Invited Paper: RapidWright: Unleashing the Full Power of FPGA Technology with Domain-Specific Tooling

Chris Lavin and Eddie Hung
AMD Research and Advanced Development
Advanced Micro Devices, Inc.
chris.lavin@amd.com, eddie.hung@amd.com

***Abstract*—The era of domain-specific computing has created an environment that drives for the best performance possible from silicon and tools. Novel and unique implementation strategies for FPGAs that may have been infeasible in the past are now sought after in the search of better performance or faster compile time. The ability to optimize for domain-specific attributes in FPGA implementations is now more important than ever and both industry and research institutions need better ways to fully harness the programmable potential of FPGAs.**

This paper describes RapidWright, an open source framework from AMD Research and Advanced Development, and how it enables design implementation to be customized on commercial FPGA devices. RapidWright enables strategies previously infeasible or impossible to designers and provides sufficient flexibility to leverage domain-specific attributes in their applications for the highest performance, compile-time, or timing closure predictability. We demonstrate how the RapidWright framework has been a fundamental enabling factor in a variety of practical research efforts that have led up to 30% higher quality-of-result (QoR) and compile time improvements of 5× or greater across a number of applications.

***Keywords*-AMD, back-end tools, FPGA, placement, RapidWright, routing, timing closure, Vivado, Xilinx**

I. Introduction

Computer architects are now widely subscribed to domain-specific architectures as being the only path left for major improvements in performance-cost-energy. As a result, future compilers need to go beyond their traditional role of mapping a design input to a generic hardware platform. Emerging domain-specific compilers must subscribe to a broader view in which compilers provide more control to the end users, enabling customization of hardware components to implement their corresponding tasks. Transitioning into this new design paradigm, where control and customization are key enablers, poses new challenges for such domain-specific compilers.

Today, generic vendor-specific backend EDA compilers are the only available mechanism to realize a broad range of applications in many domains. The necessity for commercial tools to cover a broad range of applications often leads to implementations that do not take full advantage of the underlying hardware. Domain-specific compilers, on the other hand, can potentially deliver near-spec performance by taking advantage of both application attributes and architectural details. This issue is less pronounced for more generic computing platforms such as CPUs due to leveraging open source as an essential component of software development. However, high quality EDA software has remained mostly proprietary. Existing open source attempts do not produce results to be useful at commercial scale. Addressing EDA customization to achieve domain-specific compilers will require collaboration from both industry and the open source community.

This suggests the need for a framework capable of interfacing between closed source vendor backend tools and open source domain compilers. RapidWright [1] is an example of such a framework that enables a new level of optimization and customization for the application architect to further exploit FPGA silicon capabilities focusing on a specific domain. Several efforts have built on RapidWright in order to achieve domain-specific optimizations for specific applications. For example, RapidStream [2] demonstrates 30% higher performance and more than 5× faster compile time for data flow applications. The key enabler for the RapidStream domain compiler is the split-compilation that was made possible for such applications with a latency-tolerant front-end and design entry.

RapidWright offers a broad set of capabilities in an open source framework that provides the level of flexibility needed to address the growing demand for more efficient and performant implementations in a domain-specific era. Another aspect of leveraging the domain-specific approach to EDA is interoperability among other tools both from industry and the open source community. For this reason, we helped develop, prototype and implement the CHIPS Alliance FPGA Interchange Format [3]. This new exchange format is a way for backend FPGA tools—both open source and proprietary—to communicate partial FPGA design implementations as well as provide all the information needed to build a custom backend tool from scratch.

In the remainder of this paper, we describe some of the major components and capabilities within RapidWright. We also share some of the domain-specific implementation successes that have been built on top of RapidWright's framework to improve compile time and/or performance. Ultimately, we believe that there is still tremendous potential in FPGA technology that can be unlocked through the use of domain-specific tooling.

