

# An Open-source Lightweight Timing Model for RapidWright

Pongstorn Maidee, Chris Neely, Alireza Kaviani and Chris Lavin  
Xilinx Research Labs, San Jose, CA, USA  
pongstorn.maidee, chris.neely, alireza.kaviani, chris.lavin @xilinx.com

**Abstract**—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. We propose a nimble timing model for RapidWright that delivers high fidelity timing approximations while enabling faster algorithms through a frugal memory footprint. By leveraging a combination of architectural knowledge, repeating patterns and extensive analysis of Vivado timing reports, we obtain a slightly pessimistic, lumped delay model within 2% average accuracy of Vivado for UltraScale+ devices. We validate the results with over 240 designs and the proposed model shows high fidelity to Vivado with a Spearman’s  $\rho$  value of 0.99. By open sourcing the proposed model and describing the process, we empower the community to leverage and extend this work for customized domains, other device families, and additional accuracy.

**Keywords**—FPGA, timing model, RapidWright, CAD tools

## I. INTRODUCTION

Advances in process technology have enabled FPGAs to grow in capacity and implement large heterogeneous systems in a device. FPGAs often provide better performance per watt over alternative solutions; Xilinx Alveo U250 shows 2X higher Images/Second/Watt over an equivalent Nvidia implementation [1]. While these silicon advances enable broader compute usage, they also require tools that appeal to high-level software programmers. High level synthesis tools are somewhat addressing this challenge by raising the abstraction of design entry, but backend place and route compile times still pose a significant hurdle. The RapidWright framework [2] fosters a movement to tackle the challenges of backend implementation tools.

RapidWright is an open source platform providing a gateway to Xilinx’s backend implementation tools that raises the implementation abstraction, while maintaining the full potential of advanced FPGA silicon. It allows the FPGA community to earnestly explore the boundaries of performance and productivity, while enjoying the credibility of commercial FPGA silicon. Until now, RapidWright has deferred timing-relevant tasks to Vivado which may slow down custom flows for certain applications. In this work, we introduce an open source timing model, with the goal of being lightweight and extensible, while providing effective timing information for a wide range of applications.

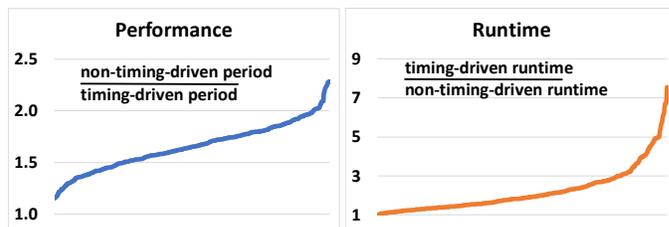


Fig 1. Performance benefit and runtime overhead of timing-driven algorithms.

To motivate the goals of this timing model, consider Figure 1, which conceptually demonstrates the performance advantages of timing-driven algorithms at the cost of runtime. The figure compares the performance and runtime for routing 240 synthetic designs in Vivado, showing more than 2X performance potential at the expense of up to 8X routing compile time. Is it possible to reduce this runtime penalty, but maintain the performance benefits?

Two significant challenges exist to creating a lightweight, yet useful timing model: 1) commercial FPGA vendors do not typically release detailed timing information for their devices and 2) providing large timing datasets can hamper performance due to their large memory footprint. In this work we address these challenges with a two-fold strategy: 1) providing an open source, data-driven, extensible model that enables the community to explore a variety of timing-driven algorithms and 2) raising the abstraction of the delay model in order to create a smaller memory footprint.

The key obstacle for timing representation is that the same technology advances that have made near GHz performance possible for the latest silicon devices, create a fine level of timing heterogeneity for the resources. Most of the FPGA architecture is still composed of replicated tiles, but low-level timing data differ due to silicon layout details explained in the next sections. Our overarching approach to address this obstacle includes three pillars: 1) grouping finer, less impactful delays, 2) first-order architectural modeling and adjustments, and 3) masking rare corner cases. Grouping fine-grained delay information into coarser Timing Groups (TGs) enables a smaller memory footprint raising the useful abstraction level. Referring to the second pillar, we model common repeating architectural patterns with adjustment to account for device heterogeneity. Finally, for rare corner cases that won’t affect the overall routability, we simply mask those resources in RapidWright’s device

model to make them unavailable during timing-driven algorithms. Specific contributions of this work include:

- 1) An open source, lightweight timing model for commercial devices that is extensible and data-driven.
- 2) Specific case studies and APIs showing how the delay model can be used in the RapidWright framework.
- 3) Validation of the model using Vivado along with the process description on how to extend the model or increase the accuracy.

