Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis

Youwei Xiao
shallwe@pku.edu.cn
School of Integrated Circuits, Peking University
Beijing, China

Zizhang Luo
semiwaker@pku.edu.cn
School of Integrated Circuits, Peking University
Beijing, China

Kexing Zhou
zhoukexing@pku.edu.cn
Peking University
Beijing, China

Yun Liang*
ericlyun@pku.edu.cn
School of Integrated Circuits, Peking University & Beijing Advanced Innovation Center for Integrated Circuits
Beijing, China

ABSTRACT

Field-programmable gate arrays (FPGAs) provide opportunities for adopting cutting-edge microarchitectural technologies to accelerate emerging applications. However, it remains challenging to program FPGAs. On one hand, hardware description languages (HDLs), although lauded for their ability to provide circuit representations that closely mimic the inherent hardware structures, have been criticized for their inherent shortcomings, including low-level programming and poor productivity. On the other hand, high-level synthesis (HLS) attempts to raise the abstraction level of hardware design to the software domain. However, it often results in unpredictable solutions due to semantic difference between software and hardware. Furthermore, domain-specific languages (DSLs) tailored for FPGA programming have their own set of limitations, particularly in terms of expressiveness and flexibility.

In this work, we introduce a novel hardware design framework named Cement, which encompasses the embedded HDL (eHDL) CmtHDL and the compiler CmtC, providing a better programming framework for FPGA. CmtHDL introduces event-based procedural specification alongside RTL description, empowering designers to describe hardware productively at a higher level of abstraction while maintaining cycle-deterministic behavior. CmtC provides a comprehensive compilation workflow that includes analyzing the timing behavior of the hardware and conducting synthesis to yield solutions with anticipated performance for FPGAs. Experiments show that Cement provides comparable productivity, but offers 1.41×-3.49× speedup, and saves 23%-82% resources compared to existing HLS or DSL tools. The practical significance of Cement is further validated through a case study of designing real-world FPGA-based accelerators.

*Corresponding author

FPGA '24, March 3–5, 2024, Monterey, CA, USA

ACM ISBN 979-8-4007-0418-5/24/03.
https://doi.org/10.1145/3626202.3637561

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**; **High-level and register-transfer level synthesis**.

KEYWORDS

Embedded HDL, Synthesis, Compiler, Rust, FPGAs

ACM Reference Format:
Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. 2024. Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626202.3637561

1 INTRODUCTION

Field-programmable gate arrays (FPGAs) provide powerful reconfigurability to enhance performance and energy efficiency for diverse applications without the high costs associated with custom silicon. In recent years, FPGA-based accelerators have been emerging across various domains, including artificial intelligence [59], graph processing [14], cryptography [1], and networking [16]. Despite their promise, FPGAs have long suffered from a fundamental challenge: the programming intricacies. The existing programming models exhibit limitations on either design complexity or behavioral accuracy, thereby impeding the FPGA-based accelerators to keep pace with the ever-evolving landscape of emerging applications.

Hardware description languages, such as SystemVerilog program FPGA explicitly at the register-transfer level (RTL), closely align with the hardware's intrinsic nature. While good for manually fine-tuning FPGA performance, HDLs pose challenges, notably their low-level abstraction, which exposes the connections between hardware components but fails to provide insights into the inter-cycle behavior of the hardware. Consequently, achieving expected performance and functionality demands extensive expertise from HDL programmers. This challenge intensifies when implementing algorithms with complex control logic, often represented as finite-state machines (FSMs), necessitating substantial effort for manual design and optimization.

High-level synthesis (HLS) tools like Vitis HLS [55] raise abstraction by synthesizing hardware from annotated subsets of software

languages, such as C/C++. They have gained popularity for improving FPGA programming productivity. Unfortunately, programming FPGAs with software languages presents inherent challenges, given the disparity between software's sequential behavior and hardware's parallel nature. Programmers are compelled to supply directives as *pragmas* to guide synthesis, which introduces limited microarchitectural expressiveness and unpredictability. Emerging domain-specific languages (DSLs) [31, 38, 54] for FPGA programming can only mitigate certain pitfalls of HLS for specific domains.

This paper introduces a novel hardware design framework, Cement, designed to facilitate productive FPGA programming while preserving performance, predictability, and expressiveness. The frontend language, CmtHDL, introduces an *event* layer and an event-based *ctrl* sub-language as the special features in addition to the standard RTL description. The *event* layer captures the occurrence of operations and connections among hardware components, while the *ctrl* sub-language specifies deterministic inter-cycle timing behavior by procedural statements. These innovations raise the level of abstraction for HDL programmers, enabling efficient implementation of target applications without the need for complex control logic design. The CmtC compiler features the control synthesis algorithm that adheres to timing specifications, producing circuits with predictable performance and optimized resource usage tailored for FPGAs. This stands in contrast to HLS tools, which take untimed software programs and yield unpredictable hardware. CmtC also incorporates timing analysis techniques to detect cycle-level timing violations in CmtHDL programs, enhancing correctness and productivity in FPGA programming.

Cement framework uses pure-Rust implementation, built on top of the intermediate representation framework *ir-rs*. We present experiments and a case study to evaluate Cement. Our aim with the Cement framework is to provide FPGA programmers with a superior alternative, allowing them to make FPGA designs with general microarchitectural features productively and deterministically.

The main contributions of this paper are as follows:

- We introduce CmtHDL, a hardware description language that is embedded in Rust and incorporates an *event* layer and the *ctrl* sub-language to facilitate cycle-deterministic FPGA programming.
- We present CmtC compiler, which is built upon the IR framework, *ir-rs*, and incorporates cycle-level timing analysis and control synthesis techniques to produce circuits with correct functionality and anticipated performance.
- We demonstrate that Cement produces low-latency and resource-efficient circuits through experiments. It offers 1.41×-3.49× speedup and saves 23%-82% resources compared to HLS and DSL tools on PolyBench benchmarks. Additionally, a case study highlights Cement's practical significance in real-world accelerator design.
- Cement framework is open source, which is available at https://github.com/pku-liang/Cement.

Paper organization. Section 2 provides background about FPGA programming frameworks and illustrates our motivation with an example. Section 3 describes the language features of CmtHDL. Section 4 introduces the CmtC compiler, including the IR framework, *ir-rs*, as well as analysis and synthesis techniques. Section 5 discusses the evaluation results from experiments and the case study. Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

We discuss existing programming frameworks for FPGAs, including hardware description languages (HDLs), high-level synthesis (HLS) tools, domain-specific languages (DSLs), and intermediate representations (IRs). We summarize the strengths and limitations of the representatives in Table 1, considering factors like microarchitectural expressiveness, preset protocol constraints, cycle determinism, and timing awareness. Subsequently, we elucidate the motivation behind the Cement framework through the design of an FPGA-based accelerator.

2.1 FPGA Programming Frameworks

Hardware Description Languages. Traditional HDLs, exemplified by (System)Verilog and VHDL, operate at the register-transfer level (RTL). Despite offering a close-to-nature representation of hardware, they are notorious for poor productivity and unawareness of cycle-level timing information. Specifically, they expose the connections and operations among hardware constructs without specifying their occurrence at particular cycles, lacking cycle determinism and necessitating meticulous handling by programmers to ensure expected functionality, which is error-prone. In practice, extra logic and signals for compelled synchronization potentially lead to worse frequency and resource consumption.

Embedded HDLs [2, 4, 8, 12, 19, 36, 42], such as Chisel [3], leverage advanced language features to enhance productivity. They provide more user-friendly description syntax for general microarchitecture and facilitate module instantiation with different parameters. Nonetheless, these languages remain rooted in RTL and require manual control logic specification, lacking both cycle determinism and timing awareness.