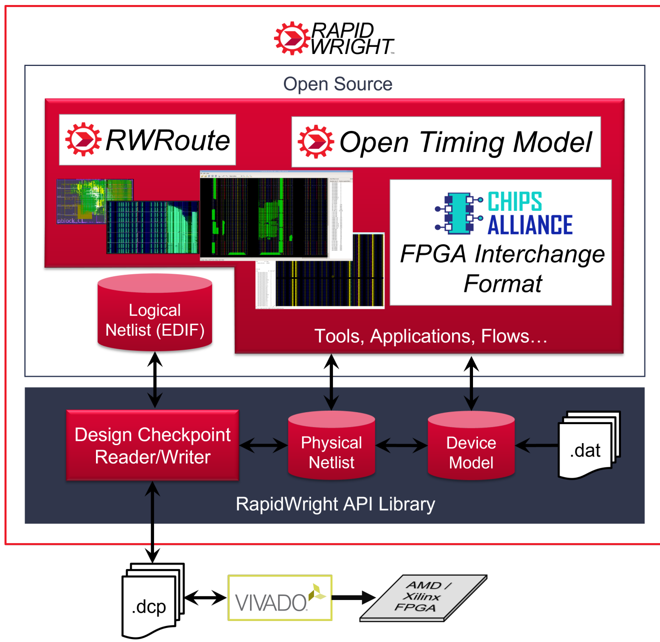


Fig. 1. The RapidWright Framework.

II. The RapidWright Framework

RapidWright is an open source framework that provides a back-end interface to AMD-Xilinx devices that compliments Vivado. Specifically, RapidWright can read and write design checkpoint files (DCPs) or snapshots of a design implementation before, during, or after the place and route stages. It also provides a rich set of APIs and higher level features to optimize and build implementation flows not possible with Vivado alone. By leveraging RapidWright, users are able to achieve higher quality-of-results (QoR), faster compile times and/or better timing closure predictability. RapidWright also offers unique capabilities not currently offered by Vivado such as fine-grained design compose-ability, replication and relocatability and on-the-fly generation (no RTL needed) of placed and routed circuits. This section describes the lower level API sets that enable a set of capabilities and later we will present how these capabilities can be used to together to build full compilation strategies.

A. Fundamental APIs

Fundamentally, there are three categories of information needed to place, route and manipulate a design that is to be mapped onto an FPGA. RapidWright has three rich API sets to satisfy the needs of these three categories: a device model, a logical netlist model and a physical netlist model. The fine details of how these APIs function is beyond the scope of this paper and we refer the reader to additional resources [4] and [5] for further details.

1) Device Model: The device model is the physical representation of the FPGA device and all of its configurable, user-facing resources that can be utilized to implement a user's design. It fully elaborates all device resources and their spatial relationship to one another. Some examples of resources found in a device model (but not limited to) are tiles, wires, PIPs (programmable interconnect points, or switches, that can connect two wires), BELs (basic elements of logic, such as a LUT), sites and package pins. Device models in RapidWright can be loaded explicitly (by device or part name) or automatically, such as when loading a DCP. RapidWright supports all device models available in Vivado and the corresponding device `.dat` file is downloaded on-demand rather than installed upfront.

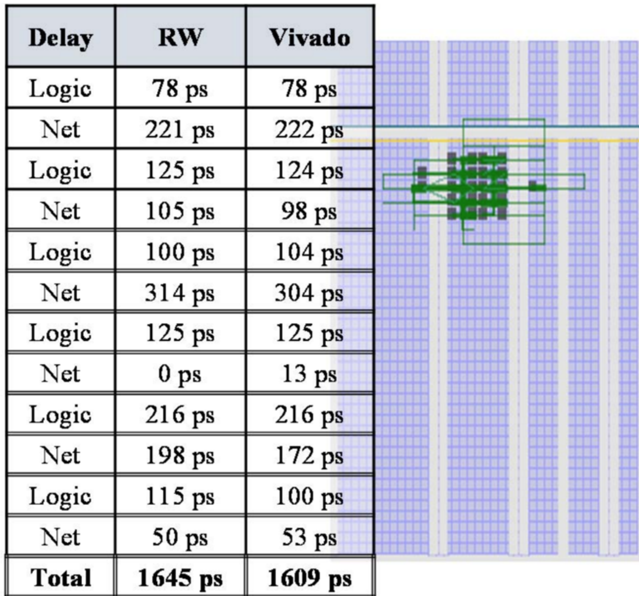
2) Logical Netlist Model: The logical netlist model refers to the synthesized netlist in which a user's design has already been mapped to UNISIM FPGA primitives. The primary file format used to represent a logical netlist is EDIF (Electronic Design Interchange Format) [6]. RapidWright is able to support hierarchical and both folded and unfolded representations of a logical netlist. Generally, the logical netlist inside a DCP is encrypted and so when designs are exported from Vivado to RapidWright, the user will also need to run the Tcl command `write_edif` to provide a readable netlist as RapidWright is unable to decrypt encrypted designs.

