SymbiFlow and VPR: An Open-Source Design Flow for Commercial and Novel FPGAs

Kevin E. Murray
University of Toronto

Tim Ansell and Keith Rothman
Google

Alessandro Comodi
Antmicro

Mohamed A. Elgammal and Vaughn Betz
University of Toronto

***Abstract*—As the benefits of Moore's Law diminish, computing performance, and efficiency gains are increasingly achieved through specializing hardware to a domain of computation. However, this limits the hardware's generality and flexibility. Field-programmable gate arrays (FPGAs), microchips which can be reprogrammed to implement arbitrary digital circuits, enable the benefits of specialization while remaining flexible. A challenge to using FPGAs is the complex computer-aided design flow required to efficiently map a computation onto an FPGA. Traditionally, these design flows are closed-source and highly specialized to a particular vendor's devices. We propose an alternate data-driven approach, which uses highly adaptable and retargettable open-source tools to target both commercial and research FPGA architectures. While challenges remain, we believe this approach makes the development of novel and commercial FPGA architectures faster and more accessible. Furthermore, it provides a path forward for industry, academia, and the open-source community to collaborate and combine their resources to advance FPGA technology.**

Digital Object Identifier 10.1109/MM.2020.2998435
Date of publication 28 May 2020; date of current version 30 June 2020.

Published by the IEEE Computer Society

Moore's Law has tracked our ability to perform increasingly efficient and complex computation over the past 55 years, enabling general purpose devices like CPUs (and more recently GPUs) to continually improve performance and power efficiency. However as the traditional benefits of manufacturing process technology scaling diminish, so are the performance and efficiency gains of these devices. At the same time, demand for computation from domains such as machine learning and wireless signal processing continues to rapidly increase.

This has led computer architects to investigate domain-specific computing architectures, which focus on efficiently implementing a specific domain of related computational applications. While such approaches offer significant benefits, these are largely derived from their specialization and lack of flexibility. Furthermore, developing such architectures is expensive (designing and fabbing a custom chip may cost hundreds of millions of dollars), and is risky (e.g., what if some unsupported operation becomes required?). As a result, relatively few application domains are stable enough and garner sufficient financial support for this approach.

Field-programmable gate arrays (FPGAs) present an alternative approach, which combines flexibility and general applicability, with the performance and efficiency benefits of domain specific architectures (DSAs). Instead of designing a DSA which must then be manufactured as an Application Specific Integrated Circuit (a process which usually takes years), a DSA can be implemented and reprogrammed onto an FPGA in a few hours or days. As shown in Figure 1, an FPGA consists of programmable logic blocks (which can implement arbitrary boolean logic functions), data storage (FFs and BRAMs), and a programmable routing fabric to interconnect them. This enables an FPGA to be quickly reprogrammed to implement an arbitrary digital circuit.

The architecture of FPGAs themselves continues to evolve in many ways, such as the integration of increasingly heterogeneous blocks such as digital signal processing blocks, diverse I/O controllers, embedded networks-on-chip and more. The evolution of FPGAs and the creation of FPGAs with unique features to benefit new

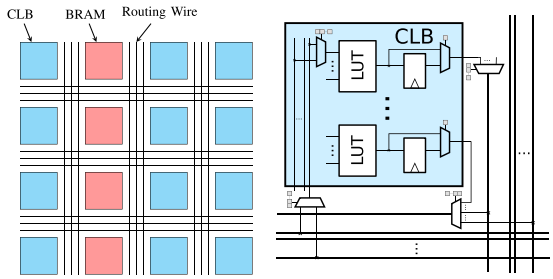

Figure 1. (Left) An FPGA consisting of configurable logic blocks (CLBs) for computation and Block RAMs (BRAMs) for storage, along with programmable routing to interconnect them. (Right) Each CLB contains lookup tables (LUTs) and flip-flops (FFs). Routing Muxes are configured to interconnect elements within each block, or interblock routing wires.

domains is hindered by the very complex computer-aided design (CAD) system needed to map a high-level computation description to the millions of configuration bits which control the FPGA. The creation of such tooling is a daunting task: FPGA vendors such as Xilinx and Intel employ more software engineers than hardware engineers in their FPGA divisions to develop their (closed-source) CAD systems.[1] Furthermore, without a prototype CAD system for each FPGA architecture of interest, an FPGA architect cannot quantitatively evaluate new architectural ideas.

