109] Fabrizio Montesi. *Choreographic programming*. IT-Universitetet i København, 2014.

[110] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 1998.

[111] 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. doi: 10.1145/2554688.2554786.

[112] Mayur Naik, Alexander Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[113] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[114] Matthias Nickel and Diana Göhringer. A survey on architectures, hardware acceleration and challenges for in-network computing. *ACM Trans.*

Reconfigurable Technol. Syst., 2024. doi: 10.1145/3699514. URL https://doi.org/10.1145/3699514.

[115] 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. doi: 10.1145/3385412.3385974.

[116] 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. doi: 10.1145/3445814.3446712.

[117] 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. doi: 10.1145/3591234.

[118] Rachit Nigam, Ethan Gabizon, Edmund Lam, and Adrian Sampson. Correct and compositional hardware generators. 2024. doi: arXiv:2401.02570.

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

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

[121] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89, 2020. URL https://developer.nvidia.com/cuda-toolkit.

[122] Marshall C. Pease. An adaptation of the fast fourier transform for parallel processing. *Journal of the ACM*, 1968. doi: 10.1145/321450.321457.

[123] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[124] 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. doi: 10.1109/FPL.2013.6645550.

[125] Stèphane Pouget, Louis-Noël Pouchet, and Jason Cong. Enhancing high-level synthesis with automated pragma insertion and code transformation framework. *arXiv preprint arXiv:2405.03058*, 2024.

[126] Raghu Prabhakar and Sumti Jairath. Sambanova sn10 rdu: Accelerating software 2.0 with dataflow. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–37. IEEE, 2021.

[127] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[128] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. 2017. doi: 10.1145/3107953.

[129] 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. doi: 10.1145/2491956.2462176.

[130] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. Warehouse-scale video acceleration: co-design and deployment in the wild. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[131] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Douglas Orr, and Richard Sanzi. Mach: a foundation for open systems (operating systems). In *Proceedings of the Second Workshop on Workstation Operating Systems*. IEEE, 1989.

[132] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[133] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*. IEEE, 2002.

[134] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. 2010.

[135] Amit Sabne. XLA: Compiling machine learning for peak performance. *Google Res*, 2020.

[136] 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.

[137] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024.

[138] Colin Schmidt and Adam Izraelevitz. A fast parameterized sha3 accelerator. In *Technical Report*. EECS Department, University of California, 2015.

[139] Ofer Shacham, Megan Wachs, Andrew Danowitz, Sameh Galal, John Brunhaver, Wajahat Qadeer, Sabarish Sankaranarayanan, Artem Vassiliev, Stephen Richardson, and Mark Horowitz. Avoiding game over: Bringing design to the next level. In *Design Automation Conference (DAC)*, 2012.

[140] 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.

[141] Rohit Sinha and Hiren D Patel. synASM: A high-level synthesis framework with support for parallel and timed constructs. 2012.

[142] Frans Skarman and Oscar Gustafsson. Spade: An expression-based HDL with pipelines, 2023.

[143] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

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

[145] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[146] Stuart Sutherland and Don Mills. Standard gotchas subtleties in the verilog and systemverilog standards that every engineer should know. In *Synopsys User Group Conference Proc.(SNUG 2006/7)*. Citeseer, 2006.

[147] Stuart Sutherland, Don Mills, and Chris Spear. Gotcha again: More subtleties in the Verilog and SystemVerilog standards that every engineer should know.

[148] Paul Teehan, Mark Greenstreet, and Guy Lemieux. A survey and taxonomy of gals design styles. *IEEE Design & Test of Computers*, 24:418–428, 10 2007. doi: 10.1109/MDT.2007.151.

[149] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A

language for streaming applications. In *International Conference on Compiler Construction*. Springer, 2002.

[150] Samuel Thomas and James Bornholt. Automatic generation of vectorizing compilers for customizable digital signal processors. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024.

[151] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[152] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[153] 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.

[154] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: a virtual machine for programming modern FPGAs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021. doi: 10.1145/3453483.3454075.

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

[156] Muralidaran Vijayaraghavan et al. Bounded dataflow networks and latency-insensitive circuits. In *Conference on Formal Methods and Models for Co-Design*. IEEE, 2009.

[157] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM)*, 2010.

[158] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[159] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. The q100 database processing unit. *IEEE Micro*, 2015.

[160] Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. Cement: Streamlining FPGA hardware design with cycle-deterministic eHDL and synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024.

[161] Xilinx Inc. SDAccel: Enabling Hardware-Accelerated Software, . https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

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

[163] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. Hector: A multi-level intermediate representation for hardware synthesis methodologies. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022. doi: 10.1145/3508352.3549370.

[164] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C Myers. Pdl: a high-level hardware design language for pipelined processors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[165] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices*, 50(4):503–516, 2015.

[166] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

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

[168] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd generation berkeley out-of-order machine. May 2020.

ACRONYMS

AHG Automatic hardware generation. The process of transforming computational descriptions into circuit implementations. For example: high-level synthesis (HLS). 120

AST Abstract syntax tree. A representation of a program where each construct is a node with edges to all of the constructs contained within it. Commonly used as the first syntactic representation of a program in a compiler after parsing. 72, 106

CDFG Control-data flow graph. A representation of programs that describes both control flow and data flow. 73

DSL Domain-specific language. A specialized programming language designed use in a particular domain like image processing, scientific computing, genomics, etc. 13, 14, 53

eHDL Embedded hardware description language. An hardware description language (HDL) embedded within a software programming language. The software language acts as a host language and provide metaprogramming capabilities. 12, 160

FPGA Field programmable gate array. Implemeted as sea of configurable logic units (such as look-up tables) and interconnect, FPGAs can be used to simulate arbitrary digital circuits. 17, 106, 225

HDL Hardware description language. A programming language used to describe hardware using abstractions such as register transfer level (RTL). 3, 5, 9, 11, 13, 120, 122, 160, 165, 199, 224

HLS High-level synthesis. The process of compiling high-level, computational programs to low-level, circuit descriptions. 13, 17, 53, 67, 71, 108, 198, 224

II Initiation interval. The number of a cycles before which a pipelined module can accept a new set of inputs. 164, 166

IL Intermediate language. Representation of a program designed to simplify analysis and optimization. Often generated from an AST after initial validity checks. 6, 15, 16, 53, 54, 103, 106, 108, 197

IPC Number of instructions executed by a processor in one cycle. More than 1 for superscalar processors. 10

IR Intermediate representation. The data structures used to implement the intermediate language. 106

LUT Look-up table. Maps input binary values to outputs. Used as the logical building blocks for field programmable gate arrays (FPGAs). 47, 111

PDK Process design kit. Interface between circuit designers and chip foundaries. A PDK contains the necessary information to map a circuit to transistors. 11

PE Processing element. A simple computational unit like an arithmetic logic unit (ALU) that acts as a building block for more complex accelerators. 67, 108, 109

PPA The physical constraints involved in the design of circuits. Power refers to the power consumption of the chip, area refers to the physical resources used, and performance is the task-specific performance metric. 10, 13, 14

RTL Register transfer level. Design abstraction that models hardware as stateful registers and the logical performed to them every cycle. 11, 13, 53, 71, 79, 108, 112, 199, 224

GLOSSARY

control logic Signals within a circuit that are used to control the flow of computations. 13, 120

datapath The part of a circuit that performs computations. 13, 120

latency-insensitive A circuit that does not have statically known timing behavior. For example, a *variable-latency* divider takes an input-dependent number of cycles to produce an output. 54, 84

latency-sensitive A circuit that has statically known timing behavior. For example, a fully-pipelined, 4-cycle multiplier has a latency of 4 and accepts new inputs every cycle. 54

netlist A low-level, fully structural description of a circuit which specifies hardware instances and connections between them. 11

synchronous Synchronous hardware design uses a global clock signal that is the fastest toggling signal in a circuit. 5, 54