3) Physical Netlist Model: The physical netlist is a flattened version of the logical netlist that only describes how the primitives and connections are mapped onto the device model. The physical netlist consists of placement (how the leaf cell primitives of the logical netlist are mapped or "placed" onto device BELs) and routing (the set of configured interconnect switches on the device that complete net connections in the logical netlist).

B. An Open Timing Model

Access to detailed timing information for FPGA resources is essential to achieving the highest performance. Yet, for commercial FPGAs, much of this information is not published or available. At the same time, deploying large, fine-grained timing datasets adversely affects the speed of timing-driven place and route algorithms. To remedy these challenges, RapidWright includes an open timing model [7] that provides a high fidelity, approximate timing model for UltraScale+ devices. The timing model is lightweight as to limit its memory footprint and make it both nimble and fast to execute timing lookups. It was validated on over 240 designs and includes an intentional 2% pessimistic bias so that designs have a high degree of timing sign off when loaded into Vivado. The proposed model shows high fidelity to Vivado with a Spearman's ρ value of 0.99. Figure 2 shows an example of the timing model performance on a modest PicoBlaze design with a 2% pessimistic estimation.

The development and availability of the RapidWright open timing model was instrumental in the development of a timing-driven router called RWRoute. By making the model available to the broader community, we empower users to extend the capabilities, apply the same lightweight estimation techniques to other architectures and extend the work for other domains.



Delay	RW	Vivado
Logic	78 ps	78 ps
Net	221 ps	222 ps
Logic	125 ps	124 ps
Net	105 ps	98 ps
Logic	100 ps	104 ps
Net	314 ps	304 ps
Logic	125 ps	125 ps
Net	0 ps	13 ps
Logic	216 ps	216 ps
Net	198 ps	172 ps
Logic	115 ps	100 ps
Net	50 ps	53 ps
Total	**1645 ps**	**1609 ps**

Fig. 2. Critical Path Delay Timing Breakdown of a PicoBlaze Design Using the Open Timing Model in RapidWright vs. Vivado.

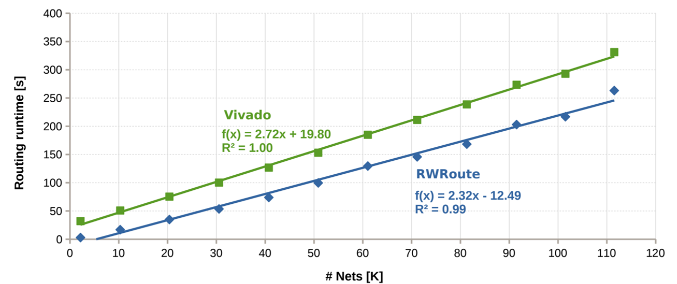


Fig. 3. Routing Runtime Scalability of RWRoute and Vivado.

C. RWRoute: A Fast, Timing-driven Router

One of the most time consuming steps of FPGA backend compilation is routing a placed design using an FPGA's programmable interconnect resources. Typical commercial solutions for FPGA routing have often been proprietary and are heavily optimized towards a common goal of increased quality of result (QoR). However, in the age of domain-specific computing, compile time can become an increasingly important factor. In order to address the runtime challenges of existing FPGA routing solutions, RWRoute [8] was developed using the RapidWright framework. RWRoute was able to leverage the open timing model to provide timing-driven capabilities as well as non-timing driven (wirelength) optimization goals. It also has been augmented to route clock signals, static signals (GND and VCC) as well as being able to perform partial routing or finalize a partially routed design.

RWRoute was able to achieve geomean speedups of 4.9× compile time speedup over Vivado for 200+ smaller designs (2-8 kLUTs) with only a 10% QoR loss. For larger designs, RWRoute is still able to maintain a runtime advantage over Vivado by at least 20% on designs with 110k nets as shown in Figure 3. RWRoute has proven to be an invaluable tool for RapidWright-powered projects due to its open source nature. It can be easily customized for domain-specific tasks and optimized for runtime or niche implementation flows as we will see in Section IV.

D. FPGA Interchange Format

Commercial FPGA development tools must focus on providing adequate support and performance across the full range of application-specific domains where FPGAs are used. Customized compilers, however, can often deliver superior performance for these application-specific domains by taking full advantage of both the application-specific attributes and constraints of the challenge, synergized with the FPGA's unique architectural capabilities.

Enabling such application-specific tools demands a common interchange standard to allow interoperation between an FPGA vendor's back-end tools and the customized compilers. We worked with Google [9] to develop such an interchange standard called the CHIPS Alliance FPGA Interchange Format [3]. This format defines standardized FPGA design and device files with sufficient fidelity and architectural robustness to be used by a wide range of design tools to perform tasks such as placement and routing in a fully open source environment.

The FPGA Interchange Format promotes the free exchange of solutions and customized strategies that can be leveraged by our customers through this interoperability bridge as shown in Figure 4. RapidWright therefore serves as an open-source gateway into Vivado, laying the groundwork for a growing ecosystem aimed at further advancing FPGA use through a variety of application-specific, front-end design tools.

III. Unique Capabilities and Example Use Cases

AMD's Vivado has a broad set of capabilities and is optimized for a specific set of design criteria that is applicable for the majority of customer scenarios. However, as we enter the domain-specific era of compute, there is a growing need to adapt tools and implementation strategies that take advantage of domain-specific attributes. RapidWright provides the extra level of flexibility needed to exploit these new opportunities by providing capabilities that were not previously feasible in Vivado alone. This section describes a few of the unique RapidWright capabilities available to users of the framework.

A. Optimize, Replicate and Relocate Placed and Routed Logic

Often times when trying to accelerate an application on an FPGA, a specific computation or routine is parallelized and reused many times. However, the conventional FPGA compilation flow may not always take full advantage of this optimization opportunity. One of RapidWright's key features is the ability to preserve, replicate and reuse placed and routed circuitry in the form of a pre-implemented module.

Suppose you wanted to build a programmable accelerator array with hundreds of instances of the core processing element. The performance (without complicated clocking solu-

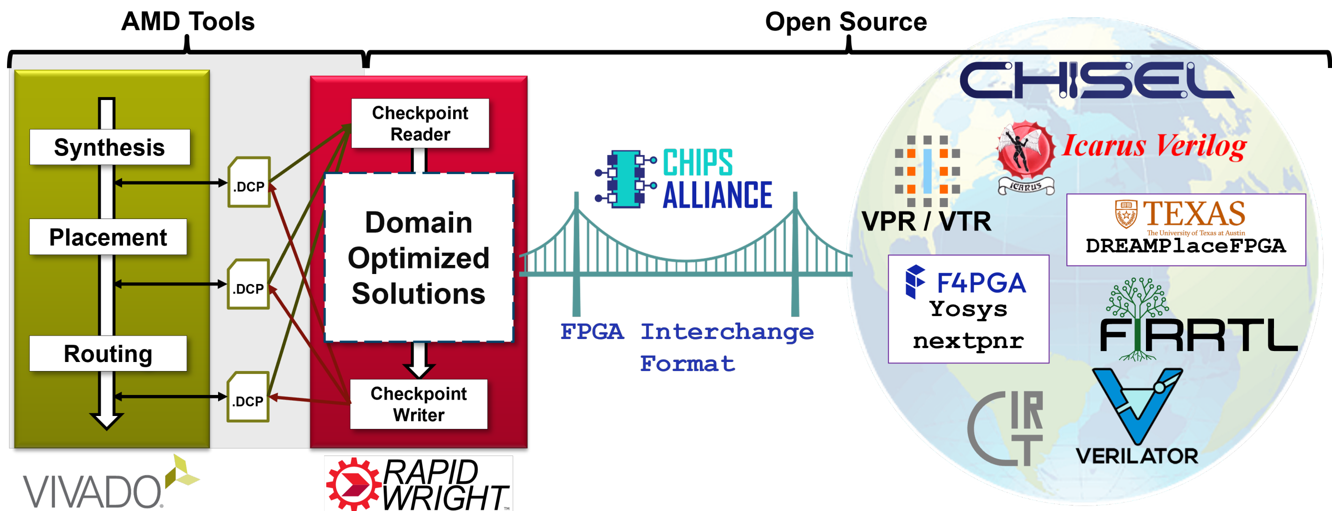


Fig. 4. FPGA Interchange Format Enabling Access to Broader Open Source Ecosystem.

tions) would be limited to the slowest implementation of each processing element. A conventional approach would simply instantiate the processing element in a `generate` statement in Verilog and let the tools synthesize, place, and route each processing element individually. This causes the place and route solution to be solved uniquely hundreds of times, generating a broad distribution of timing closure solutions for each instance.

