Timing-aware routing in the RapidWright framework

Leo Liu
School of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
l276liu@uwaterloo.ca

Nachiket Kapre
School of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
nachiket@uwaterloo.ca

***Abstract*—We can extract approximate, fine-grained timing information of routing resources of Xilinx FPGAs using the RapidWright open-source framework. The absence of timing information makes it difficult to implement timing-aware FPGA CAD tools using RapidWright. It is impractical to invoke Vivado's timing analysis engine for each choice within an optimization loop of your custom CAD algorithm as that would slow down execution by orders of magnitude. We route a set of one-time calibration tests on the FPGA using Vivado to extract path delays, and setup a system of linear equations based on the unknown delays associated with each routing resource used in the calibration route. We run this calibration for an interconnect tile but generalize the result to the entire FPGA due to device symmetry. We then solve these equations using least squares approximation as the resulting system is low-rank. This is due to the routing restrictions imposed by the FPGA fabric for legality of the connection and correctness of Vivado's timing analysis. We are able to learn an approximate timing model for RapidWright that is within 1% error (0.01 ns) of Vivado timing analysis by running ≈30 calibration runs and needing under 60 seconds of Vivado timing analysis. We demonstrate this technique on Xilinx XCKU115 FPGA (-3, -2, and -1 speed grades). The open-source RapidRoute custom router previously lost to Vivado by as much as 0.3–0.4 ns on timing slack when using a crude timing model. With our timing model enhancements, we allow RapidRoute to close the slack gap with Vivado and even outperform Vivado marginally on occasion. Our timing model generation is lightweight and can be discovered for each FPGA device instead of bundling memory-hungry timing libraries with RapidWright.**

I. Introduction

FPGA development has long been plagued by slow CAD runtimes and tedious low-level Register-Transfer Level (RTL) design. This development barrier has hurt FPGA adoption and prevented high-level software programmers from considering these devices for broader use. FPGAs provide a compelling combination of performance and power efficiency for those trained to exploit this potential. High-Level synthesis [2] tools have addressed the development challenge somewhat by making it possible to program in higher levels of abstraction in C/C++ programming languages. However, these languages are not the right abstractions for communicating parallel intent and do nothing to address the long development cycles associated with FPGA CAD.

The RapidWright framework [9] is an important step that can tackle both challenges in one go. First, it provides a Java-based high-level programming environment for accessing and traversing low-level FPGA hardware resources. Second, it makes it possible to generate datapaths directly by composing these hardware resources in a generative manner without the need for time consuming FPGA CAD. The programmer is now in control of design description, as well as placement and routing of the design. For repetitive designs with reusable layouts, or custom FPGA CAD tools, RapidWright provides a scalable design flow that is flexible, avoids the need for RTL design capture, while also eliminating the wait times of the slow FPGA CAD tools.

While this is promising, RapidWright has one key weakness – *the absence of timing knowledge of FPGA resources*. When building layouts, RapidWright provides a mechanism for navigating the connectivity structure of a modern FPGA device but does not carry any information, fine-grained or otherwise, of individual resource delays. One has to invoke the timing analysis engines in Vivado for a given netlist to evaluate timing properties of the design. This still does not provide the capacity of break down individual resource delays. The absence of timing information creates two main challenges:

- It makes it intractable to create timing-aware CAD tools. In such tools, we must use timing information to make informed implementation choices. Performing Vivado timing analysis in the inner loop of such tools is infeasible and will results in orders of magnitude slowdown in runtime, defeating the purpose of the lightweight RapidWright framework.
- If we ignore timing information entirely, delay minimization must use assumptions that may contradict resource delays. A short path that minimizes geometric hop count need not be the actual shortest path. The use of crude delay estimates in RapidRoute [11], a custom router built with RapidWright, was shown to result in 0.3–0.4 ns error in predicting the timing delays of paths.

