PRGA: An Open-source Framework for Building and Using Custom FPGAs

Ang Li
Dept. of Electrical Engineering
Princeton University
Princeton, NJ, USA
angl (at) princeton (dot) edu

David Wentzlaff
Dept. of Electrical Engineering
Princeton University
Princeton, NJ, USA
wentzlaf (at) princeton (dot) edu

***Abstract*—In this era where Moore's Law is approaching its finale, industry has started looking for alternatives to conventional CPUs in order to meet the exploding needs of more diverse, complex and evolving applications. FPGAs are one of the most promising alternatives, offering performance, programmability, and flexibility. However, FPGAs are primarily available as packaged chips from a small number of manufacturers. The tool chains of these FPGAs remain closed-source, and the internal architectures remain a mystery to the public. As a consequence, academia and the open-source community face great difficulties trying to get involved in the development of FPGAs and their tools. In this paper, we present Princeton Reconfigurable Gate Array (PRGA), a highly customizable, scalable, and complete open-source framework for building and using custom FPGAs. The front-end of PRGA, the PRGA Builder, generates synthesizable RTL for a user-defined FPGA that can be fed to the ASIC design flow to enable taping out stand-alone or embedded FPGAs. PRGA Builder features high customizability, supporting heterogeneous logic blocks, custom IP cores, custom routing networks, etc.; it also features high scalability, scaling up to billions of basic elements. PRGA Builder also generates a set of files for the back-end of PRGA, the PRGA Tool Chain, which can synthesize, place & route, and generate the bitstream for a target RTL design using several open-source CAD tools. The bitstream can then be used to program the generated FPGA so as to implement the target design. We have tested PRGA with a few small-scale FPGA architectures and target designs. Preliminary results show that the runtime of the PRGA Builder scales linearly with the total number of logic elements and wiring resources, and the memory usage grows very slowly as the FPGA becomes larger.**

***Index Terms*—open-source, FPGA**

This material is based on research sponsored by the NSF under Grant No. CCF-1453112, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7852. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), the NSF, or the U.S. Government.

I. Introduction

As Moore's Law approaches its end-game, the increasing gap between the needs of emerging applications and the hardware capacity forces programmers to look for alternatives to conventional CPUs for high-performance computing. Specialized ASICs, which deliver the highest performance for a specific small set of applications, suffer from high non-recurring engineering (NRE) cost and low flexibility, making them unsatisfactory for constantly evolving applications. Domain-specific accelerators like GPUs balance between flexibility and performance, yet are only feasible in a small number of domains where applications are composed of regular computation patterns. In comparison, FPGAs provide both high performance and flexibility, making them the most promising alternatives to CPUs. Unfortunately, FPGAs are primarily available as packaged chips from a small number of manufacturers. The tool chains of these FPGAs are expensive and closed-source, and the internal architecture of them are kept as commercial secrets. As a consequence, academia and the open-source community are hampered and unable to explore potential improvements of FPGA architectures and their tool chains. Great efforts have been devoted to developing open-source tool chains [1]–[5] which enable customized configuration of the commercial FPGAs. However, the architecture of these FPGAs remain a mystery, and there are no open-source frameworks for exploring custom FPGA designs.

Princeton Reconfigurable Gate Array (PRGA) is aimed at filling this void. As shown in Fig. 1, PRGA consists of two parts: the front-end - the PRGA Builder, a highly customizable and scalable Python API for users to describe, optimize, and build their own custom FPGAs; and the back-end - the PRGA Tool Chain, a complete RTL-to-bitstream flow for mapping target RTL designs onto the FPGAs built with the PRGA Builder. The PRGA Tool Chain uses Yosys [3] for synthesis, VPR [4] for pack, place, & route, and PRGA Bitgen (a bitstream generator specially designed for PRGA) for bitstream generation. Notably, the PRGA Tool Chain **does not modify** any of the external tools, so it is easy to adapt to new versions or substitutes of these tools.

