Towards Choreographic Programming for Hardware Design

Arjun Vedantham
University of Illinois Urbana-Champaign
USA

William S. Moses
University of Illinois Urbana-Champaign
USA

ABSTRACT

Hardware design makes heavy use of modules that can be composed with each other to enable standardization and reuse. Safely composing hardware modules typically requires passing data between them, which can impose a significant challenge - traditional hardware description languages (HDLs) leave the specific inter-component interfaces up to designers. This requires reasoning about the timing constraints of both input and output signals for each module, often without explicit timing information.

In this work, we propose using techniques from distributed systems to define inter-component communication patterns on a global level. One such distributed programming paradigm, choreographic programming, allows programmers to write a single program specification for an entire system, while obtaining safe per-node implementations without additional effort. When mapped to hardware design, choreographic programming enables the safe composition of distinct hardware modules. This allows hardware designers to write a single multi-component specification without explicit, inter-component interfaces or timing specifications.

1 INTRODUCTION

With the end of Moore's Law, hardware researchers can no longer expect steady improvements in single-core compute performance. As a result of this slowdown, there has been an increased demand for highly parallel systems. New high performance systems have embraced trends in both multicore and distributed systems research, with a special focus on heterogeneous hardware architectures and accelerators. When creating new hardware platforms, designers want high modularity, reusability, and the ability to rapidly iterate on their designs.

Mainstream hardware description languages (HDLs) typically lack the safety and reusability features that are a common part of most software languages. In particular, mainstream HDLs lack standardized inter-component interfaces, leaving it up to individual designers to implement signaling between components correctly. Figure 1 shows how inter-component interfaces can be obscured, especially when reusing a third party's design that lacks the original source code or clear documentation.

Newer HDLs like Chisel [1] include standardized ready-valid constructors, coupled with syntactic sugar (`fire` in Chisel) to make it easier for designers to check multiple readiness signals during data movement. Filament [5] takes this a step further by embedding signal lifetime information into the type system, which is then checked during a compiler pass. While these represent important advances, Chisel's approach does not guarantee timing safety, as it still relies on the designer to check the wire states to determine readiness. Furthermore, while Filament can detect such timing hazards, the compiler cannot automatically determine what the appropriate lifetime of each signal should be.

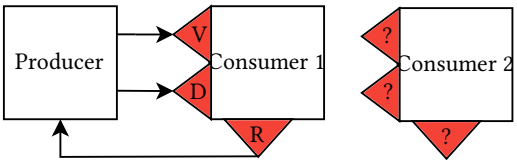

Figure 1: An example of a producer component moving data to consumer components, with signals that producer has to read/write highlighted in red. Consumer 1 has defined input/output signals ("V" for "valid" , "D" for "data", and "R" for "ready"), while consumer 2 does not have control signaling that is visible to the producer component.

These issues of timing safety and communication are readily apparent in distributed systems, where diverse clusters of processing elements or full systems are linked together over a network. ***Choreographic programming*** [4] is a paradigm for programming distributed systems by defining the behavior of a distributed system on a global level, rather than defining how each node within the system behaves. This global specification is then mapped onto the individual nodes through a technique called *endpoint projection*, which ensures that each node implementation remains free of both deadlocks and race conditions. In general, this allows for greater safety in a distributed system, and reduces the burden on programmers by presenting a simple, high level specification of how nodes exchange data. This paradigm has been particularly useful in web programming, where client-server interactions across multiple user sessions can be expressed using a choreographic model.

A choreographic HDL could solve both the interface structure and timing problems. First, it would present a standardized set of communication patterns for moving data between components, using the `comm` and `broadcast` operations from choreographies. Second, these choreographic operations can hide timing constraints by leaving the actual communication protocols between components as an implementation detail of the language or library, rather than having designers or programmers explicitly reason about them. Finally, choreographies could enable better module composition and verification by allowing designers to write a single, global specification, rather than reason about data handoffs on a per-component basis.

In this paper, we:

- Provide a framework for using choreographic programming in hardware design by modeling multi-component designs as distributed systems

LATTE '26, March 23, 2026, Pittsburgh, PA, USA

- Develop an implementation of choreographic programming targeting hardware that can be used within a general purpose programming language
- Show that constraints, operations, and techniques from distributed systems correspond to hardware platforms, and that we can mirror the network topologies of distributed systems with on-chip networks

2 APPROACH

Most implementations of choreographic programming are typically written as domain specific languages (DSLs). This presents a significant obstacle for hardware designers - not only are common hardware constructs (e.g. registers, wire connections) unavailable, but there is also no safe or verifiable way to lower a choreographic specification written in a choreographic DSL into a potentially synthesizable hardware description.

While a choreographic DSL would be difficult to adapt for hardware construction, there are multiple implementations of choreographies as a library module within a general purpose language. HasChor [7] embeds choreographies in Haskell, while ChoRus [2] does the same for Rust. Rust presents another opportunity for targeting hardware - the Calyx [6] hardware intermediate representation (IR) is written in Rust, and it is possible to build new hardware components in Calyx by using its builder APIs in Rust. As such, we chose to implement choreographic hardware design within ChoRus.

3 IMPLEMENTATION

ChoRus defines a `Transport` trait, which allows programmers to define their own wire protocol for interactions between nodes across a distributed system. Other aspects of managing the choreography, (e.g. endpoint projection) are abstracted away.

ChoRus provides default `Transport` implementations for communicating via HTTP or between threads on the same system. Both of these implementations use a blocking queue as a way to buffer messages between any pair of process nodes within the choreography, with each message represented as a serialized JSON string that is held in the queue.