Adding a timing database to RapidWright in a manner similar to Vivado would be exorbitantly expensive in terms of memory footprint. Each FPGA family, size, and speed grade requires a unique timing model that would be challenging to ship with RapidWright from a legal as well as technical perspective. Can we identify timing properties of resources without bloating RapidWright? In this paper, *we show how to cheaply extract timing information of FPGA interconnect*

resources of each device and enrich RapidWright with a timing model for routing. This is achieved through a one-time device characterization phase where a set of valid calibration routes are implemented on that FPGA device. These routes are exported to DCP (Design Checkpoint) for timing analysis via Vivado. We then build a system of linear equations based on the number of interconnect resources of each type organized within the FPGA fabric. The number of equations will be the number of calibration runs performed. The RHS (right hand side) of this set of equations is the delay vector built on the results of Vivado timing analysis. As the routing fabric imposes restrictions on connectivity the resulting system of equations has low rank. We use a Least Squares Approximation technique to solve this system of equations and identify path delays to within a 1% margin of error.

The key contributions of this work include:

- The design of an offline characterization tool to generate calibration paths and to extract timing delays of FPGA interconnect resources using RapidWright. Identification of the set of calibration paths necessary to extract sufficient timing visibility into the device. Offline characterization of the UltraScale+ KCU115 devices takes ≈60 s for 30 runs.
- The evaluation of accuracy of timing analysis of point-to-point interconnect paths using our enhancements to RapidWright and comparison against timing analysis engine in Vivado. Our models are able to predict path delays that are within 1% of Vivado's prediction and less than 0.01 ns error. We also explore accuracy-model fidelity tradeoffs to potentially reduce calibration cost.
- Adaptation of the open-source RapidRoute custom router built using RapidWright to take advantage of this approximate timing knowledge. For bussed communication networks like rings, torii, and meshes, the timing-enhanced router is indistinguishable from Vivado and can even outperform it marginally on occasion.

II. Background

A. RapidWright

As discussed earlier, RapidWright presents a different approach for implementing computations on FPGAs by exposing the low-level details of the FPGA architecture fabric directly to the developer. This may seem counter-intuitive to the goal of raising the abstraction of FPGA programming, but it targets custom "overlay" developers and custom "CAD tool" builders rather than software programmers directly. It follows the footsteps of and inherits the legacy of similar tools in the past like Xilinx XDL [1], ReCoBus-Builder [8], RapidSmith [10].

Overlays provide a coarser-grained view of hardware resources as a collection of custom ALUs, Memory (SRAM) blocks, and can be programmed with a custom set of instructions. These instructions are higher-level, word-oriented operations than low-level bit-oriented configuration of LUT contents and routing paths. FPGA experts can develop these overlays with software programmers in mind who understand traditional CPU blocks like ALUs, Memories, and Instruction Sets (ISAs). Overlays can be customized per application by expert FPGA programmers and programmed by software developers using custom Overlay-specific ISAs. The overlays themselves can be fully optimized to exploit the features of the underlying FPGA fabric in a manner automated tools, or naive software programmers attempting RTL design cannot hope to match. The overlays can then be "relocated" and "tiled" across the FPGA fabric to exploit regularity and provide an easy pathway to scaling performance at the cost of resources. RapidWright has been used to demonstrate the generation of arrays of Picoblaze processors with Linkblaze [12] interconnect.

Custom CAD tools serve to overcome the limitations of conventional FPGA CAD tools like Xilinx Vivado. Vivado is a monolithic, industry-class, software program with broad set of features to support a wide variety of customer RTL design requirements. This makes it bloated, sluggish, memory-intensive, and sometimes with bugs that make it difficult to exploit specific features advertised in datasheets. Custom CAD tools can be developed if a convenient representation of the underlying physical FPGA fabric is available. RapidWright-based CAD tools such as SAT Routing for inter-SLR (Super Logic Region) crossing [4], RapidRoute [11] show the benefit of providing device resource visibility for the development of routers. Post-implementation debug insertion [7] is another popular target for such custom tooling.