PRGA is modularized and extensible. Both PRGA Builder and the PRGA Tool Chain are composed of many fine-grained

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

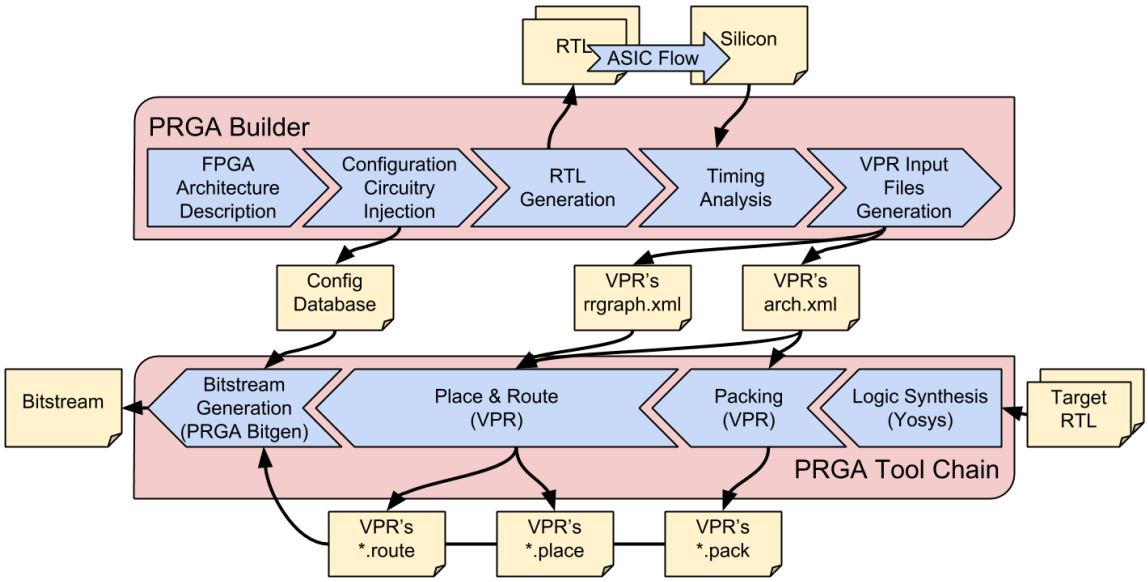

Fig. 1. PRGA Workflow: PRGA Builder generates RTL and other files needed by the PRGA Tool Chain for a custom FPGA; the PRGA Tool Chain generates the bitstream for a given target design for an FPGA built with the PRGA Builder.

steps, and users may deviate from the flow in any step, make changes, then merge back. This allows PRGA users to further customize the behavior of PRGA and achieve their custom goals. It's also welcomed for potential contributors to add additional or alternative steps to the flow.

The following are our key contributions:

1) Creation and release of a highly customizable, scalable, and complete framework for building and using custom FPGAs.
2) Description of the PRGA workflow and the extensibility available at every step of the flow.
3) Presentation of an end-to-end use case of the framework, showing its customizability and extensibility.

PRGA is released and available for download at https://github.com/PrincetonUniversity/prga. More details about PRGA can be found at https://prga.rtfd.io.

II. FPGA Background

A. FPGA Architecture

Almost all modern FPGAs are island-style, that is, organized in a two-dimensional array of configurable logic blocks (CLB) and IO blocks (IOB), as shown in Fig. 2. Each CLB/IOB may contain multiple look-up tables (LUT), flipflops, and/or other IP cores such as block RAMs (BRAM) and DSPs. The connections between these logic elements are controlled by configuration circuitry that can be reprogrammed for different applications. Between the blocks run plentiful wiring resources (① in Fig. 2), which can be connected to CLB/IOBs through connection blocks (CB, ② in Fig. 2) or connected to other wire segments through switch blocks (SB, ③ in Fig. 2). The manner that these connections are made are controlled by configuration circuitry.

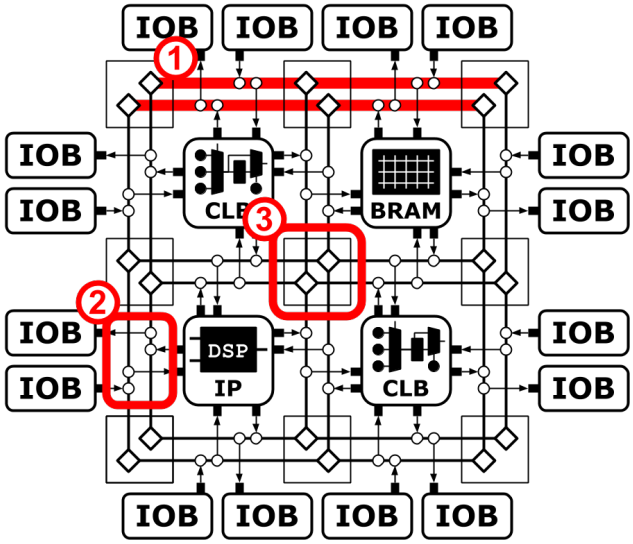

Fig. 2. Island-style FPGA Layout: FPGA is organized as a 2D array of CLB/IOBs and wire segments (①). Wire segments are connected to CLB/IOBs (②) or other wire segments (③) via configurable switches.