High-level HDLs [5, 9, 41], including Bluespec SystemVerilog (BSV), employ transactional hardware behavior description. Take BSV as an example. Though it describes general microarchitecture, it requires hardware to implement the ready-enable preset protocol, and causes unpredictable transaction selection at each cycle, lacking both cycle determinism and timing awareness. Such limitations prevent the productive description of correct functionality for FPGA programmers. While BSV introduces the Stmt sub-language for procedural control logic description, it generates FSMs with sub-optimal performance. Some other HDLs [29, 35, 39, 46] also incorporate syntax to describe control logic like looping and pipelining. For example, Filament [39] introduced the timeline type system to describe hardware of limited static pipeline microarchitecture, providing both cycle determinism and timing awareness.

High-level Synthesis. Existing HLS tools [6, 17, 23, 27, 52, 60] like Vitis HLS [55] employ a subset of software languages, such as C++, to specify the untimed behavior of target accelerators. They necessitate directives, like *pragmas*, supported by compilers, to supplement microarchitectural details. These tools rely on black-box heuristics to synthesize hardware, automatically wrapping modules with preset protocol interfaces. HLS provides timing awareness and improves the productivity of hardware design. However, its

Table 1: Comparison between Cement and other representative hardware design frameworks supporting FPGA programming.

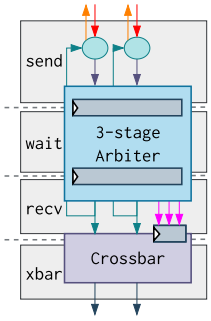
	Design Entry	Product	Control Logic Specification	Microarchitectural Expressiveness	Preset Protocol Constraint	Cycle Determinism	Timing Awareness
Chisel [3]	eHDL(Scala)	RTL	manual	**general**	**none**	no	no
BSV+Stmt [41]	HDL	RTL	procedural	**general**	ready-enable	no	no
Filament [39]	HDL	Calyx	timeline type	limited static pipeline	**none**	**yes**	**yes**
HLS tools [55]	C++/...	RTL	software+directives	limited by directives	preset interfaces	no	**yes**
Dahlia [38]	DSL	HLS/Calyx	software+affine type	only sequential controller	fixed interface	no	**yes**
Spatial [31]	DSL	RTL	software+templates	limited by templates	fixed interface	no	**yes**
Aetherling [15]	DSL	RTL	space-time type	static streaming	fixed interface	**yes**	**yes**
Calyx [40]	IR	RTL	procedural	limited by go-done protocol	go-done	partial	partial
Cement	eHDL(Rust)	RTL	procedural	**general**	**none**	**yes**	**yes**

Note: The "procedural" in the **Control Logic Specification** column means that the control logic can be specified by procedural statements (if, for, etc.) and generated automatically. **Microarchitectural Expressiveness** refers to the microarchitecture that can be described. **Preset Protocol Constraint** refers to the constraints on hardware interfaces, for example, "go-done" indicates that modules must have "go" and "done" signals. **Cycle Determinism** indicates whether the description deterministically dictates the occurrence of the hardware operations at each cycle prior to synthesis. **Timing Awareness** indicates whether programmers can get the timing report of the occurrence of the operations after synthesis or compilation.

software abstraction level, far from hardware's nature, leads to limitations in several folds: (a) The black-box tool can be controlled only via directive settings, preventing cycle-deterministic scheduling decisions and yielding unpredictable circuits under different configurations. (b) Specifying hardware in software with directives constraints microarchitectural design space, hindering advanced optimizations that require expertise in software and hardware.

Domain-specific Languages. DSLs like Dahlia [38] and Spatial [31] have emerged to address HLS limitations. Dahlia employs a time-sensitive affine type system to validate memory access constraints, preventing hardware with unpredictable performance or resource usage. Spatial [31] offers templates to support more microarchitectures for target accelerators. Both Dahlia and Spatial provide timing awareness without guaranteeing cycle determinism. Besides, Aetherling [15] introduces a space-time type system to describe static streaming hardware in a cycle-deterministic and timing-aware manner. These DSLs generate fixed hardware interfaces, and only improve the predictability or extend with more microarchitecture design options for certain applications compared to traditional HLS tools. Similarly, other DSLs [7, 11, 20, 21, 25, 32, 33, 43, 45, 47, 50, 51, 58] are tailored towards accelerator designs serving specific functions (e.g., stencil) or microarchitectures.

Hardware Intermediate Representations. The field of circuit design has witnessed the emergence of new hardware intermediate representations (IRs) [13, 24, 57], along with the CIRCT [10] community. For example, Calyx [40] introduces software-like control flow representation and generates hardware controllers via its compiler, thereby facilitating accelerator generation. However, Calyx only guarantees cycle determinism and timing awareness for programs with explicit latency attributes when static compilation passes are enabled. In other cases, it generates latency-insensitive hardware of microarchitecture limited by the go-done preset protocol. Hector [57] provides a multi-level intermediate representation for hardware synthesis, which guarantees cycle determinism and timing awareness for statically scheduled circuits.



(a) μarch

```
class shuffler ..{
 val io=IO(..)
 val arb=Module(..)
 val state=Reg(..)
 val next=Wire(..)
 state:=next
 when (..) {
  next:= ..;
 } elsewhen (..) {..}
 val S1=stage(state,1);
 when (S1) {
  // resend
  // send to arbiter
 }
 // other stages
}
```

(b) Chisel

```
void shuffler() {
 while (!loop_exit) {
#pragma HLS pipeline
#pragma HLS dependence
  for (..) {
#pragma HLS unroll
   // resend
  }
  // call arbiter
  arbiter(..);
  for (..) {
#pragma HLS unroll
   // xbar
  }
 }
}
```

(c) HLS

Figure 1: Example of a 4-stage pipelined shuffler.

2.2 Motivational Example

To illustrate the limitation of existing programming, we present an example — designing a 4-stage pipelined shuffler with a 3-stage arbiter. The shuffler has been employed in FPGA-based accelerators [14, 22] to address bank conflicts of on-chip memories. Its microarchitecture is outlined in Figure 1a.

General HDLs, such as SystemVerilog and Chisel, lack explicit descriptions of static, non-stallable pipelines that both the shuffler and the arbiter employ, leaving programmers unaware of pipeline stages or timing information. Thus, programmers are compelled to manually insert pipeline buffer registers and implement FSMs, as shown in Figure 1b, which could accidentally lead to unaligned stage schedules or incorrect FSM transactions. Regarding the shuffler design, the depth of the arbiter pipeline necessitates a 2-cycle interval between the shuffler's packet sending and receiving stages. However, when manually designed FSMs violate such timing requirements, HDL compilers generate hardware without reporting errors. This incurs extra time to debug. This issue is prevalent for HDLs without timing awareness.

Du et al. [14] introduce an HLS implementation (see Figure 1c) of the shuffler pipeline in C++, treating pipelines as loops with sequential iterations. However, the software semantics mandate

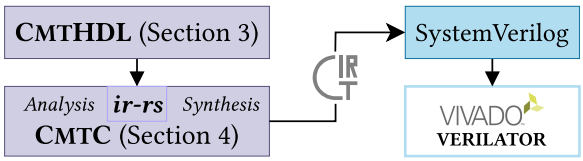


Figure 2: Overview of the CEMENT framework.

that sending packets in the current iteration depends on arbiter decisions from the previous iteration. This results in an initial interval (II) equal to the arbiter pipeline's depth (3 cycles in this example). Achieving an II of 1 requires inserting additional directives (e.g., dependence in Vitis HLS) to redefine dependency constraints. Such optimization requires a deep understanding of the synthesized hardware and the provision of directives to guide black-box synthesis. Notably, the optimization employed by Du et al. [14] worked in Vitis HLS 2020.2 but failed in subsequent releases from 2021 onward. The root cause behind this can be traced to HLS's utilization of untimed specifications instead of describing operations in a cycle-deterministic manner.