B. Routing

For arbitrary circuits implemented on FPGAs, routing algorithms such as Pathfinder [13] are vital to deliver a timing- and congestion-aware mapping solution. Pathfinder operates in an iterative manner by allowing routes to compete for resources and incrementally develop a picture of congestion in the FPGA interconnect network. This allows the routes to negotiate their way to the resources they need to meet timing and area constraints of the developer. For inter-SLR routing [4], or for regular layouts [11], we have additional design constraints that could be exploited effectively to create custom routing solutions. This is possible as placement of the communication objects have specific constraints depending on the application. For instance inter-SLR crossings impose placement requirements on the usage of Laguna tiles. For overlay designs, there is an abundance of symmetry in the physical layout of the design. For these cases, the inbuilt Vivado router can be too slow to deliver the required solution which is a must for fast overlay generation, or even fail to meet timing requirements due to the odd nature of inter-SLR communication. However, RapidWright currently does not expose an interface to extract fine-grained resource delays that would be necessary for timing-aware routing. While tempting, it would be tricky to add Vivado's timing databases directly into RapidWright. These databases hold detailed timing information for each device, speed grade, family combinations that would quickly bloat the RapidWright distribution.

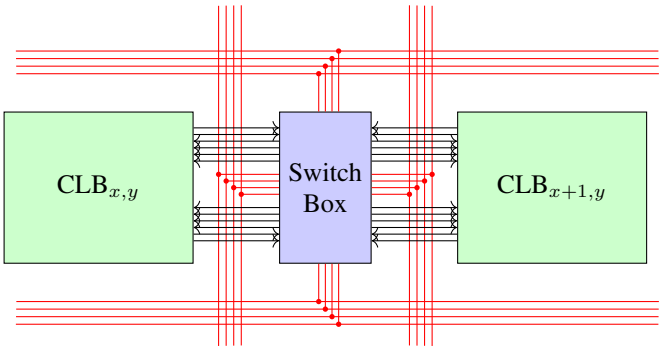

Fig. 1: Xilinx CLB and Switchbox architecture exposed by RapidWright. Red wires are horizontal and vertical global interconnect. Black wires allow CLB IOs to enter the global interconnect. PIPs inside the Switch Box allow a router to establish a path between source-destination pairs.

C. Architecture View for FPGA Interconnect

Bulk of the delays in modern FPGA design arises from interconnect. When looking at interconnect structures, modern FPGAs are typically organized in a hierarchical manner. A cluster of LUTs has rich intra-cluster connectivity using *local interconnect* and data movement between clusters is supported with an inter-cluster network using *global interconnect*. We show a closeup of one Switchbox and two CLBs in Figure 1. In Xilinx Architecture terminology the FPGA is organized as a collection of BELs (Basic Element of Logic) with input and output pins. These may be of two types: Logic BEL and Routing BEL. In the simplest case Logic BELs are LUTs, and Routing BELs are programmable multiplexers in the interconnect. Within a Routing BEL, there are programmable connections between inputs and outputs within a BEL called PIPs (Programmable Interconnect Points). BELs are connected to each other using Wires. From Figure 1, we observe that Xilinx bundles the traditional *cbox* and *sbox* switches from island-style FPGA [14] into a single Switchbox block. When modeling FPGA interconnect delays using the Elmore approximation [15], [3], we need to know the individual resources delays of various buffered wire segments, and programmable switching points (PIPs). These quantities are then summed together to extract overall path delay between a source and destination pair. These are typically LUT or FF outputs (source) and LUT or FF inputs (destination). However, this fine-grained breakdown of delays is not available in RapidWright.

III. Timing Characterization with RapidWright

In this section, we will inspect the cost of ignoring timing information during routing, the process to acquire timing information using RapidWright, and the use of this information during routing to determine the fastest path between two points on an FPGA.

A. Cost of ignorance

RapidWright ships with an in-built router for constructing overlays. It is not an industry-strength router and it does not produce high quality results. RapidRoute [11] is a custom open-source router built on top of RapidWright for connecting bussed networks on FPGAs. It is no surprise that it also lacks a sophisticated timing model to drive the routing algorithm but offers an improved QoR (quality of result) and significantly faster runtime. In both routers, this timing obliviousness forces the tools to aim for minimization of *hops* along the route. As each *hop* is not made equal, the resulting route is not the true slowest route on the FPGA. For example, let us route a 1-bit net from X48Y20 to X48Y23 (2 tiles away) on a Xilinx UltraScale+ XCKU115 -3 speed grade part. When Vivado routes this net, the path takes 0.795 ns of delay but RapidRoute takes 0.958 ns. Both paths take the same number of interconnect hops as shown in Table I, but end up selecting slightly different resources. If you look closely, even the resource types are mostly identical.

