Twine: A Chisel Extension for Component-Level Heterogeneous Design

Shibo Chen Yonathan Fisseha Jean-Baptiste Jeannin Todd Austin
University of Michigan, Ann Arbor
{chshibo, yonathan, jeannin, austin}@umich.edu

***Abstract*—Algorithm-oriented heterogeneous hardware design has been one of the major driving forces for hardware improvement in the post-Moore's Law era. To achieve the swift development of heterogeneous designs, designers reuse existing hardware components to craft their systems. However, current hardware design languages either require tremendous efforts to customize designs, or sacrifice quality for simplicity. Chisel, while attracting more users for its capability to easily reconfigure designs, lacks a few key features to further expedite the heterogeneous design flow. In this paper, we introduce *Twine*—a Chisel extension that provides high-level semantics to efficiently generate heterogeneous designs. Twine standardizes the interface for better reusability and supports control-free specification with flexible data type conversion, which saves designers from the busywork of interconnecting modules. Our results show that Twine provides a smooth on-boarding experience for hardware designers, considerably improves reusability, and reduces design complexity for heterogeneous designs while maintaining high design quality.**

I. Introduction

As silicon scaling benefits wane, performance improvement in homogeneous systems is diminishing. Therefore, developers are building heterogeneous systems to meet distinct demands for various workloads. However, heterogeneous design inevitably drives up the complexity and leads to exponential design cost growth [1]. One way to counter complexity explosion is to reuse existing components for new systems.

Although reusing components can effectively reduce cost and enable fast design iteration, current hardware design tools do not aim to produce reusable components that can later be used outside the presumed system. In existing workflows, a component is tightly bound to the particular system it serves: the composing component's timing behaviors and control signals are only sufficient for the current design. To reuse that component outside the original system, one needs to adjust the timing behavior of that component to satisfy new constraints, which requires developers to understand every detail of the design. The necessity to understand every detail significantly lowers productivity and component reusability, a phenomenon that software engineering research has long recognized [2]. Moreover, for a scalable system, the growing size of the system and the large number of components involved would quickly make control signal coordination hard to manage manually. The design complexity will further increase if the composing components do not share the same data format. These situations create new challenges for hardware designers as the design becomes intellectually impenetrable, making designing hardware an expensive and error-prone process.

In our opinion, existing popular hardware description languages (HDL) do not sufficiently address the above problem, leaving designers to toil through the construction, debugging, and deployment of complex heterogeneous designs. Conventional HDLs, such as SystemVerilog [3] and VHDL [4], abstract I/O interfaces as low-level ports and do not distinguish between data and control. The low-level semantics of HDLs pose no requirement on a component's behavior, which puts the responsibility of understanding component behaviors and accommodating their incompatibilities entirely on designers. While some synthesis tools may provide building blocks, it is highly vendor-dependent, making it hard to port designs for different target devices. On the other hand, High-level Synthesis (HLS), such as Vivado HLS [5] and SystemC [6], allows designers to translate software designs directly to hardware representations. Although HLS significantly improves productivity by abstracting away low-level hardware details, recent studies show that, in most use cases, it cannot achieve the same design quality as HDL-based designs [7]. Nonetheless, HDLs remain very popular despite the power of HLS, thus, in this work, we seek to improve the heterogeneous design capabilities of the popular HDL Chisel.

Chisel [8] is a Scala-based hardware generation language that has been drawing much attention in recent years. While Chisel brings polymorphism to hardware design and provides standard components to enhance productivity, Chisel does not raise the abstraction level enough to significantly tame the challenges of heterogeneous design. Therefore, designers still need to coordinate control signals by hand and face the same challenges as they would using a more conventional HDL.