B. Tool Chain

To implement a target RTL design, FPGA programmers need to run a series of CAD tools to generate the bitstream file that programs the FPGA, similar to compiling a C program into an executable binary, only more complex. The minimum set of CAD tools needed are: a synthesis tool which translates RTL into gates; a packing tool which packs gates into CLBs; a place & route tool which places CLBs at physical locations and uses wire segments to connect them; a bitstream generator which combines the outputs from the tools above and

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

generates the final bitstream file.

III. Related Work

A. Open-source/Academic FPGA

Although there are no end-to-end, open-source framework for easily building and using custom FPGAs, there have been a few previous academic attempts on building non-commercial FPGAs. Chaudhuri *et al.* [6] successfully embedded a small 8x8 FPGA into a SoC, but this embedded FPGA is not customizable. Liu [7] developed a minimally customizable FPGA with tool support, yet the workflow is relatively rigid and inextensible. Tang *et al.* [8] focused on generating accurate SPICE models for FPGAs described in VTR, which does not produce synthesizable RTL. Grady *et al.* [9] modeled a Stratix IV-style synthesizable heterogeneous FPGA by extending VTR [4], and they adopted a similar approach to this work by generating synthesizable RTL for the custom FPGA and providing support all the way to bitstream generation.

B. Open-source CAD tools for FPGA

On the software-side of the FPGA ecosystem, academia and the open-source community have achieved much greater success.

1) Logic Synthesis: There are quite a few successful open-source/free logic synthesis tools available now. Yosys [3] and ODIN II [2] are two stable and reliable examples, both based on ABC [1]. Yosys supports more up-to-date HDL standards such as Verilog-2005 and has an active community maintaining it. ODIN II is the default logic synthesis tool in VTR [4].

2) Pack, Place, & Route: Two place & route tools are popular among FPGA fans: VPR [4] and next-pnr [5]. VPR is the core component of VTR. It was started a decade ago and is still actively developed by a team of maintainers. Next-pnr (preceded by arachne-pnr) is an emerging alternative to VPR. Both of them have been tested in a few open-source projects aimed at customizing the RTL-to-bitstream flow for commercial FPGAs.

3) Complete CAD Flow Targeting Commercial FPGAs: Academia and the open-source community have devoted great efforts revealing the logical architecture and configuration file formats of commercial FPGAs. Torc [10] reverse engineered the configuration files and bitstream format of multiple Xilinx FPGAs including all Virtex, Virtex E, Virtex 2, Virtex 2 Pro, Virtex 4, Virtex 5, Virtex 6, Virtex 6L, Spartan 3E, Spartan 6, and Spartan 6L devices, making fine-grained custom configuration possible. Verilog-to-bitstream [11], [12] is an extension to VTR which can generate bitstream for a specific Xilinx Virtex 6 FPGA. More projects followed the methodology and achieved great success, including Project IceStorm [13] targeting Lattice iCE 40 FPGA, Project X-Ray [14] targeting Xilinx 7-series FPGAs, and Project Trellis [15] targeting Lattice ECP5 FPGA.