TABLE I: Comparing routing resources selected for timing path between Vivado and RapidRoute. Bulk of the path is the same, with a few strategic differences that lead to an inferior slower solution than Vivado router. Both paths have identical number of resources used, so minimizing hop-count is insufficient.

Vivado Path	**RapidRoute Path**
LOGIC_OUTS_E19	LOGIC_OUTS_E19
SINGLE_DOUBLE_15	SINGLE_DOUBLE_15
NN1_E_BEG4	NN1_E_BEG4
SINGLE_DOUBLE_14	SINGLE_DOUBLE_14
NN1_E_BEG4	NN1_E_BEG4
SINGLE_DOUBLE_14	SINGLE_DOUBLE_14
NN1_E_BEG4	NN1_E_BEG4
IMUX_7_INT_OUT	IMUX_7_INT_OUT
BOUNCE_E_11_FTS	**BYPASS_E9**
IMUX_0_INT_OUT	**IMUX_22_INT_OUT**
BYPASS_E2	BYPASS_E2
IMUX_13_INT_OUT	IMUX_13_INT_OUT
BYPASS_E9	**BOUNCE_E_11_FTS**
INODE_1_E_1_FTS	**IMUX_0_INT_OUT**
BYPASS_E0	BYPASS_E0
IMUX_12_INT_OUT	IMUX_12_INT_OUT
BYPASS_E13	BYPASS_E13
IMUX_31_INT_OUT	IMUX_31_INT_OUT
BYPASS_E4	BYPASS_E4
INODE_1_E_28_FTN	INODE_1_E_28_FTN
BYPASS_E15	BYPASS_E15
0.795 ns	0.958 ns

B. Building a Timing Model

It is possible to develop an approximate timing model by routing a set of calibration paths on the FPGA, recording their delay as reported by Vivado's timing analysis engine, and then solving a system of equations constructed from this experiment. RapidRoute [11] does a simplistic version of this calibration by restricting the unknowns to wire delays. This helps it outperform RapidWright's internal router, but it still loses to Vivado by a non-trivial 0.3–0.4 ns. Each path that is routed on the device will take a specific sequence of interconnect resources. Our goal is to identify delays of each

resource, so we can predict delays of paths that use these resources in different combinations than our calibration set.

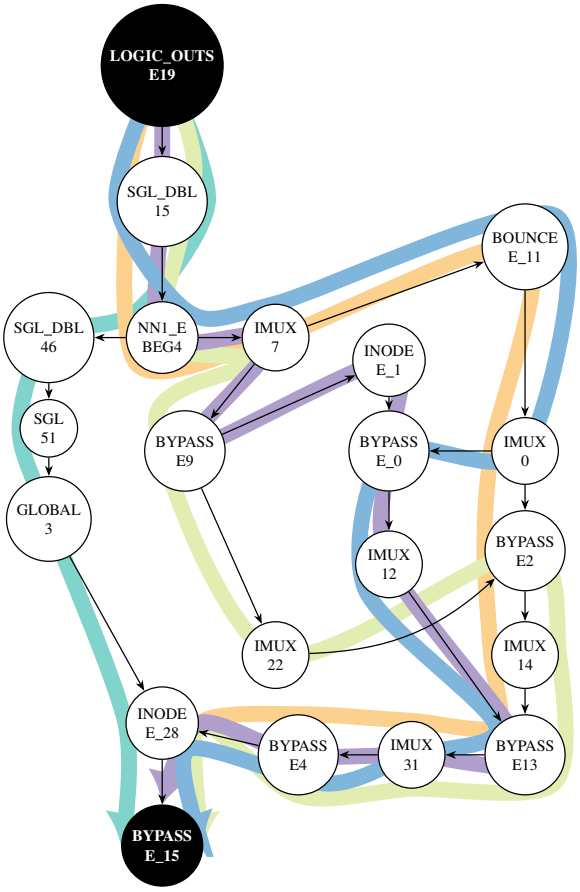

Fig. 2: Routing resource graph representation of small portion of Xilinx interconnect. Investigating paths between *LOGIC_OUTS_E19* to *BYPASS_E15*. You will notice that most of the interconnect resources that show up in Table I are visible on this resource graph.