To better address the challenges posed to the designers of heterogeneous systems, we designed Twine. Built upon Chisel, Twine preserves all the power and strength that Chisel already carries, while also introducing the following features that help alleviate much of the burden of designing scalable heterogeneous systems:

- Standard component-level control interfaces that support parameterization, buffering, and reordering.
- High-level specification of component-level producer/consumer relations and dataflow.
- Automation of system-level control signal coordination and flexible data format conversion.

Standard interfaces generalize components to meet most system design requirements. High-level semantics and automation make scaling and reconfiguring systems more manageable. Our evaluation shows that Twine can simplify heterogeneous design

978-3-9819263-6-1/DATE22/©2022 EDAA

```
// Input operands for FMA operation
class FPFMAOps(val numBit:Int,
    val entries:Int) extends Bundle{
    val op1 = Vec(entries,new FP(numBit))
    val op2 = Vec(entries,new FP(numBit))
    val op3 = Vec(entries,new FP(numBit)) }
// Input operands for Sqrt operation
class FPSqrtOp(val numBit:Int,
    val entries:Int) extends Bundle{
    val op = Vec(entries,new FP(numBit)) }
// Output Result
class FPResult(val numBit:Int,
    val entries:Int) extends Bundle{
    val result = Vec(entries, new FP(numBit)) }

class FPFMA(val numBit:Int,
    val entries:Int) extends TwineModule{
① val in = Input(new FPFMAOps(numBit, entries))
    val out = Output(new FPResult(numBit, entries)
    val ctrl = new DecoupledIOCtrl(1,2)
/*Computation logic for FMA(fused multiply-add) */ }

class FPSqrt(val numBit:Int,
    val entries:Int) extends TwineModule{
② val in = Input(new FPSqrtOp(numBit, entries))
    val out = Output(new FPResult(numBit, entries))
    val ctrl = new ValidIOCtrl(2)
    /*Computation logic for Sqrt */ }

class TopLevelDesign extends TwineModule{
③ val in = Input(new FPFMAOps(32,8))
    val out = Output(new FPSqrtOps(16,4))
    val ctrl = new DecoupledIOCtrl(3,2)
    val fma1 = Module(new FPFMA(32,4))
    val fma2 = Module(new FPFMA(32,2))
    val sqrt1 = Module(new FPSqrt(16,2))
    val sqrt2 = Module(new FPSqrt(16,2))
④ in >>> fma1 >>> sqrt1
⑤ in.op1 >>> sqrt2
⑥ TwineBundle(sqrt1, sqrt2, sqrt2) >>> fma2
⑦ fma2 >>> ctrl
}
```

Listing 1: An example of Twine in action. The highlighted lines show the features of Twine extensions, with the rest of the lines showing syntax and semantics of Chisel.

while retaining the high design quality of hand-built, low-level HDL designs.

II. Twine Overview

In this section, we will demonstrate the features of Twine with a running example shown in Listing 1. In this example, the top-level component is computing

$$out := \sqrt{in.op1 \cdot in.op2 + in.op3} \cdot \sqrt{in.op1} + \sqrt{in.op1}$$

The top level design is composed of the four components `fma1`, `fma2`, `sqrt1`, and `sqrt2` as shown in Figure 1. Component `fma1` and `fma2` both take 32-bit floating point inputs and have vectorization of 4 and 2 respectively. The components `sqrt1` and `sqrt2`, on the other hand, implement 16-bit floating point and can take 2 inputs at each cycle.

In a traditional HDL, the developers would be concerned about serialization, type conversion between component interfaces, and excessive control complexity caused by the additional auxiliary codes. To relieve developers from complex

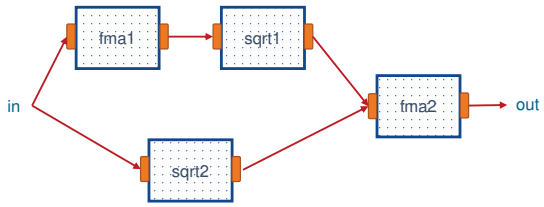


Fig. 1: An example of producer/consumer relations of components

control signal coordination and timing analysis at the high-level, Twine has the following enhancements.