In this article, we discuss the SymbiFlow project, which seeks to create an open-source CAD flow for FPGAs that can be used not only to program commercial FPGAs, but also to evaluate new FPGA architectures. This dual goal creates a challenge. On the one hand, creating a programming file for a specific FPGA requires that every device detail is perfectly supported by the CAD system. On the other hand, for a CAD system to be able to target a wide range of FPGA architectures it must be very flexible and avoid chip-specific coding. We show that a data-driven approach is possible: SymbiFlow can fully map designs to the commercial Xilinx Artix 7 devices with open-source synthesis, placement, routing, and bitstream generation tools that can still be retargeted to other (existing or novel) FPGAs. In this article, we quantify the current feature support and result quality of this flow when targeting commercial devices. While much remains to be done, we believe SymbiFlow has the potential

both to allow a much larger community to develop CAD flows for current FPGAs and to make the development of new FPGA architectures faster and more accessible.

RELATED WORK

FPGA CAD flows serve two main purposes: implementing designs in a completely specified FPGA architecture and quantitatively evaluating new hardware architectures before their creation. Each FPGA vendor has a CAD flow for the first purpose—generating a programming file for their devices—and several also have CAD flows to evaluate new architecture ideas for future microchips, such as Intel/Altera's FPGA Modeling Toolkit (FMT).[2] These tools are closed source however, so they cannot be used by academic or other researchers to test new CAD algorithms, to investigate new architectures, or to implement designs on novel FPGAs.

In this article, we discuss the SymbiFlow project, which seeks to create an open-source CAD flow for FPGAs that can be used not only to program commercial FPGAs, but also to evaluate new FPGA architectures.

Some interfaces to commercial tools have been documented, allowing some interaction between external open source tools and these commercial CAD flows. Torc[3] and RapidSmith II[4] use Xilinx Design Language (XDL) to pass intermediate CAD result information to Xilinx's (now legacy) ISE CAD flow. RapidWright[5] uses similar techniques to read and write partial CAD results to and from Xilinx's Vivado CAD tool, enabling customization of some portions of the design flow. While these interfaces are helpful to the open source community, they do not address all usage scenarios. First, device programming information is not exposed. Second, the code bases to which they interface are closed, and the devices targeted are limited to existing commercial devices. This places limits on how much the CAD flow can be changed and precludes investigation of or support for novel FPGA architectures.

To overcome these limitations, several standalone open-source frameworks have been created. The VTR framework combines Odin II for verilog synthesis, ABC for technology mapping and VPR for packing, placement, and routing.[6] VPR takes a human-readable description of an FPGA architecture as input and has been used for a wide variety of FPGA architecture research in academia and industry (Altera's FMT was originally derived from VPR). VPR/VTR have also been used as a framework for investigating many new CAD algorithms, allowing researchers to implement or modify only their novel part of the CAD flow.

While VTR has been extensively used for research, it has historically seen less use for programing actual devices. VTR-to-Bitstream[7] proposed a toolchain based on VTR that can program a Virtex 6 commercial FPGA, using Xilinx's tools only for the final programming. Several start-ups have made private copies of VPR and developed implementation tool flows for their devices on top of it—but their tools are closed source. Similarly, several academic research teams created novel spatial architectures, which use VTR as their CAD flow,[8] but they all required significant custom coding to create a full device model and programming flow. This has led to a significant duplicate work across these projects. Hence, a major recent focus of the VTR project, and this article, has been to enhance the tool flow in a data-driven way to support full implementation flows for both existing *and* future novel FPGAs with little (or ideally no) custom coding. The OpenFPGA project further builds on the VTR infrastructure to allow automatic layout of a full FPGA, enabling a complete idea-evaluation-layout-programming flow.[9]

Nextpnr[10] is another open source FPGA placement and routing tool that has been created to enable a complete open-source FPGA implementation flow for the Lattice Ice40 and ECP5 devices. Unlike VTR, its main purpose is to target existing commercial FPGA architectures with an open-source flow; as such it is more amenable to custom coding.