For example, in Figure 2, we show a physical graph representation of the interconnect resources for a small portion of the switchbox. For our example, we route a set of five connections between *LOGIC_OUTS_E19* (source) and *BYPASS_E15* (sink). These paths take different routes through the fabric and allow us to setup a system of linear equations as shown in Figure 3. Each resource delay is considered as an unknown and we mark the number of times a resource is used in the matrix location corresponding to that resource. For this example, we have 21 unknowns and 5 equations so the system is low rank. We use Least Squares Approximation to solve for the unknowns. Typically, if a chunk of resources are always used in conjunction, their delay sums are effectively a bundled unknown for our linear system. In general, when we apply this approach across the larger FPGA switchbox architecture, we still end up with low rank matrices due to the connectivity restrictions on what constitutes a legal route. This means that our delays obtained by solving the linear system will be an approximation with some error rather than exact delay. We see later in Section IV we recover sufficient information to be able to close the quality gap with Vivado's timing-aware routing.

$$\begin{bmatrix} 1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 1&1&1&0&0&0&1&1&1&1&1&0&0&0&0&1&1&1&0&1&1 \\ 1&1&1&0&0&0&1&1&1&1&0&0&0&1&1&1&1&0&0&1&1 \\ 1&1&1&0&0&0&1&1&0&0&0&1&0&1&1&1&1&0&1&1&1 \\ 1&1&1&0&0&0&1&1&0&0&1&0&1&0&0&1&1&1&1&1&1 \end{bmatrix} \cdot \begin{bmatrix} t_0 \\ t_2 \\ \vdots \\ t_{19} \\ t_{20} \end{bmatrix} = \begin{bmatrix} 0.324 \\ 0.547 \\ 0.567 \\ 0.559 \\ 0.533 \end{bmatrix}$$

Fig. 3: Linear system constructed for the five paths routed in Figure 2. The RHS of the system is delay reported by Vivado timing analysis. Paths are generated as DCPs via RapidWright.

C. Linear System Setup

Xilinx switchboxes consist of many programmable PIP resources, and it is not always possible to construct combinations that isolate each of them individually. Hence, we construct a set of calibration routes that cover many combinations of resources. To reduce the number of unknown delays and keep the size of the calibration set reasonable, we apply some simplifying approximations. We study the Xilinx interconnect fabric to identify a set of resource "types". These types are assumed to have a common delay. A Xilinx switchbox is a collection of many such resource "types" with several instances of each type connected in a certain manner. For our application, we identify the following "types": *BOUNCE*, *BYPASS*, *INODE*, and *IMUX*. We also identify wire segment types: *NN1*, *NN2*, *NN4*, *NN12* for North-bound wiring. We also wiring in the other three directions South, West, and East. In addition, each resource has XY label, {E,W,S,N} direction classification and an identifier number for each instance of that resource within an INT tile. Finally, due to FPGA symmetry, in most instances, we are able to reuse the delay information generated by solving for one interconnect tile in other locations on the chip. Certain irregularities like IO columns do interfere with this simplification, but that is very rare.

IV. Evaluation

We use Vivado 2018.2 for our timing analysis and routing experiments. We use RapidWright v2018.2.5 to generate DCPs for calibration experiments. For our calibration DCPs, we ensure that Vivado accepts our DCP as a legal design without any antennas or disconnected segments; we discovered this painfully as Vivado timing analysis would deliver unpredictable results for such illegal DCPs. We solve our linear system using Least Squares Approximation technique with the Python *numpy* package to obtain our timing model after calibration. To evaluate the effectiveness of our calibration, we partition the calibration data into training and testing sets with 30% of our runs set aside for testing. This allows us to check against overfitting. We repeat our calibration experiments on the XCKU115 FPGA with speed grade -3, -2, and -1. We integrate our timing model with RapidRoute [11] and evaluate the effectiveness on routing structures such as rings, torii, and meshes than span the chip.