Twine standardizes I/O and control interface. Twine provides four standard interfaces and requires designers to choose one of them when designing their components. The four standard interfaces are `TightlyCoupledIOCtrl`, `ValidIOCtrl`, `DecoupledIOCtrl`, and `OutOfOrderIOCtrl`. Twine imposes timing requirements on how components use the interfaces through sanity checks. ①, ②, and ③ in Listing 1 show examples of Twine interfaces. In Twine, each hardware component is a `TwineModule` which defines three variables:

- ***in***: all input ports of the component
- ***out***: all output ports of the component
- ***ctrl***: the standard control interface used

Such requirements make understanding component behaviors considerably easier. The standard interface enables Twine to implement high-level semantics and automation to simplify the design process. We describe the details of the standard interfaces in Section II-A.

Twine provides expressive high-level semantics to specify dataflow and producer/consumer relations. As we mentioned above, Twine abstracts away the control logic and timing behaviors in the high-level design specification. As such, Twine offers a set of semantics that are more expressive than those in current design languages. It is more intuitive for developers to directly define producer/consumer relations at the high level, rather than connecting low-level ports and control signals. In Twine, a new operator >>> directly specifies the relationships between components and data ports. ④, ⑤, ⑥, and ⑦ in Listing 1 show the use of Twine's high-level connection operator between modules and between separate values and a module.

Twine automates system-level control signal coordination and data format conversion between components. There are two key challenges when assembling pipelines, especially for reconfigurable accelerators: coordinate the control signals and convert data between the boundary of components. These two tasks are often considered 'busy-work,' where designers need to put in considerable effort to implement them correctly while getting little value out of doing them.

Twine automates the assembly stage for developers. Based on the producer/consumer relations and the interface of each component, Twine first inserts necessary buffers and converters as intermediate components between each component. Twine then connects the data ports to the corresponding components or its newly-generated intermediate components. Lastly, Twine coordinates the control signals through the standard interfaces

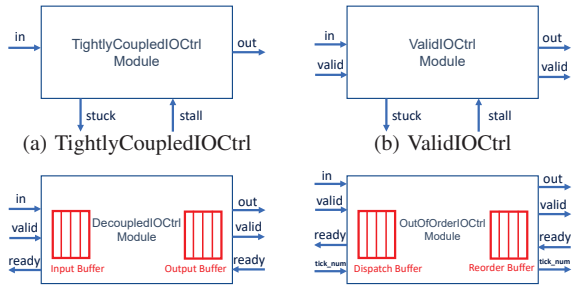
to finish the last step of interconnection. As shown in Listing 1, developers do not have to insert auxiliary logic to adapt different interfaces and re-specify control logic: they are automated in Twine. This feature is particularly useful during design space exploration with parameterization.

A. Control Interface Abstraction

In this section, we describe the four standard control interfaces in Twine and the differences between each interface.

Standard interfaces, like Advanced eXtensible Interface (AXI), have been widely adopted in industry as common practice. However, most standard interfaces, like AXI, are overly complex and inflexible for intra-core communication. Other interfaces, (*e.g.*, `DecoupledIO` in Chisel), only define a set of I/O ports. The language neither requires users to use standard interfaces in their designs nor enforces the way those standard interface ports should be used, which limits the utility of having standard interfaces in the first place.

Enforcing the use of standard interfaces makes hardware components more predictable when reusing them in the high-level design. Twine provides four standard control interfaces that free developers from understanding each component in detail. We describe the four interfaces below and discuss the further extension of those interfaces in Section V.



(a) TightlyCoupledIOCtrl (b) ValidIOCtrl

(c) DecoupledIOCtrl (d) OutOfOrderIOCtrl

Fig. 2: Four standard interfaces in Twine.

1) TightlyCoupledIOCtrl: `TightlyCoupledIOCtrl` is designed for the components with constant latency and simple timing behaviors (*e.g.*, a pipelined multiplier that always takes 4 cycles to compute the result). It has the simplest interface and the lowest overhead. Since the input and output of this interface are tightly coupled together, it cannot accommodate variable latency. As shown in Figure 2a, `TightlyCoupledIOCtrl` has two control signals: ***stall*** and ***stuck***. The component neither consumes nor produces if ***stall*** or ***stuck*** signal is asserted; otherwise, it takes inputs and produces outputs every cycle.

2) ValidIOCtrl: `ValidIOCtrl` is designed to be more flexible than `TightlyCoupledIOCtrl` to support non-deterministic latencies during execution while maintaining low overhead. As shown in Figure 2b, `ValidIOCtrl` implements an additional pair of ***valid*** bits at both ends, which allows the input side to be only loosely-coupled with the output side. The two new signals introduced in `ValidIOCtrl` are ***valid*** signals. The input ***valid*** bit notifies the component of new available inputs; and the output ***valid*** bit signals the external system to handle the new results produced by the component.

	TightlyCoupled	Valid	Decoupled	OutOfOrder
Flexibility	Very low	Low	High	High
Overhead	Low	Low	High	High
Req. fixed lat.*	Yes	No	No	No
Out-of-Order	No	Intra.*	Intra.	Inter.*
Backpressure	Yes	Yes	No	No

TABLE I: Comparison between different interfaces. *Req. fixed lat.: Require Fixed Latency, Intra.: Intra-module, Inter.: Inter-module

3) DecoupledIOCtrl: While `ValidIOCtrl` has increased flexibility, it cannot locally buffer the backpressure from downstream components, so the backpressure will propagate further upstream and stall more components. `DecoupledIOCtrl` is the first control interface in Twine where the inputs and outputs are entirely decoupled. `DecoupledIOCtrl` supports FIFO buffers at both ends of the component to accommodate backpressure locally. Those buffers can be easily customized and generated through parameters during declaration. As shown in Figure 2c, `DecoupledIOCtrl` implements ***valid/ready*** pairs at both ends of the component. ***Valid/ready*** pairs provide a handshake mechanism to adapt to more complex control logic. ① in Listing 1 shows the declaration of a `DecoupledIOCtrl` with 1 and 2 buffer entries on the input end and the output end of the component respectively.

4) OutOfOrderIOCtrl: For operations that have large latency variances (*e.g.*, memory access), out-of-order execution is necessary to exploit parallelism. `OutOfOrderIOCtrl`, shown in Figure 2d, is the interface designed for such use cases. It is one of the most flexible but the most complex interfaces. The request that goes into an out-of-order component would be assigned a ticket number `tick_num` to enable out-of-order process and completion. At the end of execution, the requests would be either automatically reordered with `tick_num` or passed on to another out-of-order component. Buffer management and reordering are transparent to developers so they only need to focus on handling incoming requests.

Comparison Between Interfaces: Table I provides a qualitative comparison of interfaces. As a rule of thumb, components with naïve timing behaviors should use low overhead interfaces; those with complex functionalities and long latency operations should implement the more flexible interfaces.

Adapting and Reusing Existing Designs in Twine: There are two ways to adapt and reuse existing designs, in either Verilog or Chisel, in Twine. First, the developers can extend a `TwineAdaptionModule` and implement one of the four standard Twine control interfaces without changing the internal logic of the existing designs. The extended module can take the fullest advantage of Twine's automation framework and be integrated to the existing Twine ecosystem seamlessly. The second way is to integrate the existing modules manually. Developers can benefit from the part of the system which implements Twine interfaces without sacrificing anything on the parts that do not implement Twine interfaces.