The next section summarizes relevant background. Section III describes the delay model, followed by section IV that validates the model. Section V shows how the model can be used in RapidWright framework. Sections VI and VII summarize the related work, concludes the paper and point to future work.

## II. BACKGROUND

### A. RapidWright

RapidWright is an open source framework for backend FPGA implementation work and provides sufficient architectural detail about Xilinx devices to perform detailed placement and routing tasks. RapidWright also enables replicating pre-implemented circuits, allows exploration of domain-specific tool flows, and helps with building experimental tools and algorithms.

Included with RapidWright is a basic, non-timing driven router that uses a cost function based on Manhattan distance [2]. Due to the lack of availability of commercial FPGA timing data, RapidWright does not currently have any timing-driven algorithms, which limits its usefulness in certain situations. A goal of this work is to enable the RapidWright ecosystem with an open source timing-driven toolset for Xilinx FPGAs.

### B. UltraScale+ Architecture

Traditional FPGAs include Configurable Logic Blocks (CLBs) that are arranged in a regular array. Each CLB includes an interconnect switch matrix for access to the general routing resources, which run vertically and horizontally between the rows and columns. Interconnect, also called routing, is segmented for optimal speed-performance. Segmented routing wires in the UltraScale+ architecture span 1, 2, 4, or 12 CLBs to ensure that all signals can be transported from source to destination with ease and efficiency. We refer to these resources as Single (S), Double (D), Quad (Q), and Long (L) for length 1, 2, 4, and 12, respectively. This *segmentation* is the key notion

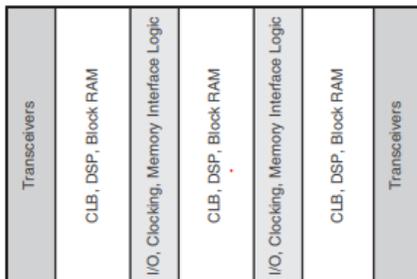


Fig 2. FPGA with columnar resources.

behind the simplified equation-based delay model that will be explained further in the next section.

Figure 2 depicts another relevant architectural factor that UltraScale devices are arranged in a column-and-grid layout. Columns of CLBs and hard blocks are combined in different ratios to provide the optimum capability for the device density, cost, and target market. Interconnect is the programmable network of signal pathways between the inputs and outputs of functional blocks within the device, such as IOBs, CLBs, DSPs, and Block RAMs (BRAMs). This columnar architecture enables different devices to have a mix of varying features that are optimized for different application domains [3]. However, columns of hard blocks introduce timing *discontinuity* in the routing architecture, especially in the horizontal dimension as hard blocks are wider than the typical logic resource stride. This is mainly due to additional wire length needed to traverse those blocks and our proposed delay model needs to adjust for that delay increase.

Most of the interconnect features are transparent to designers. However, details of device interconnect are available through Vivado Tcl APIs and can be used to guide design techniques. Interconnect delays vary according to the specific implementation, neighboring context and capacitive loading of a specific wire. The type of interconnect, distance required to travel in the device, and number of switch matrices to traverse factor into the total delay. Conventional efforts to meet design timing requirements are addressed by ensuring they are properly expressed through constraint files. When automatic place and route tools fail to meet these requirements, manual examination of the design’s critical path delays is undertaken. This generally leads to design changes such as using fewer logic levels or leveraging faster paths.

To gain better insight into architectural features for timing performance, consider Figure 3 which shows two CLBs, including one back to back interconnect (INT) tile, and two (Configurable Logic Element (CLE) tiles on the right and left side. INT tiles connect to each other through S/D/Q/L wire segments in the four cardinal directions. Each back to back INT tile includes a set of long and global resources that are shared for both CLBs and two similar sets of resources for the west and east CLE tiles. The name of a wire segment is encoded with these details. For example, a Quad wire segment traveling north on the west CLB is labeled NN4\_W. The INT tile containing these resources is the centerfold of the delay model presented in the next section.