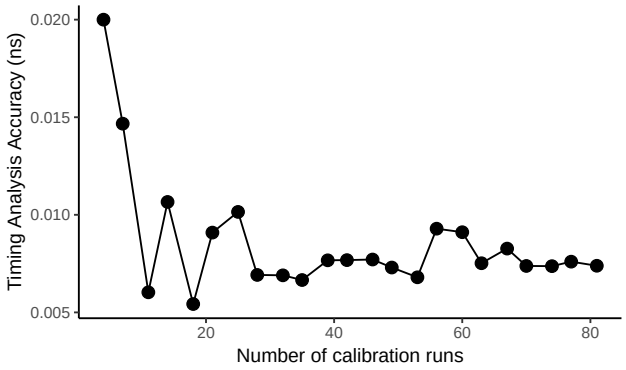

Fig. 4: Improving accuracy of timing analysis through offline one-time calibration. Timing error is <0.01 ns after 30 calibration runs. Some error remains even at 100 calibration runs to low rank system of equations and use of least squares approximation.

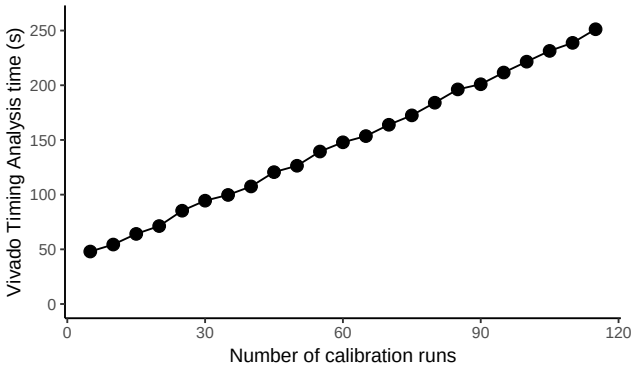

Fig. 5: Timing model construction time (DCP generation with RapidWright + Vivado timing analysis). Vivado timing analysis for the entire set of calibration DCPs loaded one after another takes <250s of runtime on Intel Xeon E5-1630.

A. Calibration

To build the timing model we must first run an offline one-time calibration round per FPGA device. A calibration round consists of a set of timing experiments. Each timing experiment involves routing a point-to-point net (FF→FF) that traverses the global interconnect in a manner that aims to cover a diverse set of PIP types. Due to the nature of FPGA connectivity, it is highly unlikely that we can guarantee full coverage of all PIP combinations. But we observe that we are able to characterize most PIP types in the device in under ≈30 calibration runs. We generate a set of DCPs for a set of nets and invoke Vivado timing analysis on each DCP (net). This data is fed into our model building least square approximation routine to compute timing delays of PIPs. Armed with this timing delay, we then compare the timing predicted by our model with that of Vivado's timing analysis engine. In Figure 4, we show the effect of increasing our calibration set size on prediction accuracy of our timing model. After around 30 runs, the calibration error drops to 0.01 ns (<1%). While not identical to Vivado due to limited coverage of the routing network, this is adequate for implementing a timing-aware router as we will demonstrate in Section IV-C. In Figure 5, we report the total Vivado timing analysis time required to evaluate timing of the point-to-point nets in our calibration set. The linear least squares approximation on this data takes negligible time and runtime is dominated by invoking Vivado timing analysis engine. For 30 calibration runs, we need around 100 s of sequential runtime on an Intel Xeon CPU E5-1630. This calibration is a one-time task for a particular FPGA device and once the model is built it can be reused freely. The model is lightweight in memory footprint as well associating a simple floating-point delay quantity with each of 50 or so wire and PIP parameters. Due to symmetry we can reuse these values almost all across the chip (except some cases with interference from IO columns).