As part of extending ChoRus to target hardware, we reimplemented the queue structure provided in ChoRus as a hardware module within Calyx. While Calyx includes a Python-based generator for parameterizable queues [3], we chose to reimplement this in Rust to best preserve compatibility with other parts of the library. Each hardware FIFO is coupled with a driver module, both of which are instantiated for each node in the choreography (as seen in Figure 2). However, the actual use of these hardware FIFOs is hidden from hardware designers. Instead, the `Transport` trait handles adding and removing data from each component's queue. With this abstraction implemented, hardware designers can then use the default communication operations in ChoRus (`comm`, `broadcast`). Each use of these operators generates a new set of wire assignments that push or pop values from the message queue for each component.

When complete, this choreographic description of the system abstracts away the per-component interfaces, and the Rust based hardware description just describes data handoffs in terms of communications between components. The choreography is then compiled to hardware descriptions in Calyx, which is consequently lowered to Verilog.

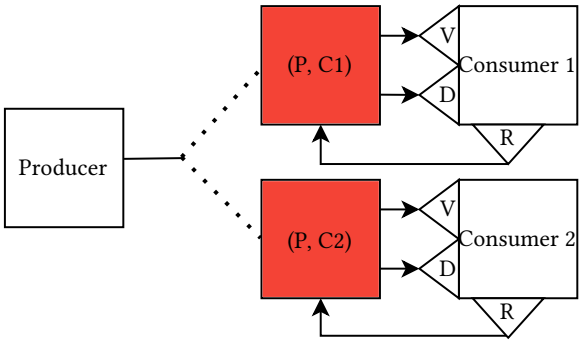

Figure 2: A block diagram of what using a choreography across hardware modules could look like, with a producer node sending a value via `broadcast`. The inter-component signaling (ready-valid) is hidden from the hardware designer, and data is pushed into hardware FIFOs to avoid timing hazards.

```
let data_set = locally(producer, 42);
// 42 is sample data here
let data_get = op.broadcast(producer, &data_set);
    op.locally(consumer, |un| {
        // on the consumer component
        // fetch the value that was sent with op.comm
        let msg = un.unwrap(&data_get);
        set(data)
        // 'consumer' out line is now 42
    });
```

Figure 3: A pseudocode example of what using a choreography across hardware modules could look like, in the style of a ChoRus program specification. Similar to 2, this shows a `broadcast` operation.

4 APPLICABILITY

Dataflow architectures execute instructions when operands for a particular computation are available. Choreographies could serve as a natural way to model data movement between functional units on a processor using a dataflow architecture. A functional unit that produces operands could then use a `comm` or broadcast operation in the choreography to move the instruction product to each of the units that depend on that operand. This particularly fits in with the "dependency injection" technique used by ChoRus to provide dependent parameters to a node while performing endpoint projection.

Similarly, choreographies could be used to implement ***cache coherence*** protocols in hardware. Cache lines could store data by having a producer node use the `comm` operation. Likewise, invalidating outdated data in the cache could be accomplished through `broadcast` operation. ChoRus also has *enclaves*, or specific sets of nodes that respond to group-specific `broadcast` operations. These enclaves could correspond to lines just in the instruction cache,

allowing a CPU to purge incorrectly fetched instructions while preserving the state of the data cache.

Choreographies could also be applied to communication in ***embedded systems***. While most of our discussion has been in the realm of HDLs, this paradigm could also be useful for programming across reconfigurable hardware clusters. For instance, a UART or SPI `Transport` within ChoRus would allow FPGA engineers to write hardware descriptions running that are projected to multiple FPGAs, rather than just arbitrating data movement within a single hardware unit.

5 FUTURE WORK

To fully leverage choreographies within an HDL, we need to enhance the FIFO implementation to handle dynamically changing signals, rather than just constant values. In addition, choreographies in hardware present potential scheduling challenges - components would need to be able to react in real time to operands becoming available from other producing components, which could complicate control flow within each component.

ChoRus's approach to implementing endpoint projection as dependency injection (EPP as DI) means that the per-node behavior is implemented by having each node execute a section of the program specification that corresponds to its particular role in the choreography. One challenge in adopting these ideas to hardware is filtering the specific parts of the choreographic spec that a particular component is responsible for implementing - here, we can take advantage of Calyx's support for multi-component designs and selectively schedule operations like in most ChoRus applications.

Finally, we need to certify that choreographies have only minimal impacts on the overall power, performance, and area of the final hardware design. For simple designs, instantiating hardware queues to implement communication between hardware components could incur significant overhead, but larger-scale dataflow processors may be able to amortize this cost more effectively.

6 ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 2346519. This work was also supported in part by the Alan Turing Institute, and the U.S. Department of Energy, National Nuclear Security Administration under Award Number DE-NA0004266.

Additionally, we thank Lindsey Kuper and Jonathan Castello for sparking the original discussion surrounding the use of choreographic programming in implementing low-level protocols and hardware applications.

REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R Avižienis, J. Wawrzynek, and K. 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

[2] Shun Kashiwa and Lindsey Kuper. 2024. ChoRus: Library-Level Choreographic Programming in Rust. https://users.soe.ucsc.edu/~lkuper/papers/chorus-cp24.pdf

[3] Anshuman Mohan. 2023. *Queues Compiler*. https://docs.calyxir.org/frontends/queues.html

[4] Fabrizio Montesi. 2013. Choreographic Programming. https://www.fabriziomontesi.com/files/choreographic-programming.pdf

[5] Rachit Nigam, Pedro Henrique Azevedo De Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. 7 (2023), 343–367. Issue PLDI. https://doi.org/10.1145/3591234

[6] 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, 2021-04-19). ACM, 804–817. https://doi.org/10.1145/3445814.3446712

[7] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). 7 (2023), 541–565. Issue ICFP. https://doi.org/10.1145/3607849