Using CIRCT for FPGA Physical Design

Designing for performance with higher-level models

John Demme
john.demme@microsoft.com
Microsoft, USA

Aaron Landy
aalan@microsoft.com
Microsoft, USA

ABSTRACT

Wire delays dominate the performance of most FPGA designs. Few designers, however, reason about those delays during initial functional RTL coding, leading to slow designs. During the "timing closure" phase, they are boxed in by the microarchitectural decisions they made early on.

We posit that the RTL model is too low level for serious optimization – that in the modern hardware design world, we need to specify hardware closer to the architectural level. Reasoning about constructs like elastic data pipelines, communication channels, systolic arrays, and FSMs allows compilers to make meaningful optimization transformations.

To that end, we extend the CIRCT compiler framework to enable designing hardware which can be compiler optimized for the physical substrate on which it runs. Using CIRCT benefitted us through network effects – despite CIRCT's youth and being a relatively small network of contributors which we expect to grow.

1 INTRODUCTION

When writing RTL, the designer must always be cognizant of the hardware which is going to be created. If the designer fails to consider the physical implementation and focuses instead on functional correctness via simulation, the resulting design is almost guaranteed to miss performance and area targets – often by drastic amounts. FPGAs, in particular, have limited and highly structured resources into which designs must be mapped, making physical design extremely important. Worse yet, RTL code typically cannot be locally optimized without affecting correctness. A classic example, pipelining can require a cascading series of changes, which often result in "off-by-N-cycles" bugs.

Designs typically start as architectural block diagrams on a whiteboard. They are then refined gradually until one can specify the RTL microarchitecture. The architectural level contains a wealth of information relevant to physical design that is lost while lowering. Our position is that if the high level could be specified, a compiler could use that high level information to better reason about the physical design. We address this need by adding higher-level constructs to CIRCT which can reason about placement and pipeline themselves, replacing the tedious work which often makes the approaches we discuss infeasible at scale.

2 PLACEMENT FIRST PIPELINING

High-performance FPGA design revolves around *deep* pipelines. To determine exactly how deep, we've had success reversing the process: physically design first then incrementally add correctness. Early design typically involves floor planning and running design experiments on "close enough" subdesigns to gather data on precise placements and other strategies. Only once a decent set of placements are found, can we design a properly pipelined microarchitecture.

2.1 Point-to-point connections

Determining the optimal depth of simple connection (based on the distance between the source and sink) seems simple; however, (1) FPGAs tend to have heterogenous routing fabrics with huge discontinuities, often these must be known accounted for in estimates. (2) RTL compiler placement engines tend to place pipeline stages nowhere near where you imagined they'd go, making intuitive latency estimates inaccurate. We can manually place registers to constrain the routing as a mitigation. (3) FPGAs contain several wire classes with wide-ranging propagation delays. For latency-sensitive connections, we can adjust the pipelining down to demand use of faster wires. Most other connections can tolerate artificially increased pipelining, freeing up fast wires for other parts of the design.

2.2 Broadcast networks

When the same data must be communicated to multiple destinations, there exist infinite ways to design the broadcast structure, as demonstrated in Figure 1. In addition to the opportunities with point-to-point connections, the crux of this problem is determining which wires should be shared – its

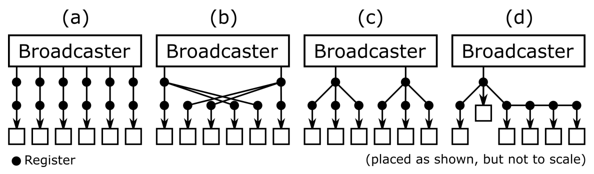

Figure 1: Broadcast networks (a), (b), and (c) are balanced, but the (a) wastes registers and (b) wastes routing resources. (d) is purely systolic and uses fewer resources at the cost of signal latency.

connectivity. The correct choice isn't clear without knowing placements but with them it often becomes plainly apparent.

2.3 Systolic arrays

Despite looking very regular, systolic arrays need only be logically regular – their placement and scheduling can be very irregular. They also tend to be used in bandwidth-driven applications. As such, systolic arrays have many degrees of freedom in terms of placement and pipelining. If we assume the PEs' placements are fixed (for brevity we do), we can determine the pipeline depth of each connection using the ideas discussed in 2.1 and/or 2.2.