B. Wire and PIP Delays

The primary objective of our work is to extract the fine-grained timing information of interconnect resources such as wires and PIPs. After calibration runs, we run least squares approximation on the data to solve for the unknown resource delays. We show the trends for these resources when considering KCU115 FPGA with speed grades of -1, -2, and -3. We see several interesting trends:

- Speed grade -3 is faster than -2 which in turn is faster than -1 grade device as expected.
- When inspecting wire delay trends (NN, SS, EE, or WW resources) in Figure 6 for xcku115, we observe the longer resources take longer delay as they travel further along the die. For instance, for speed grade -1 of xcku115, the one-hop *NN1* wire takes 18.4 ps per hop while a four-hop *NN4* wire needs 59.1 ps per hop. Thus, a router can choose *NN4* instead of taking 4×18.4 ps=73.6 ps route. The 12-hop wire *NN12* is surprisingly faster at 46.9 ps than even an *NN4*.
- For xcku115, the most expensive individual PIP delays belong to *LOGIC_OUTS* (94.4 ps), *BOUNCE* (79.9 ps), *INODE* (79.6 ns). They happen to be larger than wire delays and only approaching delays of the 5-hop wires.

C. Integrating with Custom Router

Finally, we investigate the effect of integrating this approximate timing model to perform customized routing in RapidRoute [11]. RapidRoute is an open-source custom router that generates communication structures for overlays from the ground up for bussed networks like rings, torii, and meshes. It exploits symmetry in the connectivity requirements to generate routed DCPs in 8× less CPU time and 1000× less memory footprint compared to Vivado. However, it operates on a crude timing model and loses to Vivado in terms of timing slack by as much as 0.3–0.4 ns. We run Vivado with *MoreGlobalIterations* mode to optimize circuit timing and provide a tough baseline to beat.

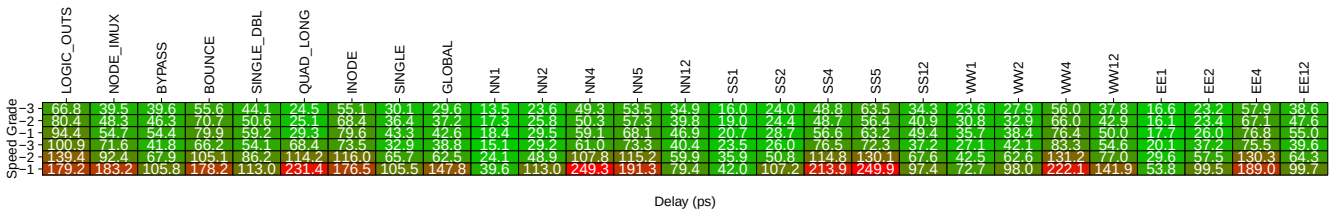

Fig. 6: Approximate timing delays of fine-grained interconnect resources including wires and PIPs solved through least squares method for xcku115 (top 3 rows) and xcku5p (bottom 3 rows). Nodes are Color coded to show slow resources with red fill and fast resources with green fill.

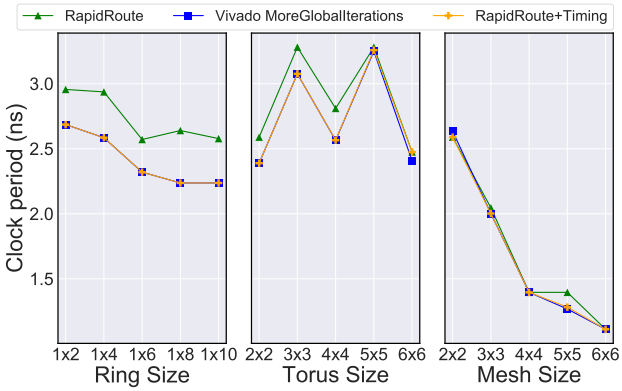

Fig. 7: Effect of enhancing RapidRoute with an approximate timing model for wire and PIP delays. Clock period is now practically indistinguishable from Vivado's *MoreGlobalIterations* high-performance mode. Timing-oblivious RapidRoute loses to Vivado by as much as 0.3–0.4 ns.

As we see in Figure 7, RapidRoute enhanced with our approximate timing model is practically indistinguishable from Vivado router when considering the critical path delay. In one case we actually discover a *faster* route than Vivado and another case we are slower than Vivado by 0.05 ns. Even with the approximate nature of our timing model, the tool has recovered enough detail to drive a router. The router can mimic Vivado's QoR (Quality of Result) for all practical purposes when routing bussed communication structures on the FPGA fabric. We do see a couple of minor outliers where we lose or win by 0.05 ns but we attribute that to tool noise. Importantly, after integrating a timing model, the runtime advantages of RapidRoute are still retained and the router runs $\approx 8\times$ faster than Vivado as before.