APPENDIX A

DAHLIA SEMANTICS AND SOUNDNESS

A.1 Semantics

The following lists the grammar for Filament, the core language of Dahlia.

$$x \in \text{variables} \quad a \in \text{memories} \quad n \in \text{numbers}$$

$$b ::= \textbf{true} \mid \textbf{false} \qquad v ::= n \mid b \mid v_1 \ \textbf{bop} \ v_2$$

$$e ::= v \mid \textbf{bop}\ e_1\ e_2 \mid x \mid a[e]$$

$$c ::= e \mid \textbf{let}\ x = e \mid c_1 \multimap c_2 \mid c_1 \overset{\rho}{\sim} c_2 \mid c_1\ ;\ c_2 \mid \textbf{if}\ x\ c_1\ c_2 \mid$$
$$\textbf{while}\ x\ c \mid x := e \mid a[e_1] := e_2 \mid \textbf{skip}$$

$$\tau ::= \textbf{bit}\langle n \rangle \mid \textbf{float} \mid \textbf{bool} \mid \textbf{mem}\ \tau[n_1]$$

The large-step operational semantics listed below capture the complete evaluation of an expression or command. The environment σ maps variables and memory names to values, and the context ρ is the set of memories the program has accessed.

$$\boxed{\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v} \qquad ()$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, v_1 \qquad \sigma_2, \rho_2, e_2 \Downarrow \sigma_3, \rho_3, v_2 \qquad v_3 = v_1\ \textbf{bop}\ v_2}{\sigma_1, \rho_1, \textbf{bop}\ e_1\ e_2 \Downarrow \sigma_3, \rho_3, v_3} \qquad \frac{\sigma(x) = v}{\sigma, \rho, x \Downarrow \sigma, \rho, v}$$

$$\frac{a \notin \rho_1 \qquad \sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2(a)(n) = v}{\sigma_1, \rho_1, a[e] \Downarrow \sigma_2, \rho_2 \cup \{a\}, v}$$

$\boxed{\sigma_1, \rho_1, c \Downarrow \sigma_2, \rho_2}$ *()*

$$\frac{\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v}{\sigma_1, \rho_1, \mathbf{let}\ x = e \Downarrow \sigma_2[x \mapsto v], \rho_2} \qquad \frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \mathrel{--} c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \overset{\rho}{\sim} c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3} \qquad \frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1\ ;\ c_2 \Downarrow \sigma_3, \rho_3}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{true} \qquad \sigma_2, \rho_2, c_1 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, \mathbf{if}\, x\, c_1\, c_2 \Downarrow \sigma_3, \rho_3}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{false} \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, \mathbf{if}\, x\, c_1\, c_2 \Downarrow \sigma_3, \rho_3} \qquad \frac{\begin{array}{c}\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{true} \\ \sigma_2, \rho_2, c \mathrel{--} \mathbf{while}\, x\, c \Downarrow \sigma_3, \rho_3\end{array}}{\sigma_1, \rho_1, \mathbf{while}\, x\, c \Downarrow \sigma_3, \rho_3}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{false}}{\sigma_1, \rho_1, \mathbf{while}\, x\, c \Downarrow \sigma_2, \rho_2} \qquad \frac{\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v}{\sigma_1, \rho_1, x := e \Downarrow \sigma_2[x \mapsto v], \rho_2}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2, \rho_2, e_2 \Downarrow \sigma_3, \rho_3, v \qquad a \notin \rho_3}{\sigma_1, \rho_1, a[e_1] := e_2 \Downarrow \sigma_3[a[n] \mapsto v], \rho_3 \cup \{a\}}$$

The small-step operational semantics capture incremental evaluation of an expression or command and form the basis of the proof of soundness in Appendix A.2.

$\boxed{\sigma, \rho, e \to \sigma', \rho', e'}$ *()*

$$\frac{\sigma, \rho, e \to \sigma', \rho', e'}{\sigma, \rho, a[e] \to \sigma', \rho', a[e']} \qquad \frac{a \notin \rho}{\sigma, \rho, a[n] \to \sigma, \rho \cup \{a\}, v} \qquad \frac{\sigma, \rho, e_1 \to \sigma', \rho', e_1'}{\sigma, \rho, \mathbf{bop}\ e_1\ e_2 \to \sigma', \rho', \mathbf{bop}\ e_1'\ e_2}$$

$$\frac{\sigma, \rho, e_2 \to \sigma', \rho', e_2'}{\sigma, \rho, \mathbf{bop}\ v_1\ e_2 \to \sigma', \rho', \mathbf{bop}\ v_1\ e_2'} \qquad \frac{v_3 = v_1\ \mathbf{bop}\ v_2}{\sigma, \rho, \mathbf{bop}\ v_1\ v_2 \to \sigma, \rho, v_3} \qquad \frac{\sigma(x) = v}{\sigma, \rho, x \to \sigma, \rho, v}$$

$\boxed{\sigma_1, \rho_1, c \to \sigma', \rho', c'}$ ()

$$\frac{\sigma, \rho, e_1 \to \sigma', \rho', e_1'}{\sigma, \rho, a[e_1] := e_2 \to \sigma', \rho', a[e_1'] := e_2} \qquad \frac{\sigma, \rho, e \to \sigma', \rho', e'}{\sigma, \rho, a[n] := e \to \sigma', \rho', a[n] := e'}$$

$$\frac{a \notin \rho}{\sigma, \rho, a[n] := v \to \sigma[a[n] \mapsto v], \rho \cup \{a\}, \mathbf{skip}} \qquad \frac{\sigma, \rho, e \to \sigma', \rho', e'}{\sigma, \rho, \mathbf{let}\ x = e \to \sigma', \rho', \mathbf{let}\ x = e'}$$

$$\frac{}{\sigma, \rho, \mathbf{let}\ x = v \to \sigma[x \mapsto v], \rho, \mathbf{skip}} \qquad \frac{\sigma, \rho, c_1 \to \sigma', \rho', c_1'}{\sigma, \rho, c_1\ ;\ c_2 \to \sigma', \rho', c_1'\ ;\ c_2}$$

$$\frac{}{\sigma, \rho, \mathbf{skip}\ ;\ c_2 \to \sigma, \rho, c_2} \qquad \frac{}{\sigma, \rho, c_1 \multimap c_2 \to \sigma, \rho, c_1 \overset{\rho}{\sim} c_2}$$

$$\frac{\sigma, \rho, c_1 \to \sigma', \rho', c_1'}{\sigma, \rho, c_1 \overset{\rho''}{\sim} c_2 \to \sigma', \rho', c_1' \overset{\rho''}{\sim} c_2} \qquad \frac{\sigma, \rho'', c_2 \to \sigma', \rho''', c_2'}{\sigma, \rho, \mathbf{skip} \overset{\rho''}{\sim} c_2 \to \sigma', \rho, \mathbf{skip} \overset{\rho'''}{\sim} c_2'}$$

$$\frac{}{\sigma, \rho, \mathbf{skip} \overset{\rho''}{\sim} \mathbf{skip} \to \sigma, \rho \cup \rho'', \mathbf{skip}} \qquad \frac{\sigma(x) = \mathbf{true}}{\sigma, \rho, \mathbf{if}\, x\, c_1\ c_2 \to \sigma, \rho, c_1}$$

$$\frac{\sigma_1(x) = \mathbf{false}}{\sigma, \rho, \mathbf{if}\, x\, c_1\ c_2 \to \sigma, \rho, c_2} \qquad \frac{}{\sigma, \rho, \mathbf{while}\, x\, c \to \sigma, \rho, \mathbf{if}\, x\, (c \multimap \mathbf{while}\, x\, c)\, \mathbf{skip}}$$

To enforce Dahlia's safety condition, the typing judgments use the typing context Γ for variables and the affine context Δ for memories.

$\boxed{\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2}$ *()*

$$\frac{}{\Gamma, \Delta \vdash v : \tau \dashv \Delta} \qquad \frac{\Gamma, \Delta_1 \vdash e_1 : \tau \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash e_2 : \tau \dashv \Delta_3 \qquad \mathbf{bop} : \tau \to \tau \to \tau}{\Gamma, \Delta_1 \vdash \mathbf{bop}\ e_1\ e_2 : \tau \dashv \Delta_3}$$

$$\frac{\Gamma(x) = \tau}{\Gamma, \Delta_1 \vdash x : \tau \dashv \Delta_1} \qquad \frac{\Gamma, \Delta_1 \vdash e_1 : \mathbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Delta_2 = \Delta_3 \cup \{a \mapsto \mathbf{mem}\,\tau[n_1]\}}{\Gamma, \Delta_1 \vdash a[e] : \tau \dashv \Delta_3}$$

$\boxed{\Gamma_1, \Delta_1 \vdash c \dashv \Gamma_2, \Delta_2}$ *()*

$$\frac{}{\Gamma, \Delta \vdash \mathbf{skip} \dashv \Gamma, \Delta} \qquad \frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : \mathbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash e_2 : \tau \dashv \Delta_3 \\ \Delta_3 = \Delta_4 \cup \{a \mapsto \mathbf{mem}\,\tau[n_1]\}\end{array}}{\Gamma, \Delta_1 \vdash a[e_1] := e_2 \dashv \Gamma, \Delta_4}$$

$$\frac{\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2 \qquad (x \to \tau) \notin \Gamma}{\Gamma, \Delta_1 \vdash \mathbf{let}\ x = e \dashv \Gamma[x \mapsto \tau], \Delta_2} \qquad \frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1\ ;\ c_2 \dashv \Gamma_3, \Delta_3}$$

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_1 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \multimap c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \bar{\rho} \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \overset{\rho}{\sim} c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash x : \mathbf{bool} \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash c_1 \dashv \Gamma_2, \Delta_3 \\ \Gamma, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_4\end{array}}{\Gamma, \Delta_1 \vdash \mathbf{if}\, x\, c_1\, c_2 \dashv \Gamma, \Delta_2 \cap \Delta_3 \cap \Delta_4}$$

$$\frac{\Gamma, \Delta_1 \vdash x : \mathbf{bool} \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash c_1 \dashv \Gamma_2, \Delta_3 \qquad \Gamma, \Delta_3 \vdash c_2 \dashv \Gamma_3, \Delta_4}{\Gamma, \Delta_1 \vdash \mathbf{if}\, x\, c_1\, c_2 \dashv \Gamma, \Delta_4} \qquad \frac{\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2 \qquad \Gamma(x) = \tau}{\Gamma, \Delta_1 \vdash x := e \dashv \Gamma, \Delta_2}$$

$$\frac{\Gamma, \Delta_1 \vdash x : \mathbf{bool} \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash c \dashv \Gamma_3, \Delta_3}{\Gamma, \Delta_1 \vdash \mathbf{while}\, x\, c \dashv \Gamma, \Delta_3 \cap \Delta_2}$$

A.2 Proof of soundness

If there exists a typing context Γ and affine memory context Δ under which a command c type-checks, and Γ, Δ is equivalent to an environment σ and context ρ, then either $\sigma, \rho, c \to^* \sigma', \rho', \mathbf{skip}$ or c diverges.

To prove this theorem, we will prove the supporting progress and preservation lemmas (stated below), which together imply soundness.

Supporting definitions

- Defined: a is *defined* in Δ if $\exists\, \tau, n$ with $(a \mapsto \mathbf{mem}\, \tau[n]) \in \Delta$. x is *defined* in Γ if $\exists\, \tau$ with $(x \mapsto \tau) \in \Gamma$.
- Type-check: If $\Gamma, \Delta \vdash e : \tau \dashv \Delta'$ then e *type-checks to* τ *under* Γ, Δ *producing* Δ'. If $\Gamma, \Delta \vdash c \dashv \Gamma', \Delta'$ then c *type-checks under* Γ, Δ *producing* Γ', Δ'.
- $\sim$ (equivalence): $\Gamma, \Delta \sim \sigma, \rho$ if
    1. $\forall\, x$ with $(x \mapsto \tau) \in \Gamma, \exists\, v$ with $(x \mapsto v) \in \sigma$ and v type-checks to τ under Γ, Δ

2. $\forall\, l$ with $(l \mapsto \tau) \in \Delta, l \notin \rho$.

- $\underline{\bar{\rho}}$: $\bar{\rho} = \{a \mapsto \textbf{mem}\ \tau[n] \in \Delta^* \wedge a \notin \rho\}$ where Δ^* is the affine context of memories initially available to a program.
- Construction: Γ, Δ can be *constructed* from σ, ρ if

  1. $\forall\, (x \mapsto v) \in \sigma, (x \mapsto \tau) \in \Gamma$
  2. $\forall\, l \in \rho, (l \mapsto \textbf{mem}\ \tau[n]) \notin \Delta$
  3. v type-checks to τ under Γ, Δ

Supporting lemmas

- L1: If $\Gamma, \Delta \vdash c \dashv \Gamma', \Delta'$, then $\Gamma \subseteq \Gamma'$. Proof: The only typing rule that modifies Γ is CHECK_LET. Under this rule, $\Gamma' = \Gamma$ extended to add a mapping for a variable x. There is no rule that removes mapping from Γ. So $\forall m \in \Gamma, m \in \Gamma'$.
- L2: If c type-checks under Γ, Δ, then c type-checks under Γ', Δ where $\Gamma \subseteq \Gamma'$. Proof: The only typing rule that reads Γ is CHECK_UPDATE, which checks in its premises that x is defined in Γ. By L1, if x is defined in Γ it is defined in Γ'. There is also no rule that changes the type τ of x in Γ, so x will have the same type τ in Γ'.
- L3: If $\sigma, \rho, e \to \sigma', \rho', e'$, then $\sigma = \sigma'$. Proof: There is no step rule for expressions that extends σ, which by the grammar is the only modification possible to memory stores.
- L4: If $\sigma, \rho, e \to \sigma', \rho', e'$ and $\rho' \neq \rho$, then e is a read $a[n]$ and $\rho' = \rho \cup \{a\}$. Proof: the only step rule for expressions that adds elements to ρ is READ2, by which $\rho' = \rho \cup \{a\}$. There is no step rule that removes elements from ρ.

A.2.1 Progress

If $\exists\, \Gamma, \Delta, \sigma, \rho$ such that $\Gamma, \Delta \sim \sigma, \rho$ and command c type-checks under Γ, Δ, then either c is a value or

1. $\exists\, \sigma', \rho', c'$ with $\sigma, \rho, c \to \sigma', \rho', c'$ or
2. $c = \textbf{skip} \overset{\rho}{\sim} c_2$ with $c_2 \neq \textbf{skip}$ and $\exists\, c_2', \rho''$ with $\sigma, \rho'', \textbf{skip} \overset{\rho}{\sim} c_2 \to \sigma', \rho'', \textbf{skip} \overset{\rho'}{\sim} c_2'$.

Proof

Inductive hypothesis: Progress holds for sub-forms of any inductive form. Assumptions: $\Gamma, \Delta \sim \sigma, \rho$ and c type-checks under Γ, Δ.

Case: c is an expression.

- Case: c is a value. Progress holds by assumption.
- Case: $c = \textbf{bop}\ e_1\ e_2$. For simplicity, we ignore the cases in which **bop** is incompatible with the types of e_1 and e_2. We have three possibilities:
    1. Neither e_1 nor e_2 is a value. By assumption c type-checks under Γ, Δ, so by CHECK_BOP e_1 type-checks under Γ, Δ. By the inductive hypothesis, $\sigma, \rho, e_1 \to \sigma', \rho', e_1'$ so we have $\sigma, \rho, \textbf{bop}\ e_1\ e_2 \to \textbf{bop}\ e_1'\ e_2$ as needed.
    2. Only e_1 is a value v_1. By assumption c type-checks under Γ, Δ and $\Gamma, \Delta \vdash v_1 \dashv \Delta$, so e_2 type-checks under Γ, Δ. By the inductive hypothesis, $\sigma, \rho, e_2 \to \sigma', \rho', e_2$ so $\sigma, \rho, \textbf{bop}\ v_1\ e_2 \to \sigma', \rho', \textbf{bop}\ v_1\ e_2'$.

3. Both e_1 and e_2 are values v_1 and v_2. $\sigma, \rho, \mathbf{bop}\ v_1\ v_2 \rightarrow \sigma, \rho, v_1\ \mathbf{bop}\ v_2$ by BOP3.

- Case: $c = x$. By assumption x type checks under Γ, Δ, so x is defined in Γ. Then $\exists\, v$ with $(x \mapsto v) \in \sigma$, so $\sigma, \rho, x \rightarrow \sigma, \rho, v$ by VAR.
- Case: $c = a[e]$. By assumption $a[e]$ type-checks under Γ, Δ, so by CHECK_READ, e type-checks under Γ, Δ producing Δ_2, and a is defined in Δ_2. By the the inductive hypothesis, progress holds for e, so $\sigma, \rho, e \rightarrow \sigma', \rho', e'$ or e is a value n. a is defined in Δ_2, so it must be defined in Δ since there is no type-checking rule for expressions by which $\Gamma, \Delta \vdash e \dashv \Delta_2$ and $\exists\, l \notin \Delta, \in \Delta_2$. So $a \notin \rho$. So if e is a value, then $\sigma, \rho, a[e] \rightarrow \sigma, \rho \cup \{a\}, \sigma(a)(n)$. If e is not a value, then $\sigma, \rho, a[e] \rightarrow \sigma', \rho', a[e']$.

Case: $c = \mathbf{let}\ x = e$. By assumption this form type-checks under Γ, Δ. By CHECK_LET, e type-checks under Γ, Δ. By the inductive hypothesis, e is either a value v or $\sigma, \rho, e \rightarrow \sigma', \rho', e'$. In the first case we have $\sigma, \rho, \mathbf{let}\ x = v \rightarrow \sigma[x \mapsto v], \rho, \mathbf{skip}$. In the second case we have $\sigma, \rho, \mathbf{let}\ x = e \rightarrow \sigma', \rho', \mathbf{let}\ x = e'$.

Case: $c = c_1 \text{ —— } c_2$. $\forall\, \sigma, \rho,\ \sigma, \rho, c_1 \text{ —— } c_2 \rightarrow \sigma, \rho, c_1 \overset{\rho}{\sim} c_2$.

Case: $c = c_1 \overset{\rho''}{\sim} c_2$. We have three possibilities:

- $c_1 \neq \mathbf{skip}$. By assumption c type-checks under Γ, Δ. By CHECK_INTER_SEQ_COMP, c_1 type-checks under Γ, Δ. By the inductive hypothesis, $\sigma, \rho, c_1 \rightarrow \sigma', \rho', c_1'$ and so $c_1 \overset{\rho''}{\sim} c_2 \rightarrow c_1' \overset{\rho''}{\sim} c_2$.
- $c = \mathbf{skip} \overset{\rho}{\sim} c_2$. By assumption c type-checks under Γ, Δ. By CHECK_INTER_SEQ_COMP, c_2 type-checks under $\Gamma, \bar{\rho}$. By the inductive hypothesis (and the definition of

$\sim$, under which we have ρ), $\sigma, \rho, c_2 \to \sigma', \rho', c_2'$, so we have $\sigma, \rho'', \mathbf{skip} \overset{\rho}{\sim} c_2 \to \sigma', \rho'', \mathbf{skip} \overset{\rho'}{\sim} c_2'$.

- If $c_1 = c_2 = \mathbf{skip}$, $\forall\, \sigma, \rho$, we have $\sigma, \rho, c_1 \overset{\rho''}{\sim} c_2 \to \sigma, \rho \cup \rho'', \mathbf{skip}$.

Case: $c = c_1; c_2$. We have two possibilities:

- $c_1 \neq \mathbf{skip}$. By assumption, c type-checks under Γ, Δ, so c_1 type-checks under Γ, Δ by CHECK_PAR_COMP. By the the inductive hypothesis, $\sigma, \rho, c_1 \to \sigma', \rho', c_1'$, so $\sigma, \rho, c_1; c_2 \to \sigma', \rho', c_1'; c_2$.
- $c_1 = \mathbf{skip}$. $\forall\, \sigma, \rho$, we have $\sigma, \rho, \mathbf{skip}; c_2 \to \sigma, \rho, c_2$.

Case: $c = \mathbf{if}\ x\ c_1\ c_2$. By assumption, c type-checks under Γ, Δ. Then x type-checks to bool, so x is either **true** or **false**. $\forall\, \sigma, \rho$, we have $\sigma, \rho, \mathbf{if\ true}\ c_1\ c_2 \to \sigma, \rho, c_1$ and $\sigma, \rho, \mathbf{if\ false}\ c_1\ c_2 \to \sigma, \rho, c_2$.

Case: $c = \mathbf{while}\ x\ c_1$. $\forall\, \sigma, \rho$ we have
$\sigma, \rho, \mathbf{while}\ x\ c_1 \to \sigma, \rho, \mathbf{if}\ x\ (c_1 \text{ —— } \mathbf{while}\ x\ c_1)\ \mathbf{skip}$.

Case: $c = x := e$. If e is a value v, then $\forall\, \sigma, \rho$ we have $\sigma, \rho, x := e \to \sigma[x \mapsto v], \rho, \mathbf{skip}$. If e is not a value, then by assumption that c type-checks under Γ, Δ, e type-checks under Γ, Δ by CHECK_UPDATE. Then by the inductive hypothesis, $\sigma, \rho, e \to \sigma', \rho', e'$, so $\sigma, \rho, x := e \to \sigma', \rho', x := e'$.

Case: $c = a[e_1] := e_2$. We have three possibilities:

- e_1 and e_2 are values n and v. By assumption c type-checks under Γ, Δ, so by

CHECK_WRITE, a is defined in Δ. By definition of $\sim$, $a \notin \rho$, so the premise of WRITE3 is satisfied. Then $\sigma, \rho, c \to \sigma[a[n] \mapsto v], \rho \cup \{a\}, \mathbf{skip}$.

- e_1 is a value n. By assumption c type-checks under Γ, Δ, so by CHECK_WRITE n type-checks under Γ, Δ producing Δ and e_2 type-checks under Γ, Δ. By the the inductive hypothesis, $\sigma, \rho, e_2 \to \sigma', \rho', e_2'$, so $\sigma, \rho, a[n] := e_2 \to \sigma', \rho', a[n] := e_2'$.
- Neither e_1 nor e_2 is a value. By assumption c type-checks under Γ, Δ, so as does e_1. By the the inductive hypothesis, $\sigma, \rho, e_1 \to \sigma', \rho', e_1'$, so $\sigma, \rho, a[e_1] := e_2 \to \sigma', \rho', a[e_1'] := e_2$.

A.2.2 Preservation

If:

1. $\exists\, \Gamma, \Delta$ such that command c type-checks under Γ, Δ
2. $\exists\, \sigma, \rho$ with $\Gamma, \Delta \sim \sigma, \rho$
3. $\exists\, \sigma'\rho', c'$ with $\sigma, \rho, c \to \sigma', \rho', c'$ or $\exists\, \sigma', \rho', \rho'', c_2'$ with $c = \mathbf{skip} \overset{\rho}{\sim} c_2$ and $\sigma, \rho'', \mathbf{skip} \overset{\rho}{\sim} c_2 \to \sigma', \rho'', \mathbf{skip} \overset{\rho'}{\sim} c_2'$

then Γ', Δ' can be constructed from σ', ρ' such that c' type-checks under Γ', Δ'.

Proof

Inductive hypothesis: Preservation holds for sub-forms of any inductive form. Assumptions: 1., 2., 3.

Case: c is an expression.

- Case: c is a value. c does not step, so preservation vacuously holds.
- Case: $c = \textbf{bop}\ e_1\ e_2$. For simplicity, we ignore the cases in which **bop** is incompatible with the types of e_1 and e_2. We have three possibilities:

  1. e_1 is not a value. By assumption, c type-checks under Γ, Δ and $\sigma, \rho, c \to \sigma', \rho', \textbf{bop}\ e_1'\ e_2$. So $\sigma, \rho, e_1 \to \sigma', \rho', e_1'$. By CHECK_BOP $\Gamma, \Delta \vdash e_1 \dashv \Delta_2$. By the inductive hypothesis, $\Gamma', \Delta' \vdash e_1' \dashv \Delta_2'$. From L3, $\sigma' = \sigma$, so $\Gamma' = \Gamma$. If $\Delta' = \Delta$, then e_2 type-checks under Γ', Δ_2' and we are done. If $\Delta' \neq \Delta$, then $\rho' \neq \rho$. So by L4 e_1 was a read $a[n]$. By assumption and CHECK_BOP $\Gamma, \Delta \vdash e_1 \dashv \Delta_2$ and e_2 type-checks under Γ, Δ_2. Since $e_1 = a[n], a$ could not have been defined in Δ_2. e_1' must be a value, so $\Gamma', \Delta' \vdash v \dashv \Delta'$, so $\Delta' = \Delta_2$. So e_2 must type-check under Γ', Δ'.
  2. e_1 is a value v_1. Then by assumption, $\sigma, \rho, \textbf{bop}\ v_1\ e_2 \to \sigma', \rho', \textbf{bop}\ v_1\ e_2'$. By assumption, c type-checked under Γ, Δ, so e_2 type-checks under Γ, Δ by CHECK_BOP. By the inductive hypothesis, e_2' type-checks under Γ', Δ', and values always type-check, so we are done.
  3. Both e_1 and e_2 are values v_1 and v_2. By assumption, $\sigma, \rho, c \to \sigma, \rho, v_1 \textbf{bop} v_2$. Values always type-check, so we are done.

- Case: $c = x$. By assumption x type-checks under Γ, Δ, so $(x \mapsto \tau) \in \Gamma$ and $(x \mapsto v) \in \sigma$, and by assumption $\sigma, \rho, x \to \sigma, \rho, v$. Values always type-check, so we are done.
- Case: $c = a[e]$. The first possibility is that e is not a value. Then by assumption $\sigma, \rho, a[e] \to \sigma', \rho', a[e']$, and so $\sigma, \rho, e \to \sigma', \rho', e'$. We need to show that $a[e']$ type-checks under Γ', Δ'. By the inductive hypothesis, e' type-checks

under Γ', Δ'. To satisfy the second premise, it should be that a is defined in Δ'. By assumption that $a[e]$ type-checked, we know from CHECK_READ that a is defined in Δ and so $a \notin \rho$ (by definition of $\sim$). If a was not defined in Δ', that would mean $a \in \rho'$, but by L4 this would mean that e was a read $a[n]$, meaning a is not defined in Δ_2 where $\Gamma, \Delta \vdash e \dashv \mathit{Delta}_2$ and c could not type-check under Γ, Δ - this is a contradiction. So a must be defined in Δ' and so $a[e']$ must type-check under Γ', Δ'. The second possibility is that e is a value n. Then $\sigma, \rho, a[n] \to \sigma, \rho, v$ and since values always type-check we are done.

Case: $c = \textbf{let}\ x = e$. The first possibility is that e is not a value. By assumption c type-checks under Γ, Δ. By CHECK_LET, so does e. By assumption $\sigma, \rho, \textbf{let}\ x = e \to \sigma, \rho, \textbf{let}\ x = e'$, so $\sigma, \rho, e \to \sigma', \rho', e'$. By the inductive hypothesis, e' type-checks under Γ', Δ'. Then we have that $\textbf{let}\ x = e'$ type-checks under Γ', Δ', so we are done. The second possibility is that e is a value v. In this case $\sigma, \rho, \textbf{let}\ x = v \to \sigma[x \mapsto v], \rho, \textbf{skip}$. $\textbf{skip}$ always type-checks, so we are done.

Case: $c = c_1 \text{ —— } c_2$. By assumption, $\sigma, \rho, c_1 \text{ —— } c_2 \to \sigma, \rho, c_1 \overset{\rho}{\sim} c_2$ and c type-checks under Γ, Δ. $\sigma', \rho' = \sigma, \rho$, so $\Gamma' = \Gamma$ and $\Delta' = \Delta$. By assumption $\Gamma, \Delta \vdash c_1 \dashv \Gamma_2, \Delta_2$ and c_2 type-checks under Γ_2, Δ. We need to show that c_2 type-checks under $\Gamma_2, \bar{\rho}$. Since c_2 type-checks under Γ_2, Δ, it does not use any memories in ρ by definition of $\sim$. So it must type-check under $\Gamma_2, \bar{\rho}$.

Case: $c = c_1 \overset{\rho''}{\sim} c_2$. We have three possibilities.

- Neither c_1 nor c_2 is $\textbf{skip}$. In this case, we have $\sigma, \rho, c_1 \overset{\rho''}{\sim} c_2 \to \sigma', \rho', c_1' \overset{\rho''}{\sim} c_2$ and $\sigma, \rho, c_1 \to \sigma', \rho', c_1'$. From assumption, c type-checks under Γ, Δ, so 1)

c_2 type-checks under $\bar{\rho''}$ and 2) by the inductive hypothesis, c_1' type-checks under the constructed Γ', Δ'. We need to show c_2 type-checks under $\Gamma', \bar{\rho''}$. By L2, if c_2 type-checks under $\Gamma, \bar{\rho''}$, it will type-check under $\Gamma', \bar{\rho''}$, so we are done.

- $c_1 = \textbf{skip} \neq c_2$. We have that $\sigma, \rho'', \textbf{skip} \overset{\rho}{\sim} c_2 \rightarrow \sigma', \rho'', \textbf{skip} \overset{\rho'}{\sim} c_2'$, so $\sigma, \rho, c_2 \rightarrow \sigma', \rho', c_2'$. By the inductive hypothesis, c_2' type-checks under the constructed Γ', Δ' (so it will type-check under $\Gamma', \bar{\rho'}$) and **skip**always type-checks, so we are done.
- $c_1 = c_2 = \textbf{skip}$. This form steps to **skip**, which always type-checks, so we are done.

Case: $c = c_1; c_2$. We have two possibilities:

- $c_1 \neq \textbf{skip}$. By assumption $\sigma, \rho, c \rightarrow \sigma', \rho', c'$, so $\sigma, \rho, c_1 \rightarrow \sigma', \rho', c_1'$. We have by assumption that c_1 type-checks under Γ, Δ to produce Γ_2, Δ_2, and c_2 type-checks under Γ_2, Δ_2. By the inductive hypothesis, c_1' type-checks under Γ', Δ' to produce Γ_2', Δ_2'. We need to show that c_2 type-checks under Γ_2', Δ_2'. We have two possibilities: $\rho' = \rho$ and $\rho' \neq \rho$. Consider the first possibility. We would have $\Delta = \Delta'$, so $\Delta_2 = \Delta_2'$. By L2, c_2 type-checks under Γ', Δ_2' as needed. With the second possibility, it can only be that $\rho \subset \rho'$. There are then only two cases to consider:

    1. c_1 contained a read or write involving $a[n]$ and c_1' is a value v. Then $\rho' = \rho \cup \{a\}$ and a is not defined in Δ'. By CHECK_WRITE and CHECK_READ, a could not have been defined in Δ_2. Since $\Gamma', \Delta' \vdash v \dashv \Delta'$, $\Delta_2 = \Delta'$. So c_2 must type-check under Γ', Δ'.

2. $c_1 = \textbf{skip} \overset{\rho''}{\sim} \textbf{skip}$. By INTER_SEQ3 $c_1' = \textbf{skip}$ and $\sigma' = \sigma$, so $\Gamma' = \Gamma$. By assumption c_2 type-checks under Γ_2, Δ_2 where $\Gamma, \Delta \vdash c_1 \dashv \Gamma_2, \Delta_2$. By CHECK_INTER_ SEQ_COMP $\Gamma_2 = \Gamma$.

   We need to show c_2 type-checks under $\Gamma_2', \Delta_2' = \Gamma', \Delta' = \Gamma, \Delta'$ (since $\Gamma, \Delta \vdash \textbf{skip} \dashv \Gamma, \Delta$). For this to be the case, c_2 cannot use any memories in ρ or ρ'' (by definition of construction).

   1) Because $\Gamma, \Delta \sim \sigma, \rho$ and c_1 type-checks under Γ, Δ, c_1 does not use any memories in ρ. Then by assumption that c_2 type-checks under Γ_2, Δ_2 and by CHECK_PAR_COMP, c_2 also cannot use any memories in ρ.

   2) By assumption c_1 type-checks under Γ, Δ to produce Γ_2, Δ_2. By CHECK_INTER_SEQ_COMP and SMALL_SEQ $\Delta_2 \subseteq \bar{\rho''}$, and by assumption c_2 type-checks under Γ_2, Δ_2, so c_2 does not use any memories in ρ''. So c_2 type-checks under Γ', Δ' as needed.

- $c_1 = \textbf{skip}$. By assumption $\textbf{skip}; c_2$ type-checks under Γ, Δ, so c_2 type-checks under Γ, Δ. $\sigma, \rho, \textbf{skip}; c_2 \to \sigma, \rho, c_2$ so $\Gamma' = \Gamma$ and $\Delta' = \Delta$. Then we need to show c_2 type-checks under $\Gamma', \Delta' = \Gamma, \Delta$, which we have from assumption, so we are done.

Case: $c = \textbf{if}\ x\ c_1\ c_2$. By assumption c type-checks under Γ, Δ, so c_1 and c_2 both type-check under Γ, Δ, and x is either **true** or **false** by CHECK_IF. If true, $\sigma, \rho, c \to \sigma, \rho, c_1$. If false, $\sigma, \rho, c \to \sigma, \rho, c_2$. We need to show that c_1 and c_2 type-check under Γ, Δ ($\sigma', \rho' = \sigma, \rho$, so $\Gamma', \Delta' = \Gamma, \Delta$). This is given by assumption so we are done.

Case: $c = \textbf{while}\ x\ c_1$. By assumption c type-checks under Γ, Δ, so by CHECK_WHILE x type-checks to bool and c_1 type-checks under Γ, Δ to produce Γ_2, Δ_2. We need

to show that **if** x (c_1 ;; **while** x c_1) **skip** type-checks under Γ, Δ ($\sigma', \rho' = \sigma, \rho$, so $\Gamma', \Delta' = \Gamma, \Delta$). For this, it should be the case that x type-checks to bool. This is already given. It should also be the case that (c_1 ;; **while** x c_1) type-checks under Γ, Δ. This requires that c_1 type-checks under Γ, Δ, which is given by assumption, and that **while** x c_1 type-checks under Γ_2, Δ. By L2 if **while** x c_1 type-checks under Γ, Δ (which it does by assumption), it type-checks under Γ_2, Δ, so we are done.

Case: $c = x := e$. By assumption c type-checks under Γ, Δ, so $(x \mapsto \tau) \in \Gamma$ and e type-checks under Γ, Δ producing Δ_2. We have two possibilities:

- e is not a value. By assumption and CHECK_UPDATE $\sigma, \rho, x := e \to \sigma', \rho', x := e'$ and $\sigma, \rho, e \to \sigma', \rho', e'$. By the inductive hypothesis, e' type-checks under the constructed Γ', Δ'. Then by L1 and L2, if $\Gamma, \Delta \vdash e : \tau \dashv \Delta_2$ then $\Gamma', \Delta' \vdash e' : \tau; \dashv \Delta_2'$. So $(x \mapsto \tau) \in \Gamma'$ and c' type-checks under Γ', Δ' as needed.
- e is a value v. By assumption $x := v$ type-checks under Γ, Δ so $\Gamma, \Delta \vdash v : \tau \dashv \Delta$ and $(x \mapsto \tau) \in \Gamma$. $\sigma, \rho, x := v \to \sigma[x \mapsto v], \rho,$ **skip**, and **skip** always type-checks, so we are done.

Case: $c = a[e_1] := e_2$. By assumption c type-checks under Γ, Δ, so e_1 type-checks under Γ, Δ producing Δ_2 and e_2 type-checks under Γ, Δ_2 by CHECK_WRITE. Additionally, a is defined in Δ and Δ_2 so neither e_1 nor e_2 use memory a. We then have three possibilities:

- Neither e_1 nor e_2 is a value. Then $\sigma, \rho, a[e_1] := e_2 \to \sigma', \rho', a[e_1'] := e_2$ and $\sigma, \rho, e_1 \to \sigma', \rho', e_1'$. By the inductive hypothesis, e_1' type-checks under the

constructed Γ', Δ' to produce Δ'_2. Either $\rho' = \rho$ or not. If $\rho' = \rho$, then by L2, e_2 will type-check under Γ', Δ'_2 since $\Gamma \subseteq \Gamma'$ and $\Delta' = \Delta$. If not, then by L4 e_1 is a read $a_1[n]$. By assumption and CHECK_WRITE $\Gamma, \Delta \vdash e_1 \dashv \Delta_2$ and e_2 type-checks under Γ, Δ_2. Since $e_1 = a_1[n], a_1$ could not have been defined in Δ_2. e'_1 then must be a value, so $\Gamma', \Delta' \vdash v \dashv \Delta'$, so $\Delta_2 = \Delta'$. So e_2 must type-check under Γ', Δ'.

- e_1 is a value n and e_2 is not a value. Then $\sigma, \rho, a[e_1] := e_2 \to \sigma', \rho', a[e_1] := e'_2$ and $\sigma, \rho, e_2 \to \sigma', \rho', e'_2$. By the inductive hypothesis, e_2 type-checks under Γ', Δ'. e_1 is a value and always type-checks, so we are done.
- e_1 is a value n and e_2 is a value v. Assuming this type-checks, we step to **skip**, which always type-checks, so we are done.

APPENDIX B

FILAMENT SEMANTICS AND SOUNDNESS

B.1 Syntax

Figure B.1 presents the syntax of a desugared version of Filament that only allows parameterization of components using one event. Neither simplification loses generality because user-level components with multiple events cannot define any form of interaction between them—they are functionally equivalent to multiple components with disjoint events. Filament's external components can express more behaviors than this formalism allows for and are therefore not covered by this formalism.

A Filament program is a sequence of components M each of which encapsulates the structure and schedule of a pipeline. Commands c include port connections, component instantiation, invocations. Component are parameterized using exactly one event and invocations allow scheduling using one event.

$$
\begin{aligned}
&x, C \in \mathit{vars} \quad t \in \mathit{events} \quad p, q \in \mathit{ports} \\
&\quad T ::= t \mid T + n \quad \pi ::= [T_1, T_2] \\
&\quad M ::= \textbf{def}\ C\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)\{c\} \\
&\quad c ::= c_1 \bullet c_2 \mid p_d = p_s \mid x := \textbf{new}\ C \mid x := \textbf{inv}\ x\langle T\rangle(p_1, \ldots, p_j)
\end{aligned}
$$

Figure B.1: Syntax of desugared Filament programs.

$$
\begin{aligned}
&[\![c]\!] && : \mathcal{L}og \to \mathcal{L}og \qquad (\mathcal{L}og = \mathcal{T} \to \mathcal{R} \times \mathcal{W}) \\
&[\![x := \textbf{new}\, C]\!] && = \mathrm{id} \\
&[\![p_d = p_s]\!] && = \lambda(R, W).\, \mathrm{if}\, p_s \in W\, \mathrm{then}\, (R\{p_s/p_d\}, W)\, \mathrm{else}\, (R, W) \\
&[\![c_1 \bullet c_2]\!] && = [\![c_1]\!] \cup [\![c_2]\!] \\
&[\![x_1 := \textbf{inv}\, x_2\langle T'\rangle(q_1, \cdots, q_m)]\!] && = [\![\mathrm{connects}(x_1, [q_1, \ldots, q_m])]\!] \circ [\![x_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\![M]\!] && : \mathcal{P} \\
&[\![\textbf{def}\, C\langle t : n\rangle(i_0 : \pi_0, \ldots, o_0 : \pi_i, \ldots)\{c\}]\!] && = \{\pi_0 \mapsto i_0, \ldots\} \times \{\pi_i \mapsto o_0, \ldots\}
\end{aligned}
$$

Figure B.2: Log-transfomer semantics for Filament's core language. Each command produces a log ($\mathcal{L}$) which maps events ($\mathcal{T}$) to multisets of reads ($\mathcal{R}$) and writes ($\mathcal{W}$). Component definitions produce partial logs ($\mathcal{P}$) by mapping availabilities of inputs to reads and availabilities of outputs to writes.

B.2 Semantics

The basis of our semantics is logs of reads and writes. Intuitively, every command generates a function from events to (multi)sets of reads and writes indicating which ports were read or written to at that particular event. More formally, every command is interpreted as a function over logs as presented in Figure B.2 which provides a denotation of Filament programs as a log transformer ($\mathcal{L}$) of events to multisets of port reads ($\mathcal{R}$) and port writes ($\mathcal{W}$). Since components are allowed to use exactly one event, say T, the log maps events such as T, $T + 1$, etc. to reads and writes to ports defined by the subcomponents. Instantiation does not affect the logs, while connections rewrite the logs by adding the LHS port to the set of writes for event where the RHS port is defined. Parallel composition is interpreted as the union of the logs produced by the two commands.

Finally, invocations produce a log-transformer derived from its signature: the input ports are added to the reads ($\mathcal{R}$) and the output ports to the writes ($\mathcal{W}$) for

each event contained in their corresponding availability interval. Note, however, that the semantics given by the signature uses the "incorrect" ports, since it is using the ports given by the signature. We work around by using the connects metafunction which is defined was

$$\text{connects}(x_1, [q_1, \cdots q_m]) = (x_1.p_1 = q_1); \cdots ; (x_1.p_m = q_m),$$

where we are implicitly assuming that the ports $x_1.p_i$ have not been used. Intuitively, this metaprogram coverts the arguments to an invocation into connections and generates a log by interpreting them using the denotation over commands. The semantics of a program is defined by the log produced by a distinguished top-level component.

Components and its semantics For example, a combinational adder and a sequential multiplier with a two-cycle latency produce the following logs:

$$[\![\textbf{def}\,\text{add}\langle G:1\rangle(\text{l}:[G,G+1],\text{r}:[G,G+1],\text{out}:[G,G+1])]\!] = G \to (\{\,\text{l},\text{r}\,\},\{\,\text{go},\text{out}\,\})$$

$$\begin{aligned}[\![\textbf{def}\,\text{mul}\langle G:2\rangle(\text{l}:[G,G+1],\text{r}:[G,G+1],\text{out}:[G+2,G+3])]\!] = {} & G \to (\{\,\text{l},\text{r}\,\},\{\,\text{go}\,\}) \\ & G+1 \to (\emptyset,\{\,\text{go}\,\}) \\ & G+2 \to (\emptyset,\{\,\text{out}\,\})\end{aligned}$$

Note that the use of the instances is reflected through the writes their interface ports `go` (not shown in the signature). The log indicates that the multiplier accepts new values every 2 cycles by writing to the `go` port in both cycles G and $G+1$. Because we track multisets of reads and writes, we can track conflicting writes to the same port.

Using these semantics, we can define the well-formedness constraint (§7.3.2) on logs:

Definition B.2.1 (Well-Formedness)**.** *A log $\mathcal{L}$ is well-formed if and only if for all events*

- *There are no conflicting writes: $\mathcal{W}_s = \mathcal{W}$ where $\mathcal{W}_s$ is the deduplicated set of writes.*
- *Reads are a subset of writes for every event: $\mathcal{R} \subseteq \mathcal{W}_s$.*

While in real hardware, values are always available on a port or a wire, Filament's semantics only track semantically valid values from a read. Usage of hardware resources is denoted by a write, and it is physically impossible to write two values to a port; instead, the circuit uses a multiplexer to select between the two values. Multiple uses of a resource silently corrupt the data.

The safe pipelining constraints (§7.3.4) can be defined in terms of repeated execution of the semantics of a program:

Definition B.2.2 (Safe Pipelining)**.** *If a component M has an event T with delay d, and $[\![M]\!]_G$ represents its log where T is replaced with the event G, then M is safely pipelined if and only if for every $n \geq d$ the logs $L_n = [\![M]\!]_T \cup [\![M]\!]_{T+n}$ are well-formed.*

B.3 Type System

Our presentation focuses on Filament's substructural type system that is used to track non-conflicting use resources as well as signal validity. We elide the description of features that track things such as port widths which are standard.

$$\tau ::= \forall\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)$$

$$\Gamma ::= \cdot \mid \Gamma, C : \tau \mid \Gamma, p : \pi$$

$$\Delta ::= \cdot \mid \Delta, t : n$$

$$\Lambda ::= \cdot \mid \Lambda, I : \pi \mid \Lambda, p : \pi$$

Typing contexts The typing judgements use the following typing contexts:

- Γ tracks the types for components and instances and availability of ports.
- Δ tracks the delays associated with each event in the context.
- Λ is the *timeline context* and tracks the availability of each instance and port.

The type context (Γ) and timeline context (Λ) store timelines for ports and instances respectively. Timelines for ports are *reusable* since reading a port does not consume it during that cycle. However, the timeline of an instance is *consumed* when it is used in a cycle. Because of this, timeline contexts also provide a *separating union* inspired by separation logic [133].

Splitting timelines A valid separating split of a timeline context $\Lambda = \Lambda_1 * \Lambda_2$ if and only if both Λ_1 and Λ_2 bind all the same instances and for each instance, the timelines are disjoint. Formally:

$$\Lambda = \Lambda_1 * \Lambda_2 \text{ iff } \forall (I : \pi) \in \Lambda \Rightarrow \exists \pi_1, \pi_2 . (I : \pi_1) \in \Lambda_1 \wedge (I : \pi_2) \in \Lambda_2 \wedge \pi_1 \cap \pi_2 = \emptyset \wedge \pi_1 \cup \pi_2 = \pi$$

Instantiating components Instantiating a module binds the signature of the component to the instance and make the resource available throughout the timeline of the program, denoted by $[0, \infty)$.

$$\frac{\begin{array}{c}\Gamma(C) = \tau \\ \Gamma' = \Gamma, I : \tau \\ \Lambda' = \Lambda, I : [0, \infty)\end{array}}{\Delta, \Lambda, \Gamma \vdash I := \textbf{new}\ C \dashv \Lambda', \Gamma'} \quad \textsc{Instantiate}$$

Port connections Connecting ports checks that the source port is available for at least as long as the destination port requires:

$$\frac{\Lambda(p_d) \subseteq \Gamma(p_s)}{\Delta, \Lambda, \Gamma \vdash p_d = p_s \dashv \Lambda, \Gamma} \quad \textsc{ConnectValidRead}$$

Splitting timelines with composition The composition rule splits the timeline context using the separating split operator and checks the two commands. Note that that same type context Γ is used for both commands which means previously defined ports are available in both the commands:

$$\frac{\begin{array}{c}\Delta, \Lambda_1, \Gamma \vdash c_1 \dashv \Lambda_1', \Gamma_1 \\ \Delta, \Lambda_2, \Gamma \vdash c_2 \dashv \Lambda_2', \Gamma_2\end{array}}{\Delta, \Lambda_1 * \Lambda_2, \Gamma \vdash c_1 \bullet c_2 \dashv \Lambda_1' * \Lambda_2', \Gamma_1 \cup \Gamma_2} \quad \textsc{CheckComp}$$

Checking invocations The invocation rule enforces well-formedness and safe-pipelining constraints and is therefore quite verbose. We separate out type checking

of invocations into three sets of premises that logically reflect the properties presented in §7.3.

$$\frac{\text{valid reads} \qquad \text{no conflicts} \qquad \text{safe pipelining}}{\Delta, \Lambda, \Gamma \vdash x := \textbf{invoke} I\langle T \rangle(q_1, .., q_j) \dashv \Lambda, \Gamma''}$$

The first set of premises check that all the reads from all ports mentioned in an invocation are valid, i.e., they are available for at least as long as the instance's signature requires. Finally, invocations bind the availability of all the ports associated with the instance to the type context.

$$\frac{\begin{array}{l}\Gamma(I) = \forall\, \langle t : n \rangle (p_1 : \pi_1, \, .. \, , p_j : \pi_j) \\ \Gamma(q_1) = \pi_1, \, .. \, , \Gamma(q_j) = \pi_j \\ \pi'_1 = \pi_1[t/\,T], \, .. \, , \pi'_j = \pi_j[t/\,T] \\ \Gamma(p_1) \subseteq \pi'_1, \, .. \, , \Gamma(p_j) \subseteq \pi'_j \\ \Gamma' = x.\{p_1 : \pi'_1, \, .. \, , p_j : \pi'_j\} \\ \Gamma'' = \Gamma \cup \Gamma'\end{array}}{\Delta, \Lambda, \Gamma \vdash x := \textbf{invoke}\, I\langle T \rangle(q_1, \, .. \, , q_j) \dashv \Lambda, \Gamma''} \quad \textsc{InvokeValidReads}$$

The next set of premises ensure that the instance is available in the current timeline context. This ensures that there are no conflicting uses of the component anywhere else in the design.

$$\frac{\begin{array}{l}\Gamma(I) = \forall\, \langle t : n \rangle (p_1 : \pi_1, \, .. \, , p_j : \pi_j) \\ \Lambda(I) = \pi \\ [T, \, T + n] \subseteq \pi\end{array}}{\Delta, \Lambda, \Gamma \vdash x := \textbf{invoke}\, I\langle T \rangle(q_1, \, .. \, , q_j) \dashv \Lambda, \Gamma''} \quad \textsc{InvokeNoConflicts}$$

The composition rule is responsible for selecting a valid split to ensure that the above rule's constraints are satisfied. If there is no such split possible, then the program has conflicting uses of the instance.

A final set checks for the safety of pipelining an invocation (§7.3.4):

$$\frac{\begin{array}{l}\Gamma(I) = \forall\, \langle t : n \rangle (p_1 : \pi_1,\, ..\,, p_j : \pi_j) \\ \mathcal{E}\,(T) = t' \\ \Delta(t') \geq \Delta(t)\end{array}}{\Delta, \Lambda, \Gamma \vdash x := \textbf{invoke}\, I\langle T \rangle (q_1,\, ..\,, q_j) \dashv \Lambda, \Gamma''} \quad \textsc{InvokeSafePipeline}$$

B.4 Type Soundness

Our type system guarantees theorem focuses on the well-formedness property (§7.3.2). It states that well-typed commands preserve well-formed logs. A second soundness property of our semantics is that the log transformers generated by well-typed programs may only add available ports to the writes of logs. This is captured by the following theorem.

Lemma B.4.1 (Availability Soundness)**.** *If* $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$*, then for every log* $\mathcal{L}$ *an every event* T*, let* $(\mathcal{R}, \mathcal{W}) = [\![c]\!]\,(\mathcal{L}, T)$ *then* $p \in \mathcal{W}$ *if, and only if,* $p \in \pi_2(\mathcal{L}(T))$ *or* $T \in \Lambda(p)$*.*

Proof. The proof follows by induction on the typing derivation. The first case is trivial, since the identity function maps any log to itself. The port connection case does not modify the write component, which makes it similar to the identity case. The composition operation follows by the inductive hypothesis. For the invocation case consider a port p in the writes of the transformed log and assume that $p \notin \mathcal{L}$. By construction, this p has to be one of the output ports which, by the typing rule, has to be available. The other direction follows by case analysis. □

By specializing the theorem above to the composition $c_1 \bullet c_2$ and by using the fact that, by assumption, the Λ contexts of c_1 and c_2 are disjoint it follows:

Corollary B.4.2 (Disjoint Writes)**.** *If* $\Delta; \Lambda; \Gamma \vdash c_1 \bullet c_2 \dashv \Lambda'; \Gamma'$*, then for logs* $(\mathcal{R}_1, \mathcal{W}_1) = [\![c_1]\!](\mathcal{L})$ *and* $(\mathcal{R}_2, \mathcal{W}_2) = [\![c_2]\!](\mathcal{L})$*,* $\mathcal{W}_1 - \mathcal{L}$ *and* $\mathcal{W}_2 - \mathcal{L}$ *are disjoint.*

Theorem B.4.3 (Soundness Property)**.** *If* $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$ *then if* $\mathcal{L}$ *is well-formed, then* $[\![c]\!] - \mathcal{L}$ *is well-formed as well.*

Proof. The proof follows by induction on the typing derivation of c.

- Case INSTANTIATE: by assumption, $\mathcal{L}$ is well-formed.
- Case CONNECTION: by assumption $\mathcal{L}$ is well-formed, therefore $\mathcal{W}$ is a set. For the first condition, there are two possibilities: either $p_d \in \mathcal{R}$ or $p_d \notin \mathcal{R}$. If the second case holds then, $\mathcal{R}\{p_d \mapsto p_s\} = \mathcal{R} \subseteq \mathcal{W}$. If $p_d \in \mathcal{R}$ then, by assumption of the typing rule, the availability of p_d is a subset of the availability of p_s and, by well-formedness of the input log, $p_s \in \mathcal{W}$, which implies $\mathcal{R}\{p_d \mapsto p_s\} \subseteq \mathcal{W}$.
- Case COMP: by the induction hypothesis $\mathcal{L}_1 = [\![c_1]\!](\mathcal{L})$ and $\mathcal{L}_2 = [\![c_2]\!](\mathcal{L})$ are well-formed, which implies that $\mathcal{R}_1 \subseteq \mathcal{W}_1$ and $\mathcal{R}_2 \subseteq \mathcal{W}_2$. By monotonicity of the union operation, $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \subseteq \mathcal{W}_1 \cup \mathcal{W}_2 = \mathcal{W}$ and the first well-formedness condition holds. Additionally, by corollary B.4.2 the writes for the two logs are disjoint, making $[\![c_1 \cdot c_2]\!](\mathcal{L})$ well-formed.
- Case INVOKE: The only new writes done by the invocation rule are the output ports of the instance. By construction, they are a set. To prove that the reads are a subset of the writes, observe that the log generated by an instance is, on purpose, ill-formed because it uses placeholder names for the reads as writes. It is the job of the connects to ensure that the log will be well-formed. Note that, by definition, the semantics of port connection only alter the log if the source is in the writing log. Therefore, if we want to show that the reads

are a subset of the writes, every guard in the semantics of connects must be true so that there are no placeholder ports in the log. This follows from the fact that, by construction, every port connection will be well-typed and, by Lemma B.4.1, we can conclude.

□