B. Specifying Producer/Consumer Relations and Dataflow

To improve the efficiency of specifying designs, Twine provides a set of semantics to express producer-consumer relations rather than low-level port connections. The producer/consumer model abstracts away the timing behaviors and control signals

component	consumer	producer	p-stakeholder	c-stakeholder
top	fma1,sqrt2	fma2	N/A	N/A
fma1	sqrt1	top	N/A	sqrt2
fma2	top	sqrt1, sqrt2	N/A	N/A
sqrt1	fma2	fma1	sqrt2	N/A
sqrt2	fma2	top	sqrt1	fma1

TABLE II: The relationship between components in a system shown in Figure 1 and specified in Listing 1

and allows users to focus entirely on data dependency. Such design philosophy makes design much more intuitive and enables high-level design automation. Twine needs to collect information of data dependency between each component to synthesize system control logic. To analyze the data dependency, we label each component with ***producer***, ***consumer***, and/or ***stakeholder***, relative to its neighboring components. The consumers are the components that take in values directly from the current module. The producers are the components that the input data originates from. A stakeholder is a component that needs to monitor the status of the other module to make control decisions. For example, when there is one producer with multiple consumers, the producer can only release the results when all consumers are ready to consume. In such a case, each consumer needs to monitor the status of other consumers to avoid duplication and ensure correctness. There are two types of stakeholder: producer-stakeholder (*p-stakeholder*) and consumer-stakeholder (*c-stakeholder*). P-stakeholders share at least one common consumer and c-stakeholders share at least one common producer. Taking the design shown in Listing 1 and Figure 1 as an example, relations between the components are shown in Table II. Component `sqrt1` and `sqrt2` are p-stakeholders for sharing a common consumer `fma2`; `fma1` and `sqrt2` are c-stakeholders for sharing a common producer `top`.

Users only specify the producer/consumer relations in the specification. The stakeholder relations are determined by Twine during elaboration. Users can specify a producer/consumer relationship between any two Twine objects with `>>>` operator as *producer* `>>>` *consumer*. Twine will match the ports on both sides if a bundle of ports are provided and infer the producer/consumer relations from the matching ports.

C. Component Interconnection Automation

Based on the component interfaces and high-level specification of producer/consumer relations, Twine automates two challenging parts of the design: control signal coordination and data format conversion between components.

Control signals in a hardware design determine when a component should consume and release the data. For each component, Twine checks the component status and control its behaviors through a set of pre-defined signals. Twine then elaborates the system control logic based on the interface each component uses and the high-level specification that describes the interconnection between components.

A type conversion happens when the two connecting ports have different data types. At the current stage of development, Twine supports automatic conversion between unsigned/signed integers and common formats of floating point data (*i.e.,* 8-bit, 16-bit, 32-bit, 64-bit). The framework can be easily extended to support more types as needed. Twine will automatically insert a converter to adapt different types when it detects mismatched data types at the two ends of connection.

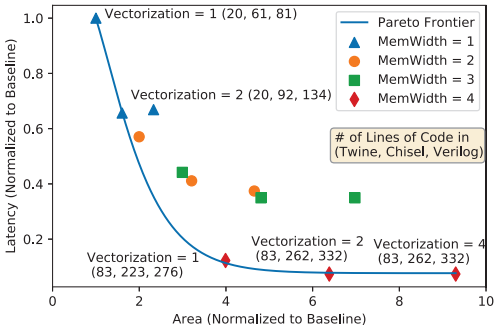


Fig. 3: Pareto chart of a data processing accelerator design space. MemWidth represents the number of requests that the accelerator can read from or write to memory each cycle. Performance is measured as the average latency of processing a batch of 80 requests.