V. Related Work

While our work aims to integrate timing knowledge into RapidWright, there is some prior art for doing a similar task in the field of variation-aware FPGA CAD on a component-specific basis.

In GROK-INT [5] and GROK-LAB [6], the authors aim to extract delays of interconnect and logic blocks inside each FPGA chip. The goal is to quantify the impact of device variation on delay. With a detailed delay map per device, a variation-aware CAD tool can take advantage of timing knowledge to generate a device-specific bitstream. We solve a different problem – extracting timing information from Vivado through RapidWright. Our work is done purely in software through a small set of calibration experiments with Vivado. We are also able to extract precise information of the interconnect resources and use that to solve a system of equations to get fine-grained timing information. The GROK-INT and GROK-LAB work have to abstract a series of connected resources into DUKs (Discrete Units of Knowledge). This limits the granularity of timing information but is adequate for device specific CAD. We are not constrained in that manner as RapidWright makes precise path breakdown available to the user. There are some limits on how interconnect resources can be chained to create legal timing paths, but this has been sufficient for our work.

VI. Conclusions

In this paper we show how to build an approximate timing model for interconnect resources within the Xilinx RapidWright framework. We do this by running Vivado timing analysis on a small set of calibration routes that aim to cover a diverse set of interconnect resources within an interconnect tile to connect nets. We then setup a linear system of equations that is low rank and is solved using least squares approximation. This is necessary as the routing rules make it challenging to generate enough path diversity for a full rank system of equations. We achieve $<1\%$ error (0.01 ns) in our timing predictions for FPGA nets after 30 calibration runs and under 60 seconds of Vivado timing analysis. Our timing model for the interconnect tile can be generalized across the FPGA due to device symmetry. Our approach also extends to FPGAs with different speed grades as well as device families. With our approximate timing model, we are able to enhance the open-source RapidRoute tool to close the timing gap with Vivado almost entirely and even surpass Vivado's slack in certain cases. With our tool, you can add timing awareness to RapidWright without the need to bloat it with memory-hungry detailed timing databases.

Acknowledgements: We would like to thank Dr. Chris Lavin and Dr. Alireza Kaviani at Xilinx Research for assisting us in the development of this project.

Source Code:
https://git.uwaterloo.ca/watcag-public/RapidRoute-TimExt
https://git.uwaterloo.ca/watcag-public/RapidRoute

REFERENCES

[1] C. Beckhoff, D. Koch, and J. Torresen. The Xilinx Design Language (XDL): Tutorial and use cases. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, June 2011.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[3] W. C. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics*, 19(1):55–63, 1948.

[4] H. Fraisse and D. Gaitonde. A SAT-based Timing Driven Place and Route Flow for Critical Soft IP. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 8–87, Aug 2018.

[5] B. Gojman and A. DeHon. GROK-INT: Generating Real On-Chip Knowledge for Interconnect Delays Using Timing Extraction. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 88–95, May 2014.

[6] Benjamin Gojman, Sirisha Nalmela, Nikil Mehta, Nicholas Howarth, and André Dehon. GROK-LAB: Generating Real On-chip Knowledge for Intra-cluster Delays Using Timing Extraction. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):32:1–32:23, December 2014.

[7] B. L. Hutchings and J. Keeley. Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 72–79, May 2014.

[8] 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*, pages 119–124, Sep. 2008.

[9] 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)*, volume 00, pages 133–140, Apr 2018.

[10] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *Proceedings of the 21th International Workshop on Field-Programmable Logic and Applications (FPL'11)*, September 2011.

[11] 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)*, Apr 2019.

[12] P. Maidee, A. Kaviani, and K. Zeng. LinkBlaze: Efficient global data movement for FPGAs. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2017.

[13] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Third International ACM Symposium on Field-Programmable Gate Arrays*, pages 111–117, Feb 1995.

[14] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 77–86, New York, NY, USA, 2012. ACM.

[15] J. Rubinstein, P. Penfield, and M. A. Horowitz. Signal Delay in RC Tree Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(3):202–211, July 1983.