The CLE tiles shown in Figure 3 contain one slice site and several Basic Elements of Logic (BELs). BELs are building blocks such as LUTs and FFs that implement the functionality of a design. There are eight 6-input LUTs, labeled A to H in each slice. Each 6-input LUT is associated with two FFs and two additional input pins labeled, I and X. Hard blocks such as DSP

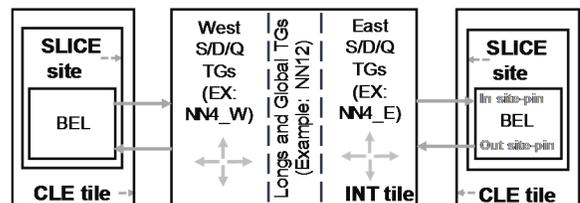


Fig 3. Two CLB tiles and their connectivity.

and BRAM replace one of these CLE tiles in a columnar architecture. The width of these hard blocks varies and is much wider than a CLE tile. Therefore, our model requires a discontinuity factor to adjust for such columnar architecture heterogeneity, as described in the next section.

### III. THE PROPOSED DELAY MODEL

Segmentation routing and columnar floorplan are first order architectural factors that affect the delay of the nets in a design. However, silicon process technology miniaturization advances have adversely impacted wire delay due to implementation and layout. Examples include non-uniform scaling of metal layers and dielectrics, wire width and spacing requirements as well as many other physical design rules beyond the scope of this paper. An accurate and comprehensive delay model needs to include fully elaborated delay scenarios to consider all these architectural and layout intricacies. Vivado employs such a delay model to guarantee complete accuracy of the reported performance at the expense of a larger memory footprint. In contrast, the RapidWright framework is not obligated by the same guarantee and is free to explore implementation tradeoffs for a more lightweight approach. To take advantage of this fact, we incorporate an equation-based delay model that captures the main architectural patterns explained. This drastically reduces RapidWright’s memory footprint of the delay model compared to Vivado’s exhaustive, enumerated case approach.

#### A. Equation-based delay model

To calculate the total delay of a timing path, one must include the sum of all configured inter-site interconnect resource delays and a fixed portion associated with intra-site or logic delays. Rather than fully enumerating the delays for each atomic interconnect resource, we introduce the notion of a *Timing Groups* (TG), which is a lumped set of inter-site switches and nodes to minimize memory requirements. TGs represent distinct decision points in the inter-site routing graph, and finer timing granularity is unnecessary in many cases. Figure 4 shows an example of a TG that represents a wire segment and its corresponding switch multiplexers and driving buffer. A MUX is used to provide routing flexibility to the segment. Connections between a MUX input and output are called programmable interconnect points (PIPs). A connection between MUXes is called a node. A node may contain several physical short wires, depending on the tiles it spans. The TG’s PIPs, buffer, and the wire segment end points all reside inside an INT tile. Note that it is not possible to route from the internal node in the TG shown in Figure 4 outside the scope of the group. Therefore, TGs can generally reduce the size and number of

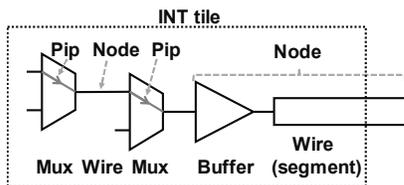


Fig 4. Timing group example.

objects examined during routing without compromising routability. Equation 1 calculates the delay of a TG

$$\text{delay}(TG) = k_0 + k_1 \cdot L(TG) + k_2 \cdot d(TG) \quad (1)$$

, where  $L(TG)$  represents the length of the wire segments, and  $d(TG)$  captures the additional distance a wire travels beyond a CLE column in the architecture.  $L(TG)$  is defined for each TG, but  $d(TG)$  is sum of  $d$  for all the tiles that TG passes through.  $k_0$  is the intrinsic delay constant and  $k_1$  is the coefficient constant corresponding to the segmented routing architecture.  $k_2$  is the coefficient constant to adjust the delay for columnar architecture discontinuities explained in the previous section. These three constants, which capture first-order architectural factors in the delay model, will be obtained through curve fitting.