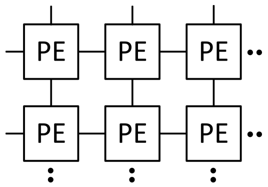

Figure 2: Where should the PEs of a systolic array live on the FPGA? How many cycles of latency should each connection be given, subject to scheduling constraints? What is the resulting input / output schedule?

Traditional arrays have scheduling constraints – link latencies need to be coordinated to ensure the horizontal and vertical data arrive at each PE on the same cycle. Fully buffered arrays (wherein each PE stores the operands), however, have no such requirement. In both cases, the schedule of the inputs and outputs must be computed by the compiler and given to a contoller[1] of some sort. Fortunately, CIRCT has a very flexible scheduling framework[2] which we have just begun using to check schedule validity and solve for the IO schedule.

3 CIRCT AND PYCDE

All of the optimizations discussed here are predicated on compiler awareness of these higher-level constructs, so we require compiler extensions.

3.1 CIRCT

MLIR (Multi Level Intermediate Representation) allows developers to easily define Intermediate Representations (called *dialects*) in a common format. It provides for progressive "lowering" from higher-level dialects to lower-level ones. CIRCT (Circuit IR Compilers and Tools) is a set of MLIR dialects and relevant supporting libraries intended for hardware design. The core dialects represent RTL-level code and are emitted to SystemVerilog. We've added a tcl emitter to specify placements to the RTL compiler and a device database to analyze and manipulate them.

[1]Synthesizing physically-aware datapath controllers is also something we are investigating.
[2]The CIRCT scheduling library is discussed in another paper in this workshop.

Mid-level constructs. In the near future, we'll be adding constructs to represent higher-level ideas: point-to-point channels (2.1), broadcast networks (2.2), and systolic arrays (2.3) to start. CIRCT already has models for (linear) data pipelines and FSMs which we plan to investigate in the future. We'll implement strategies for lowering those higher-level concepts to both instance placements and scheduled RTL dialects using the ideas discussed in the previous sections.

High-level specification. While specifying designs at the granulatity of broadcasts, unscheduled pipelines, systolic arrays, FSMs, et cetera represents a level up for hardware designers, for some users this is too low level. For those users, the CIRCT HLS flow[3] could take advantage of our physical design flows. We are actively exploring a lowering from MLIR's "affine" dialect to systolic arrays that our physical design flow can place and schedule efficiently.

Lowering. It is interesting to note the similarity of the "lowering" approach in CIRCT and the human whiteboard version. Indeed, they are complimentary: our constructs allow users to guide (or at first, fully specify) their lowering while checking for correctness and preserving semantic information.

3.2 PyCDE

We birthed the Python CIRCT Design Entry API as a strongly opinionated binding to CIRCT. PyCDE's only job is to ease surfacing CIRCT constructs to designers, differentiating it from Python HDL projects like Magma and MyHDL which intend to be Python-based RTL alternatives. Indeed PyCDE is not yet ideal for specifing any sort of math operations, preferring instead to act as a glue API, adeptly mixing externally defined RTL and CIRCT constructs.

Currently, PyCDE fully supports specifying lower-level CIRCT constructs – RTL level and instance placements – as well as provides access to the device database. Just these low-level constructs yield benefit by allowing designers to easily specify and query placements. As we add mid-level constructs, we will expose them out through PyCDE for designers to compose into existing designs.

4 CONCLUSIONS

We're not far enough down this road to have any "conclusions," but our experience thus far justify: (1) High-performance FPGA designs require some amount of microarchitecture--affecting physical design – placements without pipelining will only get you so far. (2) Verilog and other RTL-level languages encourage and indeed require brittle code thus make important optimizations more difficult than necessary – a specification at a higher level allows the compiler to help and ensure correctness. (3) Extending CIRCT for physical design has made realizing our goals vastly easier, provided unanticipated opportunities for future expansion, and allows others to easily use (and contribute) to our work. We advise considering CIRCT for your next compiler project!

[3]CIRCT HLS is discussed in another paper at this workshop