With RapidWright, we can use a tool called `PerformanceExplorer` which will take a single synthesized instance of the processing element and place and route it hundreds of times in a variety of ways to find the best implementation that closes timing at the highest frequency that can also be replicated and relocated many times across the FPGA fabric. Although FPGA fabric has a high degree of regularity, there are some exceptional cases that make relocation of placed and routed logic more complex and RapidWright is built to accommodate these complexities.

Figure 5 shows a fully placed and routed accelerator array composed of 396 PicoBlaze 8-bit microcontrollers that have been optimized, replicated and relocated into reusable locations and stitched together on an UltraScale+ VU3P. This array was built using only three instances of a PicoBlaze (each one uses a BlockRAM and each column of instances is centered around that resource). The lowest timing closure of the three instances is 374 MHz and the final array is capable of running at 365 MHz without significant effort.

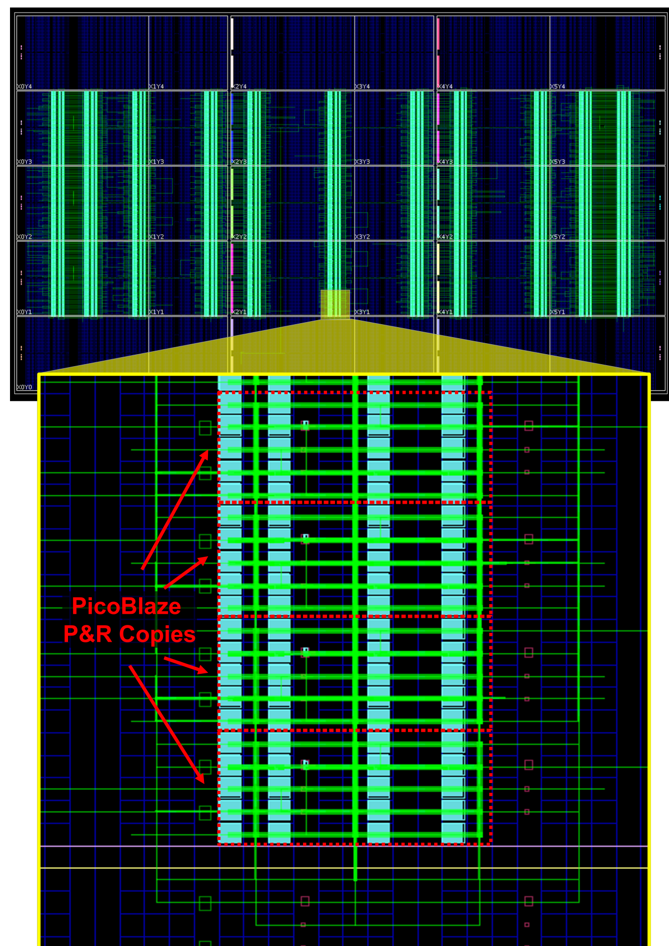


Fig. 5. Placed and Routed Accelerator Array of 396 PicoBlaze Elements Replicated and Relocated by RapidWright.

B. Generate Placed and Routed Logic On-the-fly

RapidWright's ability to create fully placed and routed circuits from scratch enables a new class of design we call generators. Several parameterizable circuit generators are included with the RapidWright distribution. One significant example is the parameterizable SLR crossing generator which can produce a placed and routed SLR crossing DCP solution within a few seconds. This SLR crossing generator targets UltraScale+ devices as they have the architectural capabilities that enable clocking techniques that achieve near-spec

(>700MHz) performance (UltraScale and Series 7 devices, do not possess these capabilities).

The generator will create pairs of flip flops in a netlist for each crossing signal and will place them at the appropriate Laguna sites to leverage the dedicated super long line (SLL) interconnect paths. As mentioned in [1], using both dedicated RX and TX Laguna site flops will often produce hold time violations. RapidWright is able to circumvent this issue by routing the clock in such a way that all TX and RX flops are connected exclusively to the same clock arm. This enables a tuning of the clock delay at the common leaf clock buffer for each group of crossing signals in each direction respectively.

Additionally, the SLR crossing generator can potentially create a custom clock root for each SLR crossing group (crossings in the same clock region) to minimize the inter-SLR compensation timing penalty. By fabricating the netlist, placing the flops onto the dedicated RX and TX Laguna sites and custom routing the clock to tune leaf clock buffers and create clock roots, the generator is able to create a placed and routed DCP of an SLR bridge in a few seconds.

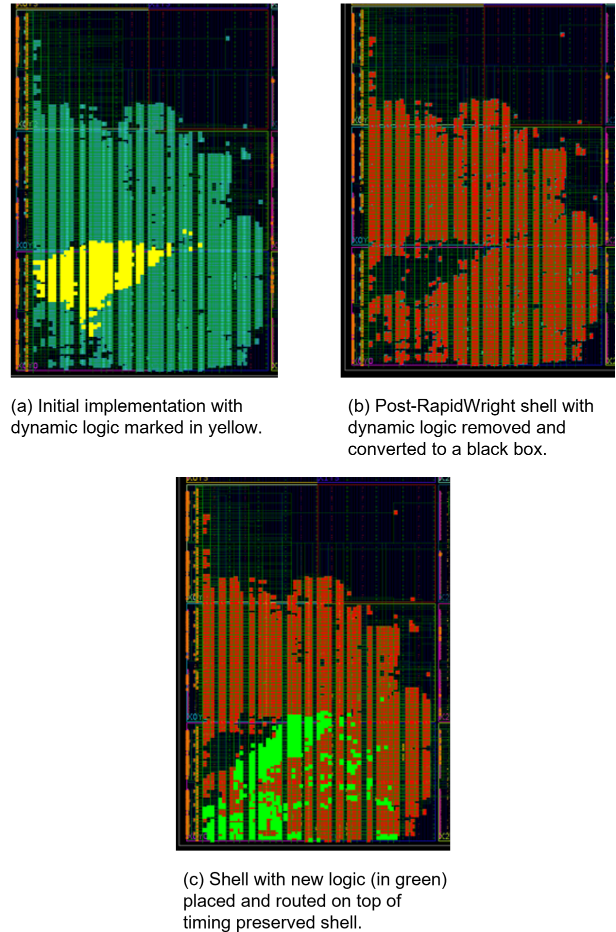
C. Turn Timing Closed Logic into a Reusable Shell

Often in FPGA development, a desirable timing-closed implementation is only achieved after several iterations or many parallel implementation runs of a design. Elusive timing closure can be caused by one or a few stubborn modules in a design that have tight constraints or a large number of moderately difficult paths that have a lower probability of timing closure on any given run.

One advantageous strategy to improve timing closure success can be to preserve and enable reuse of a known good implementation of the stubborn logic. By preserving the implementation, place and route tools can (hopefully) avoid rediscovering difficult timing closure and simply focus on the other logic.

Typically, the challenge in preserving and reusing timing closed logic is that it requires the use of area constraints (pblocks in Vivado) to spatially separate the logic to be preserved from the logic that will be changed over time. This presents another set of limitations and constraints of its own (such as the need for area constraints to be rectangular and exclusive) that are often undesirable. RapidWright is able to overcome this challenge by allowing the design to be placed and routed without area constraints and then carefully extract and remove the logic after it has been implemented. Vivado is currently not able to perform this extraction correctly and thus RapidWright is able to offer a solution that provides greater flexibility and the opportunity to preserve timing closed logic efficiently.

Figure 6 demonstrates how the dynamic logic (portion of the design to change over time) is removed to create a reusable shell that has its placement and routing locked down. The implementation tools are unable to make any changes to the static logic portion of the design and Figure 6c shows how new logic can be placed and routed on top of the existing timing-closed static logic of the shell without disturbing any of the implementation.



(a) Initial implementation with dynamic logic marked in yellow.

(b) Post-RapidWright shell with dynamic logic removed and converted to a black box.

(c) Shell with new logic (in green) placed and routed on top of timing preserved shell.

Fig. 6. Creation of a Timing-closed RISC-V Shell.

IV. Community Deployments

Several research efforts have made use of RapidWright to improve compile time and/or performance of the placed and routed implementation. This section provides a brief overview of some of those efforts and how the use of RapidWright was an instrumental part of their success.

A. RapidLayout: Fast Hard Block Placement Using an Evolutionary Algorithm

Zhang et al. [10] [11] found they could accelerate placement of hard block heavy designs (those using many DSPs and BRAMs) by employing an evolutionary algorithm to quickly select hard block placement. The designs being implemented were neural network accelerators with 480 replicated convolution blocks assembled in a systolic processing array. To implement the design, They were able to formulate hard block placement as a multi-objective optimization problem in the RapidWright framework and then hand off the rest of the placement problem to Vivado. Due to the replication found in the design, they were also able to take advantage

of RapidWright's ability to replicate and relocate logic by targeting a single SLR (super logic region, a single die of a multi-die FPGA device) and replicate it to other SLRs. This process was quite fast and led to speedups of 5-6× faster than a conventional Vivado flow that required weeks-long effort to create several placement constraints in order for the design to be fully implemented.

B. RapidRoute: Rapid Assembly of Communication Structures

Liu et al. [12] developed a customized router utilizing the RapidWright framework that was tuned for network-based communication structures (1D rings, torii, and meshes) within the FPGA fabric. Using a combination of optimization techniques, such as exploiting symmetry, multi-threading caching of routed structures and tiling hueristics, Liu et al. was able to route these network structures up to 8× faster than the conventional Vivado router and maintain a QoR within 0.2 ns of the Vivado result.

C. RapidStream: Faster Compilation and Higher Performance

Guo et al. [2] developed a novel, open source compilation flow called RapidStream that leveraged a class of latency-insensitive HLS designs to perform high-level partitioning and pipelining between partitions. By being able to decouple different parts of the HLS dataflow designs, RapidStream is able to perform placement and routing of each partition in parallel. RapidWright was able to quickly stitch all of the partitioned parts of the design into a single implementation as shown in Figure 7. By taking advantage of the latency insensitive nature of the HLS designs, RapidStream was able to compile implementations 5-7× faster and achieve 30% better QoR than the conventional Vivado flow. The benefits of RapidStream demonstrate that there is significant potential for FPGA backend improvement when the tools are enabled to take advantage of domain-specific attributes.

In follow on work from Guo et al. [13], RapidStream 2.0 was able to take advantage of RWRoute [8] by customizing its partial routing capability to be timing-driven and enable the tool to expand the routing bounding box at runtime. In routing the connections between partitions, the flip-flops (or anchor registers) in between, often would result in congestion that would cause costly rip-up and re-routing scenarios. The RapidStream 2.0 customized version of RWRoute was able to provide solutions with minimal rip-up of existing routes and could run up to 4× faster than a single-threaded Vivado run.

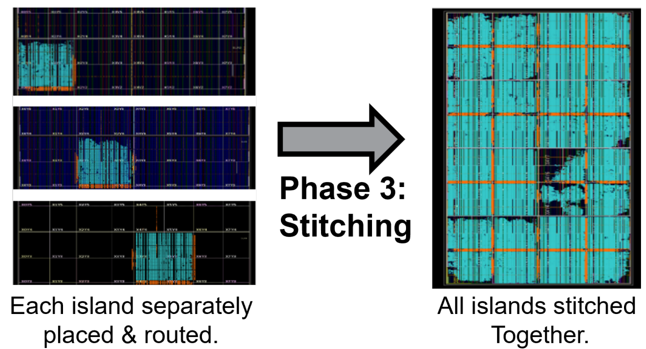


Fig. 7. RapidStream Stitching Process Accelerated by RapidWright.

V. Related Work

There have been a number of open interfaces to FPGA vendor tools in the past. The Xilinx Design Language (XDL) [14] provided an open design file format released with ISE (predecessor to Vivado). XDL provided unprecedented access to placement and routing information that made building custom CAD tools possible. Open source projects and tools such as RapidSmith [15], Torc [16], and ReCoBus-Builder [17] capitalized on the XDL interface to increase productivity. Altera (now part of Intel) also released the Quartus University Interface Program (QUIP) [18] with similar attributes to XDL.

XDL and QUIP are no longer supported by the latest vendor tool suites Vivado and Quartus Prime. Instead, Xilinx's Vivado utilizes a new Tcl-based interpreter. This allows users to perform customized tasks by writing Tcl scripts. For small tasks, this is not a problem. However, as task size and complexity grows, productivity is limited by Tcl interpreter performance. For example, Tincr [19] and the Vivado Design Interface (VDI) [20] have extended RapidSmith to be compatible with Vivado by replacing XDL with Tcl interface routines. However, design import by this method is constrained by the low speed of Tcl commands, limiting feasibility to only the smallest designs. Townsend [20] reports a design of ˜54K LUTs targeting an xc7a100t (Artix 7) takes ˜2.5 hours to import into Vivado (6 LUTs/second). By rough comparison, RapidWright writes a 210K LUT DCP targeting an xcvu190 (Virtex UltraScale) in 91 seconds. Vivado reads the DCP in 303 seconds, for a total import time of 394 seconds, or 533 LUTs/second, 88× faster than VDI. By avoiding Tcl and using DCPs, RapidWright enables a more productive interface.

VI. Conclusion

We believe the field of FPGA backend tooling is ripe with optimization opportunities and we have created RapidWright to help take advantage them. This paper has described notable works such as RapidLayout and RapidStream that domain-specific optimization for backend tooling can lead to faster compile times and/or higher QoR implementations.

There exists a growing and innovative community for FPGA backend tooling that will co-exist with the RapidWright framework. To help foster advances and interoperability with these tools, RapidWright fully supports the CHIPS Alliance FPGA Interchange Format [3] as a standardized way to exchange intermediate FPGA implementation results and build tools amongst both open source and industry participants alike. We invite both industry and academic researchers to help us move forward the domain-specific efforts of FPGA backend technology.

For more examples, documentation, tutorials and resources on RapidWright, please visit www.rapidwright.io.

References

[1] C. Lavin and A. Kaviani, “Rapidwright: Enabling custom crafted implementations for fpgas,” in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 133–140.

[2] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, “Rapidstream: Parallel physical implementation of fpga hls designs,” in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361

[3] [Online]. Available: https://www.chipsalliance.org/projects/

[4] C. Lavin and A. Kaviani, “Build your own domain-specific solutions with rapidwright: Invited tutorial,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–22. [Online]. Available: https://doi.org/10.1145/3289602.3293928

[5] [Online]. Available: http://www.rapidwright.io/docs/index.html

[6] R. LaFontaine et al., *Electronic design interchange format: EDIF. Reference manual*. Electronic Industries Association, EDIF Steering Committee, 1993, no. v. 2.

[7] P. Maidee, C. Neely, A. Kaviani, and C. Lavin, “An open-source lightweight timing model for rapidwright,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 171–178.

[8] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt, “Rwroute: An open-source timing-driven router for commercial fpgas,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–27, 2021.

[9] [Online]. Available: https://opensource.googleblog.com/2022/02/FPGA%20Interchange%20format%20to%20enable%20interoperable%20FPGA%20tooling.html

[10] N. Zhang, X. Chen, and N. Kapre, “Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithms,” in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 145–152.

[11] ——, “Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithm,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, jun 2022. [Online]. Available: https://doi.org/10.1145/3501803

[12] L. Liu, J. Weng, and N. Kapre, “Rapidroute: Fast assembly of communication structures for fpga overlays,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 61–64.

[13] L. Guo, P. Maidee, Y. Zhou, C. Lavin, E. Hung, W. Li, J. Lau, W. Qiao, Y. Chi, L. Song, Y. Xiao, A. Kaviani, Z. Zhang, and J. Cong, “Rapidstream 2.0: Automated parallel implementation of latency insensitive fpga designs through partial reconfiguration,” *ACM Trans. Reconfigurable Technol. Syst.*, apr 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3593025

[14] *Xilinx Design Language Version 1.6*, Xilinx, Inc., July 2000, Xilinx ISE 6.1i Documentation in ise6.1i/help/data/xdl.

[15] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, “RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs,” in *2011 21st International Conference on Field Programmable Logic and Applications*, Sept 2011, pp. 349–355.

[16] 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)*, February 2011, pp. 41–44.

[17] D. Koch, C. Beckhoff, and J. Teich, “ReCoBus-Builder – A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs,” in *2008 International Conference on Field Programmable Logic and Applications*, Sept 2008, pp. 119–124.

[18] S. Malhotra, T. P. Borer, D. P. Singh, and S. D. Brown, “The Quartus University Interface Program: Enabling Advanced FPGA Research,” in *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology*, Dec 2004, pp. 225–230.

[19] B. White and B. Nelson, “Tincr – A Custom CAD Tool Framework for Vivado,” in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6.

[20] T. Townsend, “Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices,” Master’s thesis, Brigham Young University, July 2017.