Figure 5 shows the results of fitting delay for one of the most common TGs: vertical wire segments. There are two data points, representing vertical TGs. If the delays for the east or west side of the INT tile are different, we choose the highest number to be pessimistic.  $L$  and  $d$  are independent variables that can be selected during curve fitting. We often start with the nominal length a TG (i.e. a double TG is spans two CLB tiles) and adjust it to minimize the curve fitting error. All the data points are extracted using Vivado’s Tcl API and synthetic idiom designs using the target resources. Figure 6 shows a similar linear approximation for horizontal resources. Keep in mind both Figures 5 and 6 focus on wire segments that don’t cross architectural discontinuities. For these resources the constant coefficient  $k_2$  is not used. As a general rule in the fabric architecture, there is only one discontinuity in the vertical direction caused by clocking resources. In the horizontal dimension, however, many discontinuities exist due to the variety of column resource types (DSP, BRAM, URAM, etc.) that might be traversed in a given context. For these discontinuities, we calculate values of  $k_2$  for each resource type to capture delay characteristics of crossing the columns.

Each path delay is a linear combination of delays of all resources along the path. These delays can be represented as a system of equations and solved using a Least Square Approximation [5,6,7]. The extent of delay increase can be different for different types of TGs even when they cross the same hard block. Therefore,  $d$  in Equation 1 is a function of TGs. Delay adjustment of TGs is obtained by subtracting the value of the first two terms in the equation from the net delay. Figure 7 shows an example plot for delays of horizontal quads TGs versus  $d(TG)$ . The

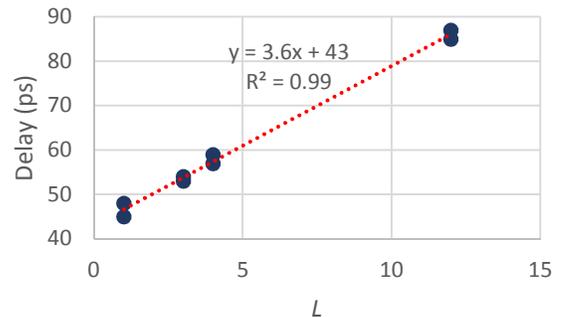


Fig 5. Curve fitting for vertical wire segments.

horizontal axis  $d$  does not directly correspond with the physical width of the columns. We can indirectly adjust  $d(TG)$  by adjusting  $d$  of hard block tiles (see Table 2). Quad wires can cross many types of hard block columns, but we could still obtain a highly correlated fit as seen in Figure 7. It is important to note that the interception point must be 0 in this case because we are not allowed to change  $k_0$ . Deriving the 3<sup>rd</sup> term of vertical TGs is similar, but simpler because only RCLK tile has non-zero  $d$  that requires adjustment.

### B. Segmentation and discontinuity parameters

We adopt a successive difference approach to determine the delays of TGs. The approach can be viewed as solving a system of equations forming a sparse triangular matrix for one variable at a time. The delay of two-terminal nets whose source and sink are in the same slice is determined first. By placing the source and sink on different LUTs within a given slice, we can find the delay of input site-pin TGs from the net delay reported by Vivado. Next, the delay of Single and Double TGs will be determined by placing source and sink at appropriate distances. The delay of Single and Double TGs is found by subtracting an appropriate input site-pin delay from the obtained net delay. The delay of Quad and Long TGs can be derived in a similar way.

Table 1 summarizes the constants for all the TGs in the interconnect that correspond to the first and second term of equation 1. As expected,  $k_0$  and  $k_1$  of horizontal and vertical TGs are similar as the architecture is designed to be symmetric. It is important to note that independent variable  $L$  does not always correspond to the length of target TG. Fitting the delay to Equation 1 yields coefficients as shown in Table 1; many other factors

such as layout, wire spacing, and width affect the delay. Global TG has no physical  $L$  associated and the length is merely virtual to calculate the delay.