Furthermore, the shuffler's complexity surpasses the capabilities of most DSLs. The inability of any of these tools motivates the CEMENT framework, which combines the desirable features elucidated in Table 1. CMTHDL, as shown in Listing 1, explicitly describes the shuffler pipeline using the seq procedural statement. This statement specifies the operations occurring in consecutive cycles deterministically. By setting II=1 for both the shuffler and arbiter pipelines, CMTC compiler verifies the timing constraints on the connection between the shuffler and arbiter and generates hardware modules of the expected performance.

3 CMTHDL

We design a cycle-deterministic HDL, CMTHDL, which is embedded in Rust. It serves as the frontend language for the CEMENT framework, as shown in Figure 2. We provide a brief introduction to CMTHDL's embedding in Rust, including how to customize module interfaces and specify hardware structure (Section 3.1). We emphasize the advantages of tight embedding in Rust. Additionally, we present the innovative features of CMTHDL, including the *event* layer and the *ctrl* sub-language (Section 3.2). These features raise the abstraction level of hardware description with timing information. CMTHDL enables FPGA programming in a more productive and deterministic manner, without sacrificing direct control over microarchitectural details. Furthermore, we explore support for external modules, such as DSP intellectual property (IP), with specified timing information (Section 3.3).

3.1 HDL Embedded in Rust

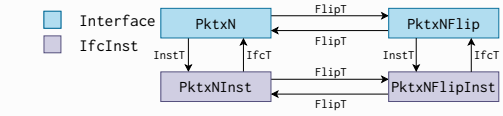
The major characteristic of CMTHDL's embedding in Rust is the tight integration with the Rust type system. Specifically, CMTHDL allows programmers to define customizable hardware constructs, including data types and module interfaces, as Rust types using *traits* [30, 49]. The four traits in Table 2 dictate all the required functionality of data types, data bundles, interfaces, and instantiated interfaces, respectively, where a data bundle represents a collection of **undirectional** data types, an interface represents a collection of **directional** data types, and an instantiated interface is the product of instantiating the corresponding interface within a target module, comprising ports of directional data types.

The traits enable the primary RTL features including module instantiation and port connection by the trait methods, while CMTHDL further provides an extensive operation mechanism to support various operations on ports. Besides, CMTHDL leverages the powerful *macro* system in Rust to provide concise syntax for hardware description. Overall, CMTHDL provides comprehensive support for RTL description within the Rust programming language.

The main benefits of this approach are the combination of enhanced type checking on hardware constructs and greater parameterization flexibility. Specifically, defining hardware constructs as Rust types promptly provides diagnostic feedback through linting or, importantly, at compile-time, prior to the elaboration and hardware generation phases, if any type violation, such as the width mismatch of the data types and the direction mismatch of the ports, is detected. Besides, traits in CMTHDL support both the compile-/elaboration-time parameterization for hardware constructs. Compile-time parameterization enables the type checking on the parameters, while elaboration-time parameterization provides greater flexibility for the parameters to remain uncertain until the elaboration phase. CMTHDL adeptly accommodates both modes of parameterization for programmers to choose, seamlessly combining their specific benefits.

Beyond the advantages above, embedding in Rust also leads to additional benefits, including access to the expansive Rust ecosystem, fast elaboration time, and minimal memory usage. We further explain CMTHDL's embedding in Rust by the example of the shuffler module (Figure 1a) described in Listing 1, which generally contains two steps: (a) define data and interfaces, and (b) instantiate modules, specify operations, and connect ports.

Data and Interface Definition. Lines 1-14 in Listing 1 effectively define the necessary data types and interfaces for the shuffler module, where B<N> implements the DataType trait (❶), representing a data type signifying a bit vector with a compile-time constant width of N, and Pkt<N,T> implements the Bundle trait (❷), representing a bundle of the data T, the destination addresses B<{clog2(N)}>, and the valid bit B<1>. Pkt<N,T> is defined using the bundle macro at line 1, which automatically implements the Bundle trait for the specified struct type. Besides, the PktxN interface is defined using the interface macro at line 7, which generates three related types—PktxNFlip, PktxNInst, and PktxNFlipInst—and implements appropriate traits for these types, as illustrated below:



Notably, the Interface trait (❸) features two associated types: FlipT and InstT. The former signifies the interface with the opposite direction, while the latter denotes the type of the corresponding instantiated interface. Similarly, the IfcInst trait (❹) has two associated types: FlipT and IfcT.

Module Instantiation, Operation, and Connection. Lines 15-39 in Listing 1 define the shuffler module with the PktxN<N,T> interface instantiated into local variable io of type PktxNInst<N,T>.

Table 2: Traits and example implementation in CmtHDL

Trait	Example Implementation

`// data types`
`trait DataType {`
`  fn width() -> usize;`
`  fn ir_type() -> IrData;`
`}`
	`impl<const N: usize> DataType`
	`  for B<N> {`
	`  fn width() -> usize { N }`
	`  // ir_type() omitted`
	`  }` ❶

`// data bundles`
`trait Bundle {`
`  fn total_width() -> usize;`
`  fn ir_types() -> Vec<IrData>;`
`}`
	`impl<const N: usize, T:DataType>`
	`  Bundle for Pkt<N,T> {`
	`  fn total_width() -> usize {`
	`  T::width() + clog2(N) + 1`
	`  } // ir_types() omitted`
	`  }` ❷

`// interfaces`
`pub trait Interface: {`
`  type FlipT: Interface<FlipT=Self>;`
`  type InstT: IfcInst<`
`  IfcT = Self,`
`  FlipT = Self::FlipT::InstT`
`  >;`
`  fn to_inst(self) -> Self::InstT;`
`  // other methods omitted`
`}`
	`impl <const N:usize, T: DataType>`
	`  Interface for PktxN<N,T> {`
	`  type FlipT = PktxNFlip<N,T>;`
	`  type InstT = PktxNInst<N,T>;`
	`  fn to_inst(self)->Self::InstT {`
	`  PktxNInst {`
	`  clk: self.clk.to_inst(),`
	`  // omitted`
	`  }`
	`  } // other methods omitted`
	`  }` ❸

`// instantiated interfaces`
`pub trait IfcInst:`
`Connect<Self::FlipT> {`
`  type FlipT: IfcInst<FlipT=Self>;`
`  type IfcT: Interface<InstT=Self>;`
`  fn ifc(&self) -> Self::IfcT;`
`  // other methods omitted`
`}`
	`impl <const N:usize, T: DataType>`
	`  IfcInst for PktxNInst<N,T> {`
	`  type FlipT = PktxNFlipInst<N,T>;`
	`  type IfcT = PktxN<N,T>;`
	`  fn ifc(&self) -> Self::IfcT {`
	`  // omitted`
	`  } // other methods omitted`
	`  }` ❹

```
1  #[bundle(Default)]
2  pub struct Pkt<const N: usize, T: DataType> {
3    data: T,
4    dest: B<{ clog2(N) }>,
5    valid: B<1>,
6  }
7  #[interface(Default)]
8  pub struct PktxN<const N: usize, T: DataType> {
9    clk: Clk,              // in
10   go: B<1>,              // in
11   i: [Pkt<N,T>; N],     // in
12   ready: <[B<1>; N] as Interface>::FlipT, // out
13   o: <[Pkt<N,T>; N] as Interface>::FlipT,// out
14 }
15 module! {
16   <const N:usize, T: DataType> PktxN<N,T>  =>
17   shuffler(io, /* args */) {
18     let arbiter = instance!(arbiter(Arbiter::<N,T>::new()));
19     let resend = wire!(arbiter.resend.ifc());
20     let sel_reg = reg!(arbiter.sel.ifc(), io.clk);
21     let receive = event! {
22       resend.i %= arbiter.resend;
23       sel_reg.wr %= arbiter.sel;
24     };
25     let send = event! {
26       for (i, ready, resend, arb_pkt) in
27         multizip((io.i, io.ready, resend.o, arbiter.pkt)) {
28           arb_pkt %= receive.mux(resend.valid.clone().mux(resend,
             ↪ i.clone()), i);
29         };
30         // omitted
31     };
32     // events `wait` and `xbar` are omitted
33     let pipeline = stmt! {
34       seq { send; wait; receive; xbar; }
35     }
36     let go = event!(io.go);
37     synth!(pipeline, Pipeline::new(io.clk, go, 1));
38   }
39 }
```

Listing 1: Shuffler (Figure 1a) in CmtHDL with more details

The `instance!` macro at line 18 instantiates an `arbiter` submodule from the interface `Arbiter::<N,T>::new()`. The inner `mux` operation at line 28 takes three operands: `resend.valid`, a single port of data type `B<1>`, as well as `resend` and `i`, both of which are a collection of ports defined by the `Pkt<N,T>` bundle. Lines 22-23 employ the overloaded `%=` operator to specify connections.

3.2 Event-based Extension

We introduce the event-based extension to CmtHDL. The extension encompasses the *event* layer that defines the events signifying the occurrence of guarded operations and connections across cycles. It also includes the *ctrl* sub-language, which specifies the timing behavior of events through procedural statements. This extension is designed to enhance CmtHDL by providing cycle determinism and timing awareness features. It empowers FPGA programmers to work at a higher level of abstraction while maintaining deterministic specifications.

Event Layer. The event layer specifies events guarding hardware behavior. An event is a set of guarded hardware behaviors that consistently occur simultaneously. This includes operations and connections, along with timing information indicating the cycle at which these behaviors occur. In CmtHDL, we extend the `Event` type, which is constructed using the `event!` macro.

Events can provide timing information in two formats: (a) a boolean signal that indicates whether the behavior occurs at a specific cycle, and (b) a sequence of cycles during which the guarded behavior occurs. *Considering the boolean signal format*, CmtHDL provides the syntax for the conversion between events and boolean signals in the RTL description. *Considering the sequence of cycles format*, we formally denote it as $sc[e]$ for the event e. Events can be classified as static or dynamic, depending on whether the sequence of cycles can be determined during elaboration. Dynamic events, also known as data-dependent events, have their cycle sequence related to data from input ports.

For instance, lines 21-24 and 25-31 in Listing 1 define the `receive` event and the `send` event, respectively. The former guards two connections. At line 36, the `go` event is constructed from the boolean port `io.go`. Meanwhile, at line 28, the `receive` event, converted to a boolean signal, becomes an operand for the outer `mux` operation.

The event layer equips CmtHDL with timing awareness, enabling programmers to access timing information for guarded hardware behavior. CmtHDL further dictates the cycle determinism feature for events, indicating that the $sc[e]$ must be deterministic for every event e given the data fed through the input ports at the specific cycles. For example, when the `io.go` signal in Listing 1 is asserted at cycle {0, 1, 2}, the `send` event has $sc[\texttt{send}]$={0, 1, 2} deterministically. It requires the timing information of events to be specified in a strict manner, as the *ctrl* sub-language observes.

Ctrl Sub-Language. We introduce the *ctrl* sub-language to specify timing information for events while maintaining determinism. This sub-language employs procedural statements to define the

Table 3: Statements in the *ctrl* sub-language

Variants	`Step(StepStmt)`	`Seq(SeqStmt)`	`Par(ParStmt)`	`If(IfStmt)`	`For(ForStmt)`	`While(WhileStmt)`
Type Definition	`struct StepStmt { events: Vec<Event>, entry: Vec<Event>, exit: Vec<Event>, }`	`struct SeqStmt { stmts: Vec<Stmt>, }`	`struct ParStmt { stmts: Vec<Stmt>, }`	`struct IfStmt { cond: Event, t_stmt: Box<Stmt>, e_stmt: Box<Stmt>, }`	`struct ForStmt { indvar: Reg, range: Range, do_stmt: Box<Stmt>, }`	`struct WhileStmt { cond: Event, do_stmt: Box<Stmt>, }`
Macro Syntax	`[x] e0, e1 [y]`	`seq { s0; s1 }`	`par { s0; s1 }`	`if { [cond] t_stmt else e_stmt }`	`for { [indvar in range] do_stmt }`	`while { [cond] do_stmt }`
Timing Rule	Wait until `x` happens, trigger `e0` and `e1` in **one** cycle, then wait until `y` happens.	Trigger `s0` and `s1` sequentially **without interval**.	Trigger `s0` and `s1` **immediately**, wait until all done	Trigger `t_stmt` or `e_stmt` **immediately** if cond happens or not.	Repeat `do_stmt` **without interval** according to `range`.	Repeat `do_stmt` **without interval** until cond fails.
Latency	1 + #entry-cycles + #exit-cycles	L[`s0`]+L[`s1`]	max{L[`s0`], L[`s1`]}	L[`t_stmt`] or L[`e_stmt`]	L[`do_stmt`] ×trip-count	L[`do_stmt`] ×trip-count
Cycle Inference	sc[`e0`] = sc[`e1`] = sc[s]+#entry-cycles	sc[`s0`] = sc[s], sc[`s1`] = sc[s]+L[`s0`]	sc[`s0`] = sc[s] sc[`s1`] = sc[s]	sc[`t_stmt`] = sc[s], if cond happens	sc[`do_stmt`] = sc[s]+k*L[`do_stmt`]	sc[`do_stmt`] = sc[s]+k*L[`do_stmt`]

timing of events. Each statement in the *ctrl* sub-language consists of sub-statements or events whose timing information adheres to deterministic rules. The *ctrl* sub-language provides an enum type called `Stmt`, which includes six variants corresponding to six supported statements (see Table 3). Each statement is implemented as a struct type (as indicated in the "Type Definition" row of the table). Furthermore, CmtHDL offers a macro called `stmt!` that constructs a statement from the provided statements or events, using the syntax informally presented in the "Macro Syntax" row of the table. For example, in Listing 1, lines 33-35 define a `seq` statement using four `step` statements, each constructed from a single event.

The "Timing Rule" row in the table illustrates how statements specify timing information for their contained statements or events. This specification maintains cycle determinism, with the unit statement, `step`, explicitly stating that its contained events are triggered at the same cycle. All other statements ensure there are no extra, unexpected cycles. As a result, the statements have deterministic latencies, denoted as $L[s]$ for the statement s, as shown in the "Latency" row. Note that the latencies of the statements are allowed to be *data-dependent*, such as the entry-/exit-cycles for the `step` statements and the branch choice for the `if` statement. However, they are determined by the timing rules given the specific inputs.

Additionally, we extend the definition of the sequence of cycles to include statements, denoted as $sc[s]$. This sequence represents the cycles at which the statement begins execution. With cycle determinism ensured by the timing rules, we can infer the timing information for the events and statements within a given statement according to the "Cycle Inference" row. This top-down inference allows for further analysis, as detailed in Section 4.2.

Finally, we provide the `synth!` macro, which takes a statement from the *ctrl* sub-language extension and a configuration value to synthesize control logic that meets the statement's timing specification. The synthesis process is elaborated upon in Section 4.3. This feature enables FPGA programmers to describe hardware behavior at a higher level of abstraction while maintaining cycle determinism, highlighting the main benefit of the *ctrl* sub-language extension for FPGA programming. For instance, line 37 in Listing 1 synthesizes the `pipeline` statement using a configuration that specifies the clock signal, triggering event, and initial interval (`II=1`).

3.3 Timed External Modules

CmtHDL provides a feature that allows for the description of external modules with cycle-level timing information specified. This capability is particularly valuable for FPGA programmers seeking to leverage on-board resources, including Block RAMs (BRAMs) and DSP slices (DSPs), to improve their target designs.

```
1 extern_module! { <T: DataType> DspBinOp<T> /* x, y, rst */ =>
2   multiply_dsp(io, lat: u32)[tcl = format!("read_ip ..")] {
3     let (e_x, e_y, e_rst) = guard!(io.x, io.y, io.rst);
4     let s_delay = delay(lat-1);
5     specify! { stmt! { seq { e_x,e_y; s_delay; e_rst } } };
6   }
7 }
```

The CmtHDL description above describes a pipelined multiplier that is implemented on DSP blocks with a parameterized latency `lat`. The `extern_module!` macro defines an external module from the provided interface `DspBinOp<T>`. Line 2 describes the Tcl command to read and configure the intellectual property (IP) used by the multiplier. Lines 3-5 define the timing information of the module, where the `guard!` macro creates events to guard ports, the `delay` function constructs a `seq` statement of empty `steps` from the given number of cycles, and the `specify!` macro specifies the timing information of the module using the `seq` statement, which is similar to the `synth!` macro but does not synthesize control logic.

This feature enables programmers to integrate black-box IPs in a cycle-deterministic manner, facilitating the detection of cycle-level timing violations, such as fetching results at inappropriate cycles, through timing analysis (detailed in Section 4.2). Additionally, the `extern_module!` macro enhances interoperability with commercial toolchains and legacy modules written in traditional HDLs, such as (System)Verilog. This is achieved by replacing the `tcl` keyword

(line 2) with the `path` keyword, which takes the path to the external (System)Verilog file as an argument.

4 CMTC COMPILER

The CmtC compiler produces hardware solutions from CmtHDL programs, as shown in Figure 2. We introduce the Rust-based intermediate representation (IR) framework, *ir-rs*, upon which we construct CmtC. Then, we describe the principal features of the compiler, including the timing analysis (Section 4.2), and the control synthesis (Section 4.3) that synthesizes the control logic from the *ctrl* sub-language. These analysis and synthesis techniques harness the event-based extension features of CmtHDL, significantly improving the productivity of hardware design by alleviating the burden of manual timing validation and FSM implementation.

4.1 *ir-rs*: IR Framework in Rust

ir-rs is an IR framework implemented in pure Rust. Inspired by projects like MLIR [34] and xDSL [53], we design *ir-rs* around operations that represent IRs in the static single-assignment (SSA) form [44]. Each operation type is defined as a Rust struct that inherently implements the trait `Op`. This trait outlines the common behavior expected from SSA IRs, which includes methods such as `get_defs` for retrieving values defined by the operation, `get_uses` for accessing values used by the operation, and more. We facilitate the creation of new operation types with the `operation!` macro. *ir-rs* also offers a mechanism for defining checking and printing rules for custom operation types. Additionally, it allows for programming transformation passes that operate on these operations. It provides the `machine!` macro to create an IR machine responsible for storing and manipulating the selected operations.

To support the CmtC compiler, we implement the operations from CIRCT core dialects [10] in *ir-rs* through the `operation!` macro. Subsequently, we define operations to accommodate the event-based extension of CmtHDL. We also implement transformation passes for timing analysis techniques (Section 4.2) and control synthesis (Section 4.3). Eventually, the CmtC compiler encompasses an IR machine, referred to as CmtIR, which incorporates the defined operations and passes through the `machine!` macro. The embedded CmtHDL programs will create IR operations in CmtIR with the provided APIs during elaboration. Additionally, CmtIR supports operation deduplication by hashing the current operation and verifying whether an identical operation exists. This feature reduces peak memory consumption during elaboration.

After applying analysis and synthesis passes within CmtIR, CmtC yields a final circuit comprised solely of operations from the CIRCT core dialects. The backend then applies CIRCT passes, such as `ExportVerilog`, to produce SystemVerilog code that can be further synthesized using tools like Vivado [56] or validated through RTL simulation using Verilator [48], Khronos [61], etc. Additionally, CmtC provides programmers with APIs to create testbenches for simulation purposes, which is omitted in this paper.

4.2 Timing Analysis

As outlined in Section 3.2, events and statements can be categorized as either static or dynamic, depending on whether their cycle sequences can be determined during elaboration. Consequently, we

```
let s_for = stmt! {
  for {
    [i in 0..4]
    e_pipe
  }
};
synth!(s_for, io.go);
```

(a) `static_m` module

```
let s_pipe = stmt! {
  seq {
    send_mul; delay1;
    recv_mul
  }
};
synth!(s_pipe, e_pipe);
```

```
let s_while = stmt! {
  while { [cond]
    [io_a_valid];
    recv_x, store_x
  }
};
synth!(s_while, io.go);
```

(b) `dyn_m` module

Figure 3: Examples of *ctrl* sub-language for timing analysis

introduce two distinct timing analysis techniques: *static analysis*, which focuses on static events and statements to identify timing violations before simulation, and *dynamic monitoring*, which observes event execution during simulation to detect violations for specific input sets. Both techniques offer unique advantages and can complement each other to enhance productivity.

Static Analysis. Static statements refer to those whose contained statements and events have a fixed sequence of cycles that can be determined during elaboration. Static statements supported by the *ctrl* sub-language, as presented in Table 3, include `seq` statements without entry and exit events, `seq` or `par` statements containing solely static sub-statements, and `for` or `while` statements with static `do_stmt` and constant trip-counts.

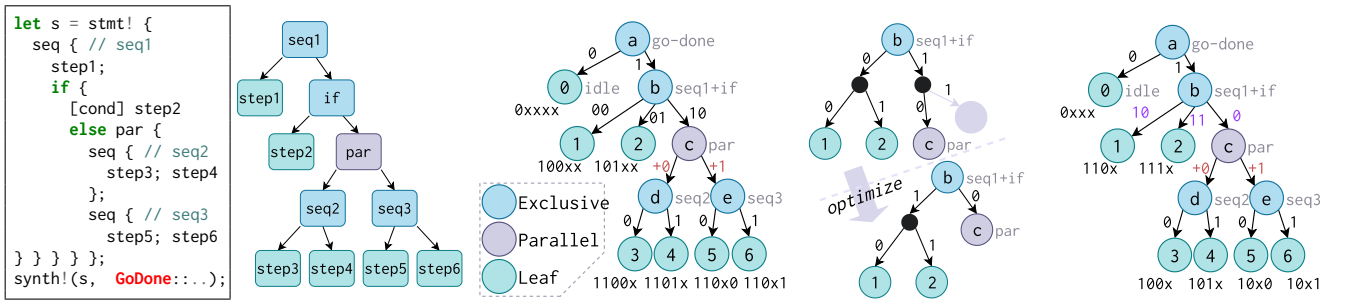
Static analysis begins by identifying all static statements and events within target modules, filtering out root statements or events that are not contained or invoked by other static constructs. For example, all statements and events in Figure 3a are static, while none are in Figure 3b. The only root event in the `static_m` module is `io.go`, which invokes the `s_for` statement. Subsequently, the analysis employs a post-order traversal to determine the latency of each statement by aggregating the latencies of its sub-statements, following the formulas in the "Latency" row of Table 3.

The analysis initializes the cycle sequence for every root statement or event as {1}, signifying execution in the first cycle. It then proceeds with a pre-order traversal to infer the cycle sequence for the remaining statements and events, guided by the formulas presented in the "Cycle Inference" row of Table 3. Subsequently, the analysis checks for timing violations. It iterates through all ports or wires within the target modules, collecting the cycle sequences for data reception and transmission for each one. By comparing them, the analysis identifies timing violations when a mismatch occurs, indicating either transmission of invalid data or data loss.

For instance, within the `static_m` (Figure 3a), `sc[io.go]` is set as {1} for the root event `io.go`. Then, both the `sc[s_pipe]` and `sc[send_mul]` are inferred as {1,2,3,4}, while `sc[recv_mul]` is inferred as {3,4,5,6}. Assuming that the `recv_mul` event guards the connection where the `rst` port of the submodule instantiated from `multiply_dsp` with `lat=3` transmits data to the current module, the cycle sequence at which the `rst` port sends data is {3,4,5,6}, which equals `sc[recv_mul]`. However, the cycle sequence at which the `rst` port receives data, inferred within the `multiply_dsp` module, is {4,5,6,7}. This mismatch indicates a timing violation.

The primary advantage of static analysis is its ability to detect timing violations during elaboration, without requiring testbenches. While this technique offers quicker violation feedback, its scope is more limited, only encompassing static statements and events.

Dynamic Monitor. The dynamic monitor technique harnesses the boolean signal format of events to oversee their execution during



(a) *ctrl* sub-language and AST (b) state tree representation (c) optimization (d) optimized state tree

Figure 4: Construction and optimization of state tree representation

simulation. The process begins by collecting all events within the target modules. For each event, the monitor introduces a boolean signal denoting whether the event executes in the current cycle. Additionally, the monitor generates two boolean signals for each port or wire within the target modules: one indicating whether the port or wire receives data in the current cycle and the other signifying whether it transmits data during the current cycle. The technique subsequently automatically devises combinational logic for these generated boolean signals. Finally, the monitor incorporates `assert` statements into the produced hardware description programs in SystemVerilog and identifies timing violations by asserting the mismatch of the two boolean signals for ports or wires at each cycle during simulation.

The advantage of the dynamic monitor technique is its capability to detect timing violations in circuits with data-dependent behavior, exemplified by the `dyn_m` module in Figure 3b. However, it possesses the limitation of solely identifying violations based on provided testbenches, and it introduces auxiliary signals that impose computational burden during the simulation phase.

4.3 Control synthesis

We introduce the control synthesis algorithm to synthesize the control logic from the *ctrl* sub-language of CmtHDL. It strictly implements the specified timing behavior, and thus, keeps the cycle determinism feature of CmtHDL to produce hardware of predictable performance. We describe the *state tree representation* that supports the synthesis process, and its construction from the *ctrl* sub-language. We then formulate an optimization problem to minimize the resource usage of the synthesized FSM and present the optimization algorithm that modifies the state tree representation to achieve the optimization target.

State tree representation. We propose the state tree representation to describe the relationship between events and their encoding in the FSM. There are four kinds of nodes in a state tree: leaf, mutually exclusive, parallel, and pipeline. A leaf node is an event under control. A mutually exclusive node indicates that only one of its children is executed at a time. A parallel node indicates that all of its children may be executed at the same time. A pipeline node is a variation of the parallel node to deal with pipelining. As shown in Figure 4a and Figure 4b, a state tree is constructed from an AST of the *ctrl* sub-language. First, `step` statements are converted to leaf nodes, `par` statements are converted to parallel nodes, and all the other statements are converted to mutually exclusive nodes. Second, connected nodes of the same type are merged. For example, the `seq1` and `if` node are merged into node `b`. Finally, *protocol* nodes are added according to the synthesis configuration, such as the mutually exclusive root node `a` and the leaf node `0` that implement the specified go-done protocol.

The state tree can decide the encoding of each leaf node. As shown in Figure 4b, a mutually exclusive node assigns one distinct binary encoding to each child, and a parallel node assigns one offset to each child so that their encoding spaces do not overlap. The route from the leaf node to the root decides its encoding. Take leaf node `5` as an example, node `e` assigns the postfix `0`, node `c` offsets it by `1` to get `x0`, and from `c` to `a` it further gets `110` in the front to make the final encoding `110x0`. The pipeline node is treated specially, it assigns a one-hot encoding to its children. As a result, a shift register can enable different stages in the pipeline.

Optimization. The optimization space is the encoding of the mutually exclusive nodes. As shown in Figure 4c, the encoding assignment of node `b` can be represented in the form of a binary tree. The child states are distinguishable as long as they occupy different leaf nodes in the binary tree, and do not need to have equal lengths. By modifying the shape of the binary tree, the total width of the state is reduced from 5 to 4 as shown in Figure 4d.

The optimization target is to minimize LUT utilization under constraints of limited FF utilization and frequency. The width of the state register determines FF utilization, and the complexity of combinational logic, including transition and output, determines LUT utilization. Simply using one-hot encoding greatly simplifies transition with the cost of an extremely wide state register, so a constraint is added that the number of FFs is bounded by a constant times the number of statements. For frequency constraint, we set the upper bound for the number of cascade LUTs. Both the numbers of the total LUTs and the cascade LUTs can be calculated from the number of the bits in the state that are required for event triggering and state transition.

We introduce a heuristic for the optimization. First, all nodes are initialized to use the Huffman-tree-like scheme with the width of their encoding in the sub-tree as sorting keys, where the child node that needs more bits is closer to the root. Then the mutually exclusive nodes are sorted by the height of the Huffman tree. From tall to short, the encoding schemes of some nodes are changed to one-hot to reduce LUT utilization until reaching the FF limitation.

5 EVALUATION

Our evaluation consists of three parts. First, we evaluate Cement's performance by testing it with kernels from the PolyBench benchmark suite. CmtHDL provides cycle-deterministic descriptions for inter-cycle hardware behavior, which requires the absence of extra cycles in the produced circuits to guarantee the expected performance, while the control synthesis algorithm of CmtC optimizes the resource efficiency of the circuits. The kernels from the Polybench have diverse behaviors in terms of computation and control, such as branches and loops, which are qualified to compare the performance and resource efficiency of the produced circuits and verify the effectiveness of our methodology. We compare Cement with the commercial HLS tool Vitis HLS [55] and the FPGA programming DSL Dahlia [38] compiled by the Calyx [40] framework. Then, we conduct a case study on systolic array accelerators to demonstrate Cement's benefits for real-world accelerator design.

5.1 Experiments on PolyBench

For our experiments with the PolyBench benchmark suite, we compare cycle count, resource utilization, and lines of code (LoC) for description[1]. We collect cycle counts by simulating the produced SystemVerilog code with Verilator[48] and estimate resources by running synthesis with Vivado 2021.2, targeting Virtex UltraScale+ XCVU9P FPGA. For Dahlia-Calyx flow, we follow the instructions provided in the calyx-evaluation repository[2]. We collect cycle counts by simulating designs in Verilator and estimate resources using Vivado 2021.2 under the same configuration as Cement. For Vitis HLS 2021.2, we collect the metrics including cycle count and resource utilization from the co-simulation and implementation reports. We set the target clock period as 7ns for Vitis HLS designs while configuring the same target clock period for the synthesis of Cement and Dahlia-Calyx for fair comparison.

Against Vitis HLS. Figure 5a shows that Cement designs use fewer cycles for all the kernels. Considering frequency, Cement designs achieve 1.41× geomean speedup compared to Vitis HLS with loop pipelining and flattening disabled. Cement outperforms Vitis HLS because Vitis HLS adopts conservative scheduling with an approximate timing model, preventing designers from making scheduling decisions and leading to poor performance. On the contrary, Cement allows users to describe hardware behavior in a cycle-deterministic manner. Take the "atax" kernel as an example, its dot-product loop is scheduled to have an iteration latency of 4 cycles by Vitis HLS for the 7ns target clock period. This can not be further optimized by directives from users. However, it's convenient to describe the control logic by a `seq` statement containing 3 steps in the *ctrl* sub-language of CmtHDL, which achieves better latency while meeting the timing target.

Moreover, Cement saves 23% LUTs and 68% FFs on average due to the optimization effects of the control synthesis technique (introduced in Section 4.3), and provides comparable productivity (0.97× geomean LoC) against Vitis HLS designs, as shown in Figure 5b, Figure 5c, and Figure 5d. However, the Cement designs consume more LUTs for 6 kernels, namely "doitgen", "gesummv", "gramschmidt", "lu", "symm", and "trmm". The reason is that CmtC generates hardware solutions with the cycle-deterministic behavior enforced by the CmtHDL specification, requiring extra overheads for additional states and transitions in control logic. For kernels with nested loops of multiple levels, such as the "doitgen" kernel with 4-level nested loops, the overheads get accumulated and lead to more resource consumption, especially for LUTs.

We further implement pipelined designs for 9 kernels in Cement in the same manner as Figure 3a does. We compare them against the Vitis HLS designs with pipelining enabled. Figure 6a shows that Cement designs use fewer cycles on all the kernels. They achieve 1.52× geomean speedup considering the achieved frequencies. As for resources, the Cement designs use fewer LUTs and FFs on most of the kernels except the kernels "doitgen" and "gesummv". On average, they save 47% LUT and 78% FF, while with only 0.95× geomean LoC for description.

Against Dahlia-Calyx flow. We enable Calyx's static timing optimization for the experiments. Figure 5a presents the performance results. Considering the achieved frequency, Cement achieves 3.49× geomean speedup against Dahlia-Calyx flow. The performance gain stems from the control synthesis technique that guarantees the expected timing behavior specified by the *ctrl* sub-language of CmtHDL, which removes all the unnecessary cycles. Besides, Cement designs save 54% LUTs and 82% FFs compared to Dahlia-Calyx designs as shown in Figure 5b and Figure 5c. In addition, Figure 5d shows that descriptions in Cement use 25% fewer lines of code on average.

Summary. The results on PolyBench demonstrate that CmtHDL provides similar-to-HLS productivity, and the CmtC compiler generates low-latency and resource-efficient circuits for most cases.

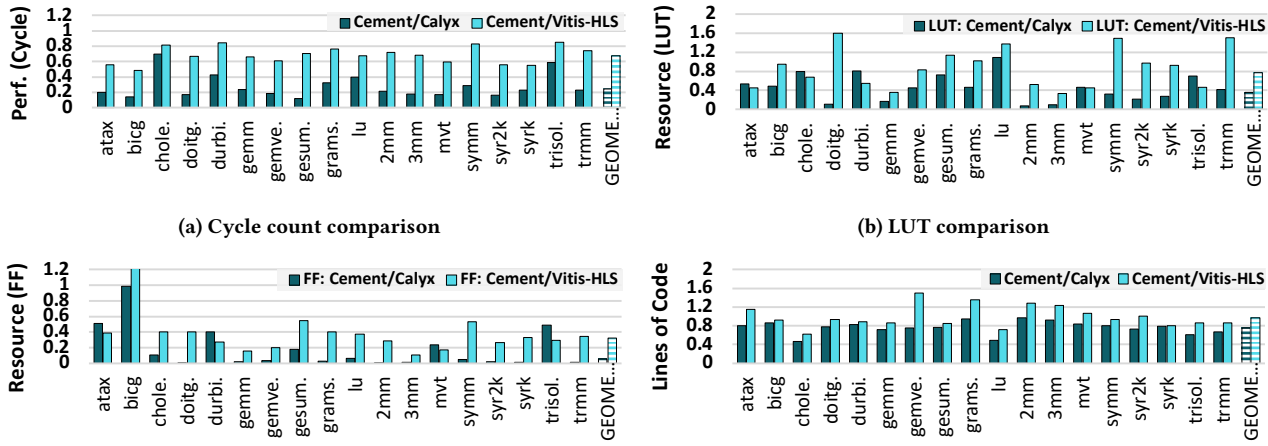
5.2 Case Study: Systolic Array

In this case study, we evaluate Cement for systolic array design to demonstrate the practical significance of the cycle determinism feature. Systolic array is the core component of various dataflow accelerators[18, 25, 26, 28, 37]. Each tensor needs a schedule-specific controller for its data movement into/from the array. Figure 7 shows a schedule for matrix multiplication $A \times B = C$, where task 1 preloads tensor B into the array and keeps it stationary. Tasks 2-3 move tensor A horizontally into the array in a systolic manner, and each row is one cycle delayed after the previous one. Tasks 4-5 vertically move tensor C out of the array.

CmtHDL's cycle-deterministic description and static timing analysis help to prevent cycle misalignment by detecting it as timing violations as introduced in Section 4.2. An appropriate number of padding cycles need to be added before each task. Figure 7 shows an example of such cycle alignment. (a) To align task 1 and task 2, the first data of tensor A reaches the systolic array at the exact cycle when tensor B finishes moving into the array. (b) Task 3 is one cycle delayed after 2 to skew the tensor. Task 4 and 5 are similar. (c) To align task 2 and task 4, when the result reaches the edge of the array, a partial sum of tensor C has just been loaded for accumulation.

We compare Cement against AutoSA[51] and EMS[26]. AutoSA is a systolic array compiler that generates Vitis HLS code. We estimate its development effort according to the total size (35k) of the definition code for the sub-modules such as PE. Though HLS

[1]We only take significant code into consideration for the LoC metric, excluding comments, empty lines, and non-synthesizable code.
[2]https://github.com/cucapra/calyx-evaluation



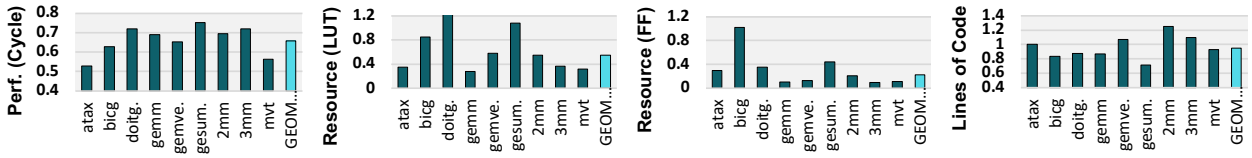
(a) Cycle count comparison

(b) LUT comparison

(c) FF comparison

(d) Lines-of-Code (LoC) comparison

Figure 5: Sequential design comparison for Cement, Dahlia-Calyx, and Vitis HLS on PolyBench benchmarks. The y-axis represents the ratio of Cement and the other two methods. The smaller the value, the better Cement performs.



(a) Cycle count comparison

(b) LUT comparison

(c) FF comparison

(d) Lines-of-Code comparison

Figure 6: Pipelined design comparison for Cement and Vitis HLS on PolyBench benchmarks. The y-axis represents the ratio of Cement and Vitis HLS. A smaller value means that Cement has better results.

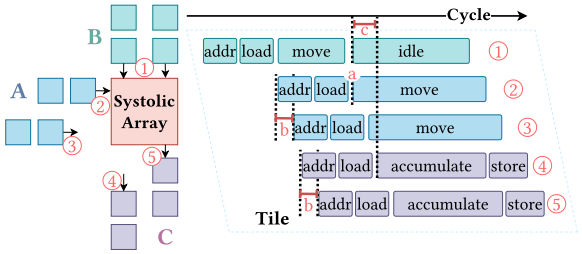


Figure 7: Illustration for systolic array

Table 4: Comparison of systolic array hardware.

Design	Device	LUT	DSP	Frequency	Throughput
AutoSA[51]	U250	968k	9462	272MHz	949.98GFLOPS
EMS-WS[26]	XCVU9P	898k	4494	301MHz	731.17GFLOPS
Cement-small	XCVU9P	437k	3840	322MHz	823.97GFLOPS
Cement-large	XCVU9P	543k	4800	333MHz	1065.60GFLOPS

prevents cycle alignment bugs, the difficulty of describing spatial hardware structure in HLS leads to more development effort. EMS implements the systolic array using Chisel[3]. CmtHDL uses less code (16.6KB) and development effort (2 person-months) than EMS (36KB, 6 person-months).

In Table 4, we parameterize two Cement designs with different systolic array sizes, namely Cement-small (32×40) and Cement-large (40×40). Cement-small saves 51% LUTs and 15% DSPs compared to EMS-WS, and improves 7% for frequency and 13% for throughput, while Cement-large saves 44% LUTs and 49% DSPs compared to AutoSA, and improves 22% for frequency and 12% for throughput. In summary, Cement helps us to achieve better accelerator designs with even less development effort.

6 CONCLUSION

We introduce the Cement framework as a better choice for FPGA programming. It comprises the Rust-based eHDL CmtHDL, which features the event-based extension for cycle-deterministic hardware description by procedural statements, and the compiler CmtC, which supports the timing analysis techniques and the control synthesis algorithm. Cement is built around the Rust-native IR framework *ir-rs*. We conduct experiments on PolyBench benchmarks to demonstrate that Cement produces circuits of the expected performance and efficient resource usage. The case study on systolic array accelerators demonstrates Cement's practical significance.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of China (Grant No. T2325001, Grant No. T2293700 and T2293701).

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 882–895. https://doi.org/10.1109/HPCA56546.2023.10070953

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. ClaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 714–721. https://doi.org/10.1109/DSD.2010.21

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584

[4] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. 2020. Creating an Agile Hardware Design Flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218553

[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[7] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240850

[8] ChipFlow. [n. d.]. *Amaranth HDL*. https://github.com/amaranth-lang/amaranth

[9] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (aug 2017), 30 pages. https://doi.org/10.1145/3110268

[10] CIRCT Community. 2022. *CIRCT: Circuit IR Compilers and Tools*. Retrieved October 31, 2022 from https://github.com/llvm/circt

[11] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240838

[12] Jan Decaluwe. [n. d.]. *MyHDL*. https://www.myhdl.org/

[13] John Demme. 2021. Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design. arXiv:2111.06584 [cs.AR]

[14] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3490422.3502368

[15] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. https://doi.org/10.1145/3385412.3385983

[16] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 345–362. https://www.usenix.org/conference/atc19/presentation/eran

[17] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[18] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[19] Sungsoo Han, Minseong Jang, and Jeehoon Kang. 2023. ShakeFlow: Functional Hardware Description with Latency-Insensitive Interface Combinators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 702–717. https://doi.org/10.1145/3575693.3575701

[20] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[21] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-Rate Image Processing Hardware. *ACM Trans. Graph.* 35, 4, Article 85 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925892

[22] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (Munich, Germany). IEEE Press, 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643582

[23] Intel. [n. d.]. *Intel® High Level Synthesis Compiler*. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[24] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[25] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 865–870. https://doi.org/10.1109/DAC18074.2021.9586329

[26] Liancheng Jia, Yuyue Wang, Jingwen Leng, and Yun Liang. 2022. EMS: Efficient Memory Subsystem Synthesis for Spatial Accelerators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/3489517.3530411

[27] Gangwon Jo, Heehoon Kim, Jeesoo Lee, and Jaejin Lee. 2020. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 295–308. https://doi.org/10.1109/ISCA45697.2020.00034

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[29] Julian Kemmerer. [n. d.]. *PipelineC*. https://github.com/JulianKemmerer/PipelineC

[30] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

[31] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[32] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[33] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. 2020. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 73, 9 pages. https://doi.org/10.1145/3400302.3415644

[34] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *ArXiv* abs/2002.11054 (2020).

[35] Sylvain Lefebvre. [n. d.]. *Silice*. https://github.com/sylefeb/Silice

[36] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, USA, 280–292. https://doi.org/10.1109/MICRO.2014.50

[37] Zizhang Luo, Liqiang Lu, Size Zheng, Jieming Yin, Jason Cong, Jianwei Yin, and Yun Liang. 2023. Rubick: A Synthesis Framework for Spatial Architectures via Dataflow Decomposition. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247743

[38] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[39] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (jun 2023), 25 pages. https://doi.org/10.1145/3591234

[40] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[41] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[42] Open-source [n. d.]. *SpinalHDL*. https://github.com/SpinalHDL/SpinalHDL

[43] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (aug 2017), 25 pages. https://doi.org/10.1145/3107953

[44] Fabrice Rastello. 2016. *SSA-Based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated.

[45] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783720

[46] Frans Skarman and Oscar Gustafsson. 2023. Spade: An Expression-Based HDL With Pipelines. arXiv:2304.03079 [cs.AR]

[47] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 639–651. https://doi.org/10.1145/3373376.3378495

[48] Veripool. [n. d.]. *Verilator*. https://veripool.org/verilator/

[49] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[50] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3050220.3050234

[51] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3431920.3439292

[52] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 27, 6 pages. https://doi.org/10.1145/3061639.3062251

[53] xDSL project. 2023. *xDSL: A Python-native SSA Compiler Framework*. Retrieved October 12, 2023 from https://github.com/xdslproject/xdsl

[54] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 78–88. https://doi.org/10.1145/3490422.3502369

[55] Xilinx Inc. 2023. *Vitis High-Level Synthesis User Guide (UG1399)*. https://docs.xilinx.com/r/2022.1-English/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS

[56] Xilinx Inc. 2023. *Vivado ML*. https://www.xilinx.com/products/design-tools/vivado.html

[57] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/3508352.3549370

[58] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A Scalable High-Level Synthesis Framework with Multi-Level Transformations and Optimizations: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1355–1358. https://doi.org/10.1145/3489517.3530631

[59] Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. 2021. Algorithm-Hardware Co-Design of Attention Mechanism on FPGA Devices. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 71 (sep 2021), 24 pages. https://doi.org/10.1145/3477002

[60] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. *AutoPilot: A platform-based ESL synthesis system*. 99–112. https://doi.org/10.1007/978-1-4020-8588-8

[61] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 180–193. https://doi.org/10.1145/3613424.3614301