In this article we detail Symbiflow: enhancements to VPR and a data-driven bitstream generator that allow a complete open-source implementation flow for a Xilinx Artix 7 commercial device, without significant custom coding. We believe this new flow fills an important

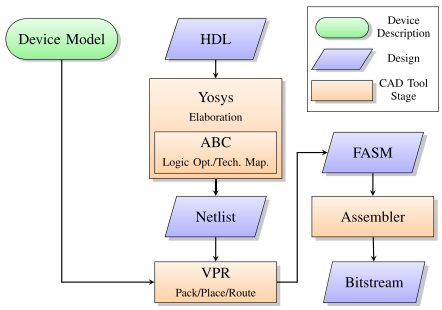

Figure 2. Design flow.

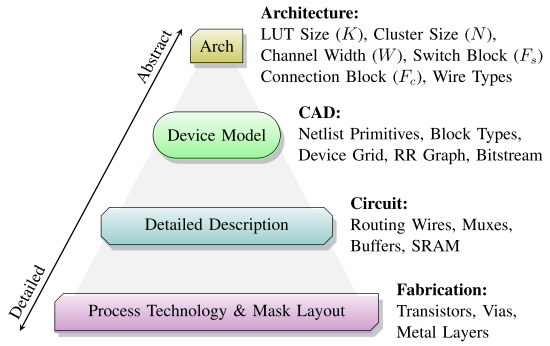

Figure 3. FPGA architecture representation.

gap by enabling rapid creation of full implementation CAD flows for both existing and new FPGA devices.

DESIGN FLOW

The open source design flow we focus on is shown in Figure 2. The end-user provides a behavioral specification of the computations they wish to perform in a hardware description language (HDL). This description is then transformed by several tools to map the computation to the target architecture, finally producing a bitstream used to configure the FPGA. This mapping occurs in several stages.

First, Yosys converts the behavioral HDL into a circuit consisting of soft logic (boolean equations) and hard architectural primitives (like adders, multipliers, and RAMs). For hard primitives Yosys' technology mapping, library lowers generic HDL operations (such as addition) to architecture specific primitives (e.g., full adders).* The soft logic is then optimized by ABC and technology mapped to the primitive computational elements (e.g., LUTs).

VPR then takes the technology mapped netlist along with a detailed model of the FPGA device and determines, where to place each circuit element (placement) and how to interconnect them (routing) while minimizing the wiring required and maximizing circuit speed. The resulting circuit implementation is then converted to a low-level FPGA Assembly (FASM) description, which can be directly translated into the binary bitstream which programs an FPGA.

*Further lowering is later performed to ensure primitives match the VPR device model.

Specifying Architectures

FPGA architectures can be described at various levels of detail, as shown in Figure 3. While the lower level descriptions are more complete they are also unstructured, making it impractical to have CAD tools target them directly. Instead CAD tools operate on a "Device Model," which captures the key information about an FPGA architecture/device in a structured manner. This allows tools to effectively optimize and implement designs based on the targeted device model. In our data-driven approach, the device model serves as an intermediate representation (IR), which can model a range of hypothetical and commercial FPGA devices.

FPGA architects developing new architectures often take a top-down approach, using high-level descriptions to quickly describe and evaluate a wide range of architectures and architectural parameters. These descriptions leave many aspects underspecified, leaving it up to the tools to automatically fill in the details (i.e., the "Elaborate" step in Figure 4). This approach is highly productive for architecture exploration and research, and if an FPGA device matches the data model well, it can produce a reasonably accurate device model. For instance, Murray *et al.*[11] used VPR to model commercial Intel Stratix IV FPGAs. However, this method can make it challenging to precisely model the details of a specific device.

To implement designs targeting commercial FPGAs and produce valid bitstreams (i.e., the bottom half of Figure 4) requires modeling the target device with bit-level accuracy. This requires a bottom-up approach, which constructs the device model from the low-level device details

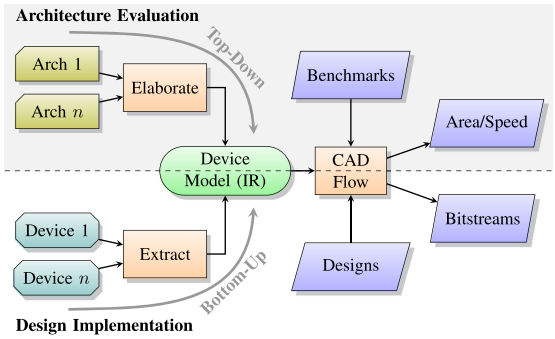

Figure 4. Architecture and implementation flows.

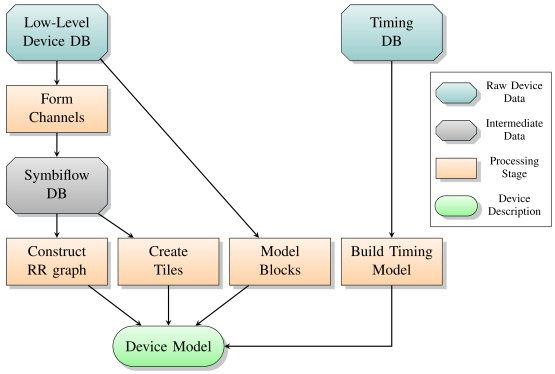

Figure 5. Architecture model generation.

(i.e., the "Extract" step in Figure 4). With such a device model, the CAD flow in Figure 2 can then be used to map applications to that device.

Recently, these low-level device descriptions have become available for some FPGAs, including Lattice's ICE and ECP5 families, and Xilinx's Artix 7. These low-level descriptions provide detailed information about the FPGA structure (logic and how routing resources interconnect), and how their use is described in the bitstream. However as noted previously, merely having access to the low-level details is insufficient to successfully target these devices. The CAD flow requires higher level context and structure to correctly implement and effectively optimize applications for the device. The process of converting these low-level device descriptions into the higher level device model can be challenging, as the low-level details are relatively unstructured. This process is typically performed through a combination of automated processing and human-driven modeling ("Extract" in Figure 4). While much of this processing can be reused within a particular device family (or potentially across related families from a particular vendor), each vendor makes a variety of different architectural, implementation, and terminology choices. The broadness and significance of these differences have often been at the root of concerns about whether flexible architecture agnostic tools like VPR can target commercial FPGAs.

Enhancements to Target Commercial FPGAs

A variety of new tools and enhancements are required in order to enable a fully open-source FPGA tool flow, which can be reused and retargeted to multiple devices.

VPR Enhancements: In order to support commercial devices, we have extend VPR's device model and flexibility to capture the details required to produce valid bitstreams. The changes include:

- generic device grids (to model the precise layouts of commercial devices)[6];
- support for loading the detailed routing architecture from a file (rather than building it from a high-level specification)[6];
- nonconfigurable routing graph edges (to model multisegment and nonlinear wiring)[6];
- support for routing clocks on dedicated clock networks;
- extensions to timing analysis to capture clock network delays and fan-out related loading effects.

Device Model Extraction: As described above, low-level device databases exist for some commercial FPGAs but are very low level and unstructured. This makes them an unsuitable description for place and route tools, which aim to optimize higher level characteristics like resource usage, wirelength, and timing. It is therefore necessary to extract additional higher level structural information, which tools like VPR can target.

Figure 5 illustrates the major steps in Symbiflow required to convert the low-level device database for Xilinx's Artix 7 family into an appropriate device model for use in VPR. The resulting device model captures all the details correctly and includes the higher level structural

Table 1. VPR 8 quality and runtime targeting Stratix IV model on Titan benchmarks.

LABs	Routed WL	Routed CPD	Pack Time	Place Time	Route Time	Total Time
0.95	1.26	1.20	1.18	1.00	0.34	0.83

Geomean of 20 mutually implementable Titan benchmarks,[11] normalized to Intel Quartus 18.0.[6]

information required for effective place and route optimizations.

The first step involves extracting higher level structural information about the interblock routing network (form channels). For instance, it is important to align the block and routing network coordinate systems, which ensures both the placer and router have aligned understandings of what components of the device are physically close together.

The second step (Construct RR Graph) builds the routing resource (RR) graph describing what routing wires exist between blocks and how they interconnect. Each routing resource is also annotated with information about its location and the characteristics of the switches to which it connects. Some routing architecture details also need to be translated to fit in the RR Graph description. For instance, the Artix 7 family includes "U"-shaped wires, which are converted into three linear wires connected by nonconfigurable edges.

The third step (Create Tiles) defines the routing interface between the RR Graph and the different block types (e.g., CLBs, BRAMs) within the FPGA. It also constructs the device grid which specifies what block types exist at each coordinate in the FPGA, and tags them with FASM metadata used by the assembler to relocate their configuration bits within the bitstream.

The fourth step (Model Blocks) creates a model of the logic and internal routing within each block type. For instance, it would model the resources available within a CLB (e.g., LUTs, FFs, Adders) and the connectivity between them. These components are all tagged with FASM metadata used to drive bitstream generation.

These steps are sufficient to generate a functionally correct device model, which captures the low-level details of the device and can be used with VPR to implement designs and generate valid bitstreams for real Artix 7 devices. However, additional information is required to enable VPR to optimize circuit implementations well.

To enable timing-driven optimization, it is necessary to provide a timing model for the various architecture components (routing wires, LUTs, FFs etc.). This is done by creating a timing model ("Build Timing Model") from a database of timing information extracted from Vivado. The timing model is used by Tatum,[12] VPR's Static Timing Analysis (STA) engine, for timing analysis and to drive timing-based optimizations.

Additionally, at each stage it is necessary to extract higher level information to enable VPR to effectively optimize the use of various routing and logic resources. For example, grouping the numerous wires in the RR graph into a smaller number of "types," which share similar electrical and connectivity characteristics (e.g., rare but fast long wires, common but slower and more flexible short wires), enables VPR to tradeoff which signals use the different wire types.

Generic Bitstream Assembler To create a generic bitstream assembler, a generic FPGA assembly (FASM) format was devised. FASM is a textual format which lists "features" to enable within the FPGA fabric. A feature might be the LUT truth-table or the input wire selected by a routing mux. Generally a FASM feature will enable one or more bits, and may also require that one or more bits remain cleared in the output bitstream.

FPGA bitstream contents can be roughly broken down into three categories: primitive block configurations (e.g., LUTs, FFs), intrablock routing muxes, and interblock routing muxes. During device model extraction, information about the FASM features associated with each of these categories are tagged as metadata on the relevant components. The corresponding FASM features are then set based on component usage in the final design implementation.

Verification and Correctness

Multiple levels of verification have been performed to ensure the resulting bitstreams are functionally correct on various example and benchmark designs. This includes programming

Table 2. Symbiflow & VPR quality and runtime on Artix 7 (XC7A50TFGG484).

Benchmark	Netlist Primitives	CLBs		RAMB18s		Crit. Path Delay (ns)		Synth. Time (sec)		Pack & Place Time (sec)		Route Time (sec)		Assembly Time (sec)		Total Time (sec)	
picosoc	8,108	895	(1.14×)	3	(1.50×)	18.47	(1.93×)	29.2	(0.49×)	62.0	(8.85×)	11.1	(0.56×)	9.5	(1.19×)	115.9	(0.83×)
murax	4,558	392	(1.23×)	4	(0.67×)	12.22	(1.45×)	18.4	(0.48×)	23.0	(7.67×)	3.9	(0.28×)	9.0	(1.00×)	58.3	(0.58×)
top_bram_n3	3,595	361	(4.25×)	1	(1.00×)	14.56	(3.17×)	14.8	(0.62×)	9.9	(9.86×)	4.2	(0.38×)	6.5	(0.81×)	37.8	(0.49×)
top_bram_n2	2,619	264	(3.72×)	1	(1.00×)	14.07	(2.84×)	13.6	(0.52×)	7.2	(7.16×)	3.0	(0.25×)	6.2	(0.77×)	32.4	(0.40×)
top_bram_n1	1,661	162	(2.49×)	1	(1.00×)	13.36	(2.57×)	11.0	(0.48×)	7.9	(7.86×)	1.2	(0.10×)	5.2	(0.65×)	27.6	(0.36×)
dram_test_64x1d	1,035	115	(1.47×)	0		10.95	(2.12×)	7.8	(0.30×)	6.6	(6.56×)	0.6	(0.06×)	4.3	(0.53×)	21.1	(0.26×)
GEOMEAN	2,902.7	292.2	(2.08×)	1.6	(1.00×)	13.75	(2.27×)	14.5	(0.47×)	13.2	(7.92×)	2.7	(0.21×)	6.5	(0.80×)	41.1	(0.45×)
% TOTAL								32.7%		38.8%		8.0%		14.1%		100.0%	

Crit. Path Delay analyzed by Xilinx Vivado 2017.2.
Ratios normalized to Xilinx Vivado 2017.2 in parentheses.

the bitstreams onto real devices and testing functionality (e.g., manually, or using built in self-test), reimporting the produced implementation into Vivado, and formally verifying the equivalence of the postsynthesis and post-place-and-route netlists.

FLOW QUALITY

To quantitatively evaluate the quality and run-time of our open-source design flow, we compare it in two scenarios: one where VPR understands the higher level structure of the architecture, and another which captures the full details of a commercial Xilinx device.

In the first scenario we use a mixed open and closed source flow, where we use Intel's Quartus (closed-source) for logic synthesis and technology mapping, and either Quartus or VPR (our open-source place and route tool) to perform the physical implementation.[†] In this scenario VPR targets a somewhat abstracted model[6] of Intel's Stratix IV FPGA family, since the low-level details of Stratix IV are unavailable. Both tools are used to implement the Titan benchmark suite,[11] which consists of modern large scale benchmarks (90K-1.9 M netlist primitives) which use all the types of heterogeneous resources available in the Stratix IV family. Table 1 shows that compared to Quartus, VPR uses similar amounts of logic (LABs) and requires a comparable amount of run-time to implement all designs. However VPR's implementations use approximately 26% more wiring and operate 20% slower than those produced by Quartus. These results show that when our tools comprehend the overall device (including higher-level structure), their implementation and optimization algorithms achieve good run-time and reasonable result quality compared to a highly tuned architecture specific tool like Quartus.

In the second scenario, we use the fully open-source Symbiflow (with VPR) design flow shown in Figure 2 to target a Xilinx Artix 7 device for which the full low-level device details are available, and compare the results with Xilinx's closed-source Vivado 2017.12 tool suite.[‡]

The results are shown in Table 2. Here, we see that Symbiflow uses roughly double the amount of logic (CLBs) to implement these designs, and produces circuits which have $2.3\times$ longer critical paths. A significant portion of this gap is due to logic synthesis and technology mapping, with Vivado producing smaller and better optimized netlists.[¶] From a tool run-time perspective, Symbiflow's average run-time is lower than Vivado's. Symbiflow spends less time on synthesis, more time on placement and completes routing faster than Vivado.

It is important to note that the results in Table 2 are achieved while targeting the full details of Artix 7, and as a result produce valid bitstreams which can be programmed onto an Artix 7 device without *any* closed-source tooling. We believe that many of the run-time and quality issues can be alleviated through either improving the modeling of Artix 7's high-level characteristics, or improving the adaptability of VPR's algorithms.

CONCLUSION and FUTURE WORK

FPGAs offer a compelling approach to achieve the performance, power, and cost benefits of

[†]This allows us to evaluate only the impact of placement and routing (since the same synthesis result is used).

[‡]Note the reported critical path delays are calculated by Vivado's timing analyzer for both design flows.

[¶]When Vivado uses the same netlist as VPR, VPR uses 24% fewer CLBs than Vivado (indicating Vivado packs the design less densely than VPR) and the critical path delay gap reduces to $1.6\times$.

domain specific architectures while retaining the flexibility to adapt to changing computational requirements and supporting a wide range of application domains. However, the complex CAD flow required to target FPGAs has hindered progress in this direction. We believe open-source tools for FPGAs are an important step toward addressing this challenge, allowing the broader academic, commercial, and open-source communities to pool their efforts in order to advance this technology.

We have shown that with a data-driven approach, it is possible to create architecture agnostic tools, like VPR, which can target the full details of commercial FPGAs and produce valid bitstreams. These same tools enable FPGA architects to experiment with and evaluate new FPGA architectures, while also providing a ready-made functional CAD flow for targeting such devices.

We have shown that with a data-driven approach, it is possible to create architecture agnostic tools, like VPR, which can target the full details of commercial FPGAs and produce valid bitstreams.

However, there is plenty of work still to be done. When architectures match VPR's data model well, it is possible to achieve reasonable implementation quality and run-time (see Table 1). However, for architectures which do not match well the results are more mixed. It will be important to continue improving the quality of architecture captures given to VPR, and continue extending VPR's device model to allow better descriptions of such architectures.

Working from low-level device descriptions is challenging as many of the important structural characteristics of an FPGA architecture are implicit. This requires complex processing and manual efforts to extract this higher level structure from the low-level data, but is key to achieve high-quality optimization and reasonable run-times. If vendors provided this higher level device information along with their chips this effort would be eased significantly.

Finally, there remains significant room for improving the quality and run-time of VPR's optimization algorithms. There are many identified areas for improvement,[6] which we hope the community will collaborate together to enhance.

ACKNOWLEDGMENTS

This work was supported in part by Google, the NSERC/Intel Industrial Research Chair in Programmable Silicon, as well as NSERC CGS-D and Ontario Graduate Scholarships. We would like to thank Q. Xu, X. Hou, and Y. Wang for assistance collecting benchmark data, as well as the numerous community members who have contributed to the development of VPR and Symbiflow through writing documentation, filling bugs, and submitting code fixes and enhancements.

REFERENCES

1. S. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.
2. D. Lewis *et al.*, "The stratix^{TM}10 highly pipelined FPGA architecture," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 159–168.
3. N. Steiner *et al.*, "Torc: Towards an open-source tool flow," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2011, pp. 41–44.
4. T. Haroldsen, B. Nelson, and B. Hutchings, "RapidSmith 2: A framework for BEL-level CAD exploration on Xilinx FPGAs," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 66–69.
5. C. Lavin and A. Kaviani, "RapidWright: Enabling custom crafted implementations for FPGAs," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2018, pp. 133–140.
6. K. E. Murray *et al.*, "VTR 8: High performance CAD and customizable FPGA architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, p. 55, 2020, Art. no. 9. [Online]. Available: https://doi.org/10.1145/3388617
7. E. Hung, F. Eslami, and S. J. E. Wilton, "Escaping the academic sandbox: Realizing VPR circuits on xilinx devices," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2013, pp. 45–52.
8. DARPA, "Reconfigurable imaging (ReImagine)," 2016. [Online]. Available: https://www.darpa.mil/attachments/Final_Compiled_ReImagineProposersDay.pdf

9. B. Chauviere *et al.*, "OpenFPGA: Complete open source framework for FPGA prototyping," in *Proc. Workshop Open Source Des. Autom.*, 2019, pp. 367–374.
10. D. Shah *et al.*, "Yosys+nextpnr: An open source framework from verilog to bitstream for commercial FPGAs," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2019, pp. 1–4. [Online]. Available: http://doi.org/10.1109/FCCM.2019.00010
11. K. E. Murray *et al.*, "Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, pp. 10:1–10:18, 2015.
12. K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *Proc. Int. Conf. Field-Programmable Technol.*, 2018, pp. 110–117.

Kevin E. Murray is currently working toward the Ph.D. degree at the University of Toronto, and has been the lead developer of the VTR project since 2014. His research interests include FPGA CAD and architecture, machine-learning-enhanced CAD, and modular hardware design flows. Murray received the M.A.Sc. degree in electrical and computer engineering, and the B.A.Sc. degree in engineering science from the University of Toronto. He is a student member of IEEE. Contact him at k.murray@mail.utoronto.ca.

Tim Ansell is a Software Engineer at Google. His research interests include open-source EDA tooling, open-source RTL design, and software speed hardware accelerator development. Ansell received the B.A. degree in philosophy and cognitive science and the B.Eng. degree in information technology and telecommunications from the University of Adelaide. He is a member of IEEE. Contact him at tansell@google.com.

Keith Rothman is a Software Engineer at Google. His research interests include open-source EDA tooling, open-source RTL design, and software speed hardware accelerator development. Rothman received the B.S. and M.S. degrees in aerospace engineering from California Polytechnic State University—San Luis Obispo. Contact him at keithrothman@google.com.

Alessandro Comodi is a Software Engineer at Antmicro. His research interests include FPGAs, EDA tools, and hardware design methodologies. Comodi received the Master's degree in computer science and engineering from the Politecnico di Milano. Contact him at acomodi@antmicro.com.

Mohamed A. Elgammal is currently working toward the Ph.D. degree at the University of Toronto. His research interests include reinforcement learning, CAD tools, and FPGAs. Elgammal received the B.Sc. and M.A.Sc degrees (Hons.) in electronics engineering from Cairo University. Contact him at mohamed.elgammal@mail.utoronto.ca.

Vaughn Betz is a Professor and NSERC/Intel Industrial Research Chair at the University of Toronto. His research interests include programmable hardware architectures and CAD tools. Previously he was Senior Director of Software Engineering at Altera Corporation, where he was one of the architects of the Quartus II CAD system and the Stratix and Cyclone FPGA architectures. Betz received the B.S.E.E. degree from the University of Manitoba, the M.S.E.E. degree from the University of Illinois at Urbana–Champaign, and the Ph.D. degree in electrical and computer engineering from the University of Toronto. He is a Fellow of IEEE. Contact him at vaughn@eecg.utoronto.ca.