Table 2 summarizes the constant  $k_2$  and corresponding  $d$  for TGs that need adjustment to cross over architectural discontinuities. There is only one discontinuity, RCLK, in the vertical direction. This is in the middle of each clock region, dedicated to some clocking resources. All other hard blocks listed in the table are for the horizontal dimension. To find  $d$  of a horizontal TG, we only need to compute and store  $d$  values for one row because the UltraScale+ architecture is columnar. We compute cumulative distances from a common point (the left most INT tile) to every point of interest and store them in an array to be used for any given TG. Because TGs straddle between INT tile, we need to store cumulative  $d$  only on INT tiles. As a result, the size of a  $d$  array linearly depends on the number of INT tiles.

All the parameters are characterized for the -2 speed grade. The delays can be extended for -1 and -3 speed grades using multiplication factor of 1.15 and 0.85, respectively. In general, there are more than 4K PIPs and other unique resources in one INT tile, which are abstracted to only 26 equivalent TGs in our model. This implies more than two orders of magnitude reduction in the timing model size for each tile. Moreover, we extend the concept of equivalent TGs to all the device tiles to reduce the memory footprint even further. Such equivalency is based

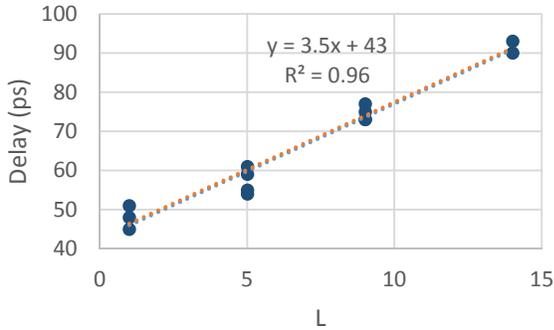


Fig 6. Curve fitting for horizontal wire segments.

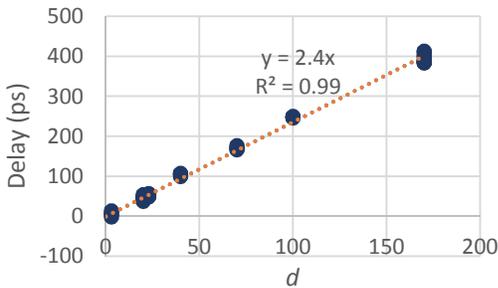


Fig 7. Curve fitting for  $k_2$  of horizontal Quad TGs.

Table 1. Segmentation parameters for delay model.

Coefficient	Hor.	Ver.
k0	43	43
k1	3.5	3.6

L of	Hor.	Ver.
Bounce	0	-
Single	1	1
Double	5	3
Quad	10	5
Long	14	12
Global	13	-

Table 2. Discontinuity parameters for delay model

k2 of	Hor.	Ver.
Single	2.3	13.5
Double	2.3	5.5
Quad	2.4	9.5
Long	1.3	3.5

d of	S, D	Q	Long
DSP	3	3	3
BRAM	16	20	20
CFRM	30	30	20
URAM	34	40	40
PCIE	65	70	140
IO	170	170	300
RCLK	3	3	3

on the last node of the TG being of the same type and is determined by TG direction, length of wire segment, and the hard block tiles on its path. Our proposed model will provide one delay value for each equivalent TG, which results in an implementation consuming less than one kilobyte of memory.

### C. Corner cases and exceptions

The equation with parameters explained previously captures first and second order architecture effects on the delay. However, there are a few corner cases that are often due to layout complexities and irregularities in the architecture. The proposed model could have been augmented to cover these cases at the expense of the model simplicity. We chose to keep the model simple and address these cases using two approaches.

1) Pessimistic delay approach: When there are multiple delays for similar TGs crossing a discontinuity column we choose the higher delay. This will make our delay somewhat pessimistic when it comes to corner cases. For example, such discrepancies often happen close to wide columns, such as IO. Our approach will encourage the algorithms to avoid crossing such blocks when designed in RapidWright.

2) Masking resources approach: When an exceptional case occurs, we disallow the use of such resources without impacting routability in RapidWright. For example, horizontal long wires on rows adjacent to RCLK tile may have large delay difference. Such rare cases constitute roughly 0.2% of all long wires ending at a given column. Therefore, we can easily mask those segments from routing resources in RapidWright with little impact on routability.

A placement or routing algorithm would be discouraged or prevented from using these corner resources with our approach. As a result, when the design is read into Vivado to finalize routing and close timing, the outcome might slightly improve in performance or routability.