III. Implementation

In this section, we describe the implementation of Twine. Twine is completely backward-compatible with Chisel and supports all Chisel functionalities. Developers can opt in to Twine for their design by simply inheriting from `TwineModule` when declaring components. A `TwineModule` can be used as a normal Chisel module as well. When a Twine operation is evaluated, it translates to a set of Chisel operations, inserts conversion components where needed, and registers the connection information into the Twine profile—a collection of information that guides the synthesis of the system at later stages. The connections between components are only complete after all statements have been evaluated and the context information has been fully collected. We add hooks into the Chisel elaboration phase to initiate Twine synthesis after all statements have been evaluated. Twine then analyzes the producer/consumer relations to finalize connections for control coordination.

The workflow from Twine to FIRRTL are as follows(*Twine-specific steps are marked with* $*$):

1. Generate low-level components using Chisel. *Twine components are implemented with a standard Twine interface.**
2*. Collect high-level information from Twine specification.
3*. Generate necessary buffers and converters. Insert them into the right locations based on the high-level specification.
4*. Reconstruct roles and relations of Twine components.
5*. Coordinate control signals. Interconnect all components.
6. Compile Chisel primitives into FIRRTL representation.
7*. Checks that components meet the required standards to generate correct Twine design (*e.g.*, `TightlyCoupledIOCtrl` module has fixed latency, `ready` and `valid` signals in `DecoupledIOCtrl` are not interdependent).
8. Emit finalized FIRRTL files.

IV. Results

We evaluate Twine by design productivity and design quality.

Design Productivity: To evaluate productivity, we designed a modular data processing component library. The library includes components that are essential to many data processing workloads (*e.g.*, *aggregate*, *column filter*, *boolean*

	Chisel w/ Twine	SystemVerilog	VHDL	Chisel HDL	Bluespec Verilog	SpinalHDL	V++	PyMTL	MyHDL
Reusable Standard Interface	✓	✗	✗	✓̸	✓̸	✓̸	✓	✗	✗
Design Elaboration	✓	✓̸	✓̸	✓	✓̸	✓	✓̸	✓	✓
Component-level Semantics	✓	✗	✗	✗	✗	✓̸	✓	✗	✗
Built-in Queueing	✓	✗	✗	✗	✗	✗	✓	✗	✗
Built-in Control Coordination	✓	✗	✗	✗	✗	✓̸	✓̸	✗	✗
Built-in Data Formatting	✓	✗	✗	✗	✗	✗	✗	✗	✗
Built-in Serialization	✓	✗	✗	✗	✗	✗	✗	✗	✗
Easy Parameterization	✓	✓̸	✓̸	✓	✗	✓	✗	✓	✓

TABLE III: Twine compared to other popular hardware design languages on features that help with heterogeneous design. ✓= fully supported; ✗= not supported; ✓̸= partially supported

generation). We use building blocks to assemble an accelerator that speeds up the processing of complex SQL queries. The accelerator first filters out the selected rows, then adds two columns together, and lastly aggregates results by the key. We explore the design space by exploiting request-level parallelism and data-level parallelism. We exploit the request-level parallelism by adding more components at the front end to filter multiple requests concurrently and exploit data-level parallelism by vectorizing the ALU.

We hand built 12 different design configurations with various memory bandwidths and degrees of vectorization to find the optimal design balances between performance and area. To take advantage of increased memory bandwidth, developers need to put more components onto the system to exploit the data parallelism while considering area and power budget. The degree of vectorization will cause the component interface to change, thus developers need to adapt and buffer the data between the vectorized modules and non-vectorized modules. These two tunable parameters will lead to a process in which the developers need to add, remove, change, and reconnect hardware components, which is common in the design space exploration phase of heterogeneous design.

We first simulate the accelerators in Verilator to estimate the average number of cycles each design takes to process 80 requests. Then we synthesize each configuration to get the clock period and the area estimation. Since we focus on the evaluation of design methodology rather than the design itself, we use performance and area as examples in this experiment. The developer can measure different metrics based on their targets. The performance is measured as the average latency required to process 80 requests. Figure 3 shows the 12 different configurations we explored with regard to performance/area. Through the figure, we are able to identify and call out five Pareto optimal configurations.

We then compared the handwritten implementations in three different design languages: Twine, Chisel, and SystemVerilog. Figure 4 shows the number of components and the total number of lines required to assemble the eight most representative configurations in different languages. We count the number of line changes for each language to specify every target configuration from the baseline design where the accelerator can read one request per cycle and has only one ALU. Thanks to high-level specification and automation, the number of changes in Twine is only affected by the numbers of new components and new producer/consumer relations, while implementations in Chisel and SystemVerilog have to consider timing behaviors and data formats. In Chisel and SystemVerilog, we need to manually

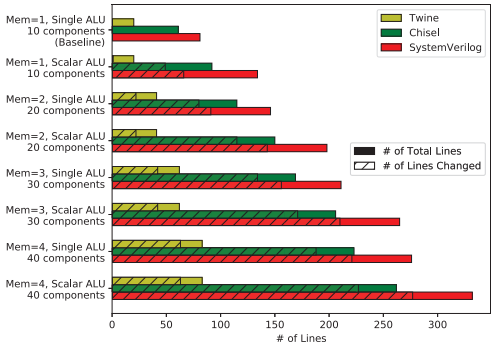


Fig. 4: The number of lines needed for different configurations when implemented in Twine, Chisel, and SystemVerilog. The hatched portion represents the number of changed lines compared to the baseline with the memory bandwidth of 1 row of data and a single ALU that processes 1 row at a time.

coordinate signals for each new component and modify the control signals for the existing ones that are indirectly impacted by the new configuration; in Twine, such tasks are automated during design elaboration. Our results show designers write and modify significantly less code in Twine to assemble a new scalable system with reusable components, resulting in improved productivity and facilitation of design space exploration.

Design Quality: To verify that Twine achieves comparable design quality to Chisel, we ported an in-order RISC-V core [9] from Chisel to Twine. In the original design, the datapath, all stage registers, and the control logic are specified inside one monolithic module. In the Twine design, all function units and stages are encapsulated into 9 separate modules. The top-level module only specifies the dataflow between modules and the communication between the datapath and the cache. Table IV shows a comparison of area and frequency between the two designs synthesized with IBM 45nm SOI12S0 CMOS process. Twine is able to achieve approximately the same performance comparing to the Chisel design. The marginal variance between the two designs is due to slight interface differences.

	RISCV-MINI in Chisel	RISCV-MINI in Twine
Area	727004.94	725937.9 (-0.14%)
Clock Period	0.85 ns	0.82 ns (-3.5%)

TABLE IV: Area and frequency comparison of RISCV-MINI in Chisel and Twine.

Limitations: During our evaluation, we identified a few cases where the highly structured nature of Twine led to negative impacts on design quality. First, the standard interfaces in Twine are *not sufficiently flexible for designs to change processing granularity dynamically*. A component may dynamically decide to either batch requests together for higher

throughput or proceed immediately for lower latency. Since the conversion logic is finalized during generation, the component cannot change it during execution. To get around this limitation, developers can fall back to Chisel to specify control and conversion logic manually or integrate the functionality inside the module. Meanwhile, dynamic scheduling can be easily achieved through software. Second, there may be *missed cross-module optimization opportunities*. Since Twine imposes high modularity requirements, developers may not be able to easily identify cross-module optimization opportunities (*e.g.*, early forwarding). However, such opportunities are usually hard to find and exploit in scalable heterogeneous systems.

V. Future Work

We plan to further extend Twine to address current limitations and further help developers improve productivity:

Customizable interface and protocol: We are working on abstracting and decoupling the interface definition and implementation so that users can easily modify the existing interface and protocol or add new ones into the Twine workflow.

Better verification and debugging capability: Twine enables us to verify properties without going down to the RTL level. We are formalizing semantics and developing a verification framework to verify high-level properties of the designs.

VI. Related Work

In recent years, many hardware design languages have been proposed to improve design productivity. Table III compares Twine with other popular hardware design languages on features that help developers design heterogeneous systems. Chisel [8] enables polymorphism in hardware design workflow and provides the flexibility to dynamically generate hardware designs. Chisel provides optional pre-defined interface, *e.g.*, `Valid` and `Decoupled`. However, the semantics of the pre-defined ports are not specified, thus the automation of control coordination is a challenge. It also lacks the functionality to help developers match the data format and queue the data in the presence of backpressure or out-of-order execution. V++ [10] proposes using compiler-generated communication channel with a multiple-writer-single-reader model as a homogeneous communication mechanism. However, such an interface is overly expensive for light modules, which leads to high and unnecessary performance, power, and area overhead. SpinalHDL [11] provides stream component interface, which is similar to `DecoupledIO`. However, the communication behaviors are only well-defined between two components thus do not support system-level elaboration. SpinalHDL does not provide queueing and data format adapting capability for the stream component interface. BaseJump STL [12] provides a standard hardware component library with templates but lacks the system-level automation between components.

Other hardware design languages are not designed to address the challenges that developers are facing in heterogeneous design. PyMTL [13] and MyHDL [14] are two Python-based HDLs, aiming to make hardware design more accessible by providing a Python frontend. However, they provide comparable gate-level semantics to existing HDLs and lack crucial features for easy meta-programming, which is fundamental for accessible heterogeneous design. Wire sorts [15] is designed to verify the correctness of component interconnections.

In comparison, Twine provides multiple universal interfaces to meet the design needs and provides system-level solutions for large-scale heterogeneous systems rather than partial automation that only works locally. Twine provides automation capabilities that are essential for fast design space exploration.

VII. Conclusion

In this paper, we discuss the emerging challenges hardware designers face in the new age of heterogeneous designs. We propose Twine, a Chisel extension that supports component-level abstraction to improve accessibility and productivity. Twine standardizes component interface, provides high-level semantics, and automates system-level control coordination with inter-component data format adaption. Our results show that Twine is easy to learn, easy to use, and considerably improves productivity for designing heterogeneous hardware.

Our initial version of Twine has been released and is accessible at *https://github.com/Twine-Umich/Twine*. We encourage you to try it out, and we welcome your feedback.

VIII. Acknowledgement

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] A. Olofsson, "Silicon Compilers-Version 2.0," *keynote, Proc. ISPD*, 2018.
[2] P. Hallam, "What Do Programmers Really Do Anyway," *Microsoft Developer Network (MSDN)—C# Compiler*, 2006.
[3] SystemVerilog Standard. [Online]. Available: https://standards.ieee.org/project/1800.html
[4] VHDL IEEE 1076-2019. [Online]. Available: https://standards.ieee.org/standard/1076-2019.html
[5] Vivado HLS. [Online]. Available: https://www.xilinx.com/video/hardware/vivado-hls-tool-overview.html
[6] SystemC. [Online]. Available: https://www.accellera.org/downloads/standards/systemc
[7] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.
[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.
[9] D. Kim. RISCV-Mini. [Online]. Available: url:https://github.com/ucb-bar/riscv-mini
[10] P. C. McGeer, S.-T. Cheng, M. J. Meyer, and P. Scaglia, "Hardware Design Language for the Design of Integrated Circuits," Jul. 16 2002, uS Patent 6,421,808.
[11] C. Papon. SpinalHDL. [Online]. Available: https://github.com/SpinalHDL
[12] M. B. Taylor, "Invited: Basejump stl: Systemverilog needs a standard template library for hardware design," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.
[13] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 280–292.
[14] MyHDL. [Online]. Available: http://www.myhdl.org/
[15] M. Christensen, T. Sherwood, J. Balkind, and B. Hardekopf, "Wire sorts: A language abstraction for safe hardware composition," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 175–189. [Online]. Available: https://doi.org/10.1145/3453483.3454037