IV. PRGA Builder

A. Overview

```
width, height = 6, 6
arch = ArchitectureContext(width, height)
# create routing resources
arch.create_segment(name='L1', width=4, length=1)
... # add more wire segment types
arch.create_global(name='clk', is_clock=True)
# create CLB
clb = arch.create_logic_block(name='CLB')
## add ports to the CLB
clb.add_input(name='I', width=4, side=Side.left)
clb.add_output(name='O', width=1, side=Side.right)
clb.add_clock(name='CLK', side=Side.bottom,
        global_='clk')
## add sub-instances to the CLB
clb.add_instance(name='LUT', model='lut4')
clb.add_instance(name='FF', model='flipflop')
## add connections
clb.add_connections(clb.ports['I'],
        clb.instances['LUT'].pins['in'])
clb.add_connections(clb.instances['LUT'].pins['out'],
        clb.instances['FF'].pins['D'])
... # add more connections to CLB
# create IOBs
for side in Side.all():
    io = arch.create_io_block(
            name='IO_'+side.name.upper())
    ... # add ports, instances, and connections to IOB
# place CLB/IOBs
arch.array.place_blocks(block='CLB',
        x=1, y=1, endx=width-1, endy=height-1)
arch.array.place_blocks(block='IO_LEFT',
        x=0, y=1, endy=height-1)
... # place more blocks
# bind global wire to a specific IOB
arch.globals['clk'].bind(0, 1, 0)
# populate routing resources
arch.array.populate_routing_channels()
arch.array.populate_routing_switches(
        default_fc=(0.25, 0.5))
# run passes
flow = Flow(arch)
os.mkdir('rtl')
flow.add_pass(VerilogGenerator('rtl'))
flow.add_pass(VPRArchdefGenerator(
        open('archdef.vpr.xml', 'w')))
... # add more passes
flow.run()
```

Listing 1: Building a simple FPGA with the PRGA Builder

PRGA Builder is the front-end of PRGA, consisting of a set of Python APIs for users to describe and build their own custom FPGAs. Code listing 1 shows a simple example that builds a small FPGA with 32 LUT4s and 32 flipflops using the PRGA Builder API.

B. Architecture Description

The API for architecture description is designed with the following goals:

- **Versatility**: The API is easy to use when describing simple FPGAs, but also highly customizable, capable of describing advanced or even commercial-class FPGAs. This is achieved by providing **reasonable default values** for the large amount of customizable properties of the FPGAs.
- **Scalability**: Modern FPGAs can be as large as billions of logic elements. The API is designed to scale to that

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

level and remain customizable.

The rest of this section covers the architecture description API step by step.

1) Configurable Logic Block: PRGA Builder supports highly customized CLB/IOB structures. Users are free to add any number of LUTs, flipflops, custom IP cores, or any combination of them to a CLB/IOB, then add arbitrary configurable connections between these logic elements. After describing a CLB/IOB, MUXes can be automatically created with a simple command.

2) Grid Layout: PRGA Builder allows users to create any number of different CLB/IOB structures, as well as CLBs with different sizes. After describing all different CLB/IOB structures, users are provided with a few simple commands to lay out the blocks into a 2D grid.

3) Routing Resources and Switches: Two types of routing resources are supported: wire segments and global wires. PRGA Builder enables very fine-grained customization of routing switches. Users can place different connection/switch blocks at different positions, and these blocks may have very different routing patterns.

4) Performance Concerns: PRGA Builder is implemented in pure Python. We chose Python because it's portable, expressive, standardized, and simple, making PRGA Builder more accessible to users. However, Python is neither a high-performance nor a memory-efficient programming language. In order to achieve high scalability, especially with respect to memory, PRGA Builder leverages the regularity in FPGAs as much as possible. The internal data of PRGA Builder can be serialized using Python's `pickle` module, enabling data reuse and the development of external tools.

C. Building Passes

After users are done describing their custom FPGAs, the rest of PRGA Builder is organized as **Passes**. A pass may modify the FPGA architecture, perform some optimization, or generate files. Users can choose any passes they need, pause before any pass, or implement their own passes. Passes necessary for generating all files required by the PRGA Tool Chain are: configuration circuitry injection, RTL generation, timing analysis, VPR input files generation, and configuration database generation. The rest of this section discusses all these passes in detail.

1) Configuration Circuitry Injection: The configuration circuitry injection pass automatically injects configuration circuitry into CLB/IOBs, connection blocks, and switch blocks. By design, PRGA Builder can be easily extended to support different types of configuration circuitry. For now, the only supported configuration circuitry type is a serial flip-flop chain. Note that for each configuration circuitry type, a corresponding configuration database schema and a bitstream generator must be implemented. The former is discussed in section IV-C5, and the later is discussed in section V-D.

2) RTL Generation: The RTL generation pass generates RTL for the CLB/IOBs, connection blocks, switch blocks, the top-level module of the FPGA, as well as all logic elements, MUXes, configuration circuitry components, etc., except for user-defined IP cores, for which users must provide their own RTL that matches their description. This pass uses Jinja2 [16], a text templating framework, and users may write their own templates to customize RTL generation. For now, RTL is generated in Verilog under the Verilog-2005 standard.

3) Timing Analysis: Timing information of all the logic elements and switches is required for the place & route tool to correctly implement a target RTL design, yet the timing information is only available after executing the ASIC flow. Unfortunately, due to licensing and complexity reasons, the ASIC flow cannot be distributed with the PRGA framework and must be completed by the users themselves, using sample scripts we provide that use commercial CAD tools such as Synopsys DC/ICC. PRGA Builder provides a timing engine base class for users to implement their own timing information extraction mechanism. For logic verification-only use cases where real timing information is not necessary, PRGA Builder includes a random timing information generator, which generates random numbers within a given range, so that users can run the entire PRGA framework without having access to any ASIC flow.

4) VPR Input Files Generation: The PRGA Tool Chain uses VPR for pack, place, & route, which requires an architecture description file, *arch.xml*, and takes an optional routing resource graph description file, *rrgraph.xml*, as input. These files can be generated by the PRGA Builder with one simple command.

5) Configuration Database Generation: PRGA Bitgen needs detailed information about the configuration circuitry in order to translate VPR's pack, place, & route results into bitstreams. The configuration database generation pass generates a database file in Protocol Buffers [17] so as to pass the information from PRGA Builder to PRGA Bitgen. We chose Protocol Buffers because it is a stable, compact, and cross-language data serialization format with enforced data schema, so PRGA developers don't need to worry about validation, data layout, etc. when implementing different types of configuration circuitry.

6) Optimization: Besides the necessary passes mentioned in the sections above, optional optimization passes can be added to optimize or customize FPGAs. For example, PRGA Builder includes an optimization that inserts buffers that disable all top-level outputs during FPGA configuration.

7) Performance Concerns: Similar to the discussion in section IV-B4, all building passes are optimized for scalability. All passes that generate files periodically free temporary data and flush rendered text onto disk so as to reduce memory usage. This is critical especially for generating VPR's *rrgraph.xml*, which grows quickly as the size of the FPGAs gets larger.

V. The PRGA Tool Chain

A. Overview

The PRGA Tool Chain wraps a series of open-source tools and provides a complete RTL-to-bitstream flow for FPGAs built with the PRGA Builder.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

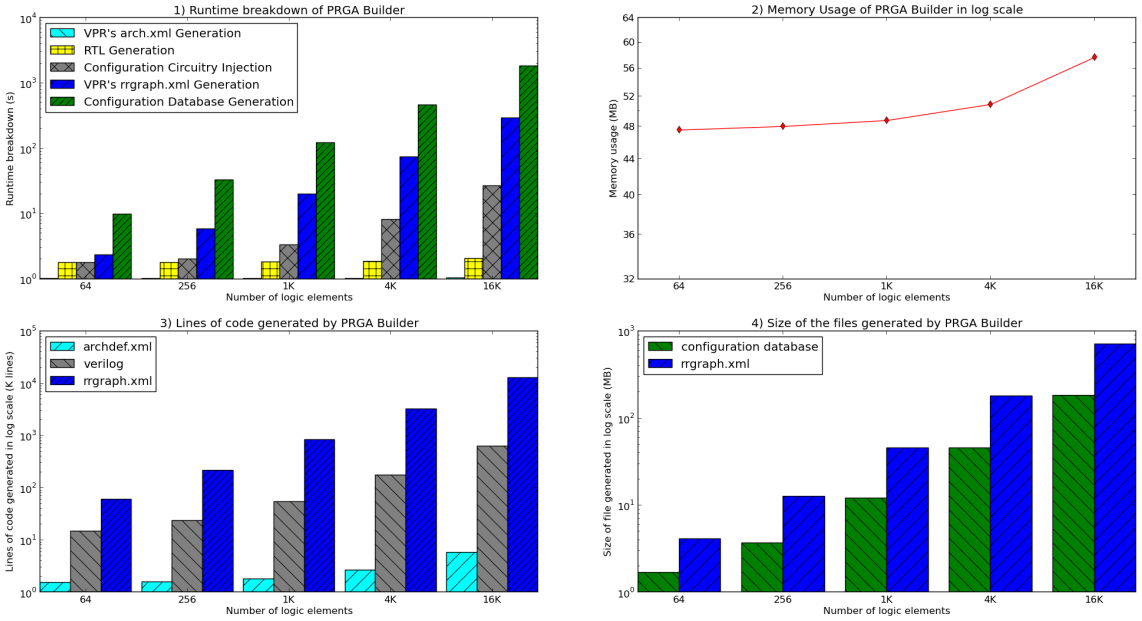

Fig. 3. Runtime Breakdown, Memory Usage and the Size of the Files Generated by the PRGA Builder

B. Synthesis

By default, the PRGA Tool Chain uses Yosys for synthesis, but users are allowed to use any flattened, synthesized circuit in BLIF format instead.

C. Pack, Place, & Route

For pack, place, & route, the PRGA Tool Chain uses VPR. However, unlike many of its predecessors, the PRGA Tool Chain **does not modify VPR**, but only runs VPR with command line arguments and input files generated by the PRGA Builder. By staying decoupled from VPR, the PRGA Tool Chain can utilize every improvement or new functionality of newer versions of VPR. PRGA is using an up-to-date version of VPR on its development trunk, and will move to VPR 8 as soon as it is released.

D. PRGA Bitgen

Since bitstream format is dependent on the configuration circuitry, generic bitstream generation is not feasible. Therefore, we developed PRGA Bitgen, a C++ framework for creating bitstream generators specifically for FPGAs built with the PRGA Builder. Bitstream generators created with PRGA Bitgen are able to combine the configuration database generated by the PRGA Builder, the synthesis result generated by Yosys, and the pack, place, & route results generated by VPR, in order to generate bitstreams. For now, only one bitstream generator for a serial flipflop-chain type configuration is included in the PRGA Tool Chain.

VI. Evaluation

A. Case Study: BCD2BIN Converter

We have verified the complete PRGA flow with a few target RTL designs. One of them is a BCD2BIN converter which includes a 3-stage FSM, a 4-bit counter, multiple shifters and combinational logic units, covering many fundamental components used in digital circuits. Two different FPGAs are built for this design, both containing 72 LUT4s, 72 flipflops, one clock tree, and 23 general-purpose, bi-directional I/O pins. The only difference is in the routing resources: in one FPGA, all wire segments are 1 tile long, while in the other FPGA, a small quantity of the wire segments are 2 tiles long, so that they are able to connect more CLB/IOBs via less switches, resulting in better timing.

The PRGA Tool Chain is used to generate the bitstream implementing the target design. A wrapper module with configuration loading logic is created for simulation purpose. We use Synopsys VCS to simulate both the target design and the FPGA implementing the target design. Our simulation shows that the complete PRGA flow is capable of generating RTL that can be simulated for custom FPGAs, as well as producing bitstreams that implement custom designs on the generated FPGAs.

B. Runtime, Memory and Disk Usage of the PRGA Builder

We built FPGAs of different sizes with the PRGA Builder using one core of a 2.5GHz Intel Ivy Bridge Xeon E5-2670 v2 processor. We measured the runtime, memory and disk usage of the PRGA Builder. The CLB/IOB structure, layout, and wire segment-to-CLB ratio are the same for all the FPGAs, so that the size of the FPGA can be represented by the total number of logic elements.

Fig. 3 shows how runtime, memory usage, and the size of the files generated by the PRGA Builder scale as the size of the FPGAs becomes larger. The runtime scales linearly with the total number of logic elements The most time-consuming passes are the *rrgraph.xml* generation and configu-

ration database generation passes. These are inevitable because one XML element is required for each wire segment and each programmable switch in *rrgraph.xml*. The configuration database has similar requirements because it is used to parse VPR's outputs.

The memory usage consists of two parts: a fixed memory overhead of the Python interpreter and imported libraries, and the memory usage of the PRGA Builder. Although the memory usage of PRGA Builder inevitably scales linearly with the number of logic elements, it grows much slower than the lines of code generated and the size of files written on disk.

VII. Enabled Applications

We hope PRGA can enable many research directions by providing FPGA researchers/programmers the ability to build and use their own custom FPGAs. Two possible applications of PRGA are listed below, but they are just a small set of all potential applications.

A. Exploring FPGA Architectures

Is LUT6 the best size? What should be the best ratio of BRAMs and LUT/flipflops? How many wire segments are optimal given an FPGA architecture? There is still plenty of space left for optimization in the design space of FPGAs. By generating many FPGA designs using PRGA and evaluating these designs, researchers may gain some insight and answer these questions.

B. Specialized FPGAs

Between FPGAs and domain-specific accelerators, is there any possibility for a new type of architecture which balances programmability and performance? Some may think of coarse-grained gate arrays (CGRA), which resemble FPGAs in which the majority of logic elements are arithmetic IP cores. How should FPGA tool chains be modified to program CGRAs? This can be answered with FPGAs built with PRGA.

VIII. Future Work

PRGA is still under development, so many features and functionalities are not implemented or perfected yet. This section discusses two of the most important works in progress.

A. Implementation with ASIC Flow

As discussed in section IV-C3, the ASIC Flow is not distributed with the PRGA framework. However, it is necessary to verify that the RTL generated by PRGA Builder is suitable for the ASIC Flow. Kim *et al.* [9] discussed a few challenges when implementing an FPGA using standard cell library methodology, such as inevitable combinational loops of wire segments. The RTL generation pass is designed with this in mind so that the RTL generated is synthesizable and hierarchical. Our next step is to run the complete ASIC Flow and tape out some FPGAs built with the PRGA Builder.

B. Other Configuration Circuitry Types

A serial flipflop chain is a good, simple, proof-of-concept type of configuration circuitry, but it is inefficient with regard to area and power. Our next step is to add other configuration circuitry types, for example latch array-based configuration circuitry.

IX. Conclusion

In this paper, we presented PRGA, a highly customizable, scalable, and complete open-source framework for building and using custom FPGAs. The front-end of PRGA, the PRGA Builder, and the back-end of PRGA, the PRGA Tool Chain, are introduced in detail. Preliminary results show that the runtime of the PRGA Builder scales linearly with the total number of logic elements and wiring resources, and the memory usage grows very slowly as the FPGA becomes larger.

References

[1] B. L. Synthesis and V. Group, "ABC: A System for Sequential Synthesis and Verification, Release 70930." http://www.eecs.berkeley.edu/~alanmi/abc/.

[2] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "ODIN II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 149–156, May 2010.

[3] C. Wolf, "Yosys Open SYnthesis Suite." http://www.clifford.at/yosys/.

[4] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, pp. 6:1–6:30, July 2014.

[5] YosysHQ, "next-pnr." https://github.com/YosysHQ/nextpnr, 2018.

[6] S. Chaudhuri, S. Guilley, F. Flament, P. Hoogvorst, and J.-L. Danger, "An 8x8 Run-time Reconfigurable FPGA Embedded in a SoC," in *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, (New York, NY, USA), pp. 120–125, ACM, 2008.

[7] H. J. Liu, "Archipelago - An Open Source FPGA with Toolflow Support," Master's thesis, EECS Department, University of California, Berkeley, May 2014.

[8] X. Tang, P. Gaillardon, and G. D. Micheli, "FPGA-SPICE: A simulation-based power estimation framework for FPGAs," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp. 696–703, Oct 2015.

[9] B. Grady and J. H. Anderson, "Synthesizable Heterogeneous FPGA Fabrics," in *2018 International Conference on Field-Programmable Technology (FPT)*, pp. 225–232, Dec 2018.

[10] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-source Tool Flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 41–44, ACM, 2011.

[11] R. K. Soni, N. Steiner, and M. French, "Open-Source Bitstream Generation," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 105–112, April 2013.

[12] E. Hung, F. Eslami, and S. J. E. Wilton, "Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 45–52, April 2013.

[13] C. Wolf and M. Lasser, "Project IceStorm." http://www.clifford.at/icestorm/.

[14] SymbiFlow, "Project X-Ray." https://github.com/SymbiFlow/prjxray, 2018.

[15] SymbiFlow, "Project Trellis." https://github.com/SymbiFlow/prjtrellis, 2018.

[16] A. Ronacher, "Jinja2." http://jinja.pocoo.org/, 2014.

[17] Google, "Protocol Buffers." https://developers.google.com/protocol-buffers/.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io