Table 3. Net delay estimating error.

	Error (ps)	Error %
Avg	0.8	0.7
StdDev	12.9	3.6
Min	-53.9	-12.7
Max	40.0	14.9

Table 4. Critical data path delay prediction errors. Positive values indicate pessimism.

	Typical frequency (T = 2 ns)	Near spec frequency (T = 1.3 ns)
Avg	33 ps (1.9%)	52 ps (3.5%)
Min	-106 ps	-35 ps
Max	127 ps	132 ps

## IV. MODEL VALIDATION

The accuracy of the model was evaluated using two sets of experiments using Vivado as the golden reference. First, we compare delays reported by Vivado against those computed by our model using point to point idiom designs. Second, we used a large set of synthetic designs and compared the overall achieved clock periods estimated by our model with those reported by Vivado.

### A. Estimating point-to-point net delay

In this evaluation, we randomly placed the source and sink of a two-pin net within a region. To limit the number of experiments to a manageable size, the placements are restricted to the top half of a clock region. In addition, subregions without wide columns, such as URAM, PCI and IO are selected to avoid any pessimistic bias, as explained in previous section. The experiments cover all possible routing directions. We also varied the vertical and horizontal distances of the route. The experimental net is connected between two LUTs to avoid interference from clock timing overhead. We then routed the net in Vivado using the `route_design` command in timing-driven mode. Finally, we compared the net delay reported by Vivado and the delay estimated by our model. In total, the experiment compared about 40k nets. Figure 8 shows the scatter correlation plot of the net delays computed by our model against those reported by Vivado. The plot visually shows high correlation (0.99) between the two measures. In particular, when the delay is below 300 ps, the error is quite small (< 50 ps).

The net delay comparison is summarized in Table 3, where a positive error value indicates that the model is pessimistic. On average, the model is pessimistic by 1.7%. In many use cases, fidelity—monotonic relationship between the estimated values and the actual values—of an estimation is more important than its absolute estimated accuracy [8]. A perfect fidelity model to estimate net delays would predict that a net has higher delay than that of another net, when indeed that is the case considering their actual delays. Our proposed model exhibits high fidelity to the net delays reported by Vivado as its Spearman’s  $\rho$  value, computed over the data shown in Fig 9, is 0.99, where the value of 1.0 indicates perfect fidelity [9].

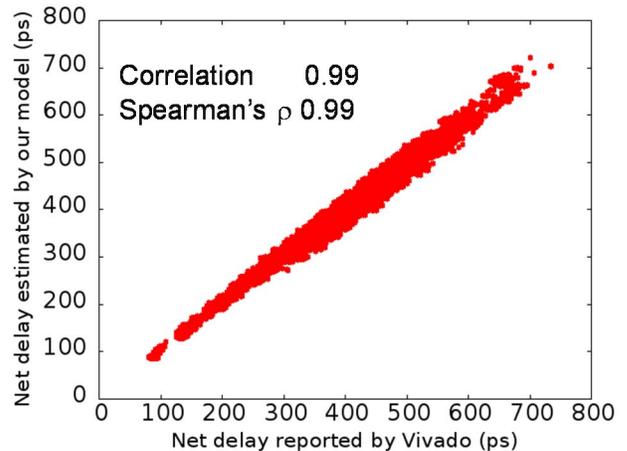


Fig 8. Correlation between our model net delay estimates and Vivado.

## B. Estimating achievable clock period

In the previous subsection, we showed the estimating errors when routing a net without routing contention. Another more comprehensive set of experiments was performed to further quantify the model accuracy. We generated 240 synthetic designs containing LUTs and FFs such that the LUT utilization over a region varies from 50% to 88%. For a given LUT utilization value, we also generated designs with Rent exponents varying from 0.3 to 0.8, and logic depth of 3, 5 and 7. Each design will be placed and routed with both 2ns (500MHz) and 1.3 ns (769MHz, near the device spec) target clock period. In total, 480 runs were executed. The setup is to measure the accuracy of the model without interference from clock skew, which is an orthogonal factor. The reference results from Vivado were also constructed to have zero clock skew for consistency.

Each design was placed and routed on a Virtex UltraScale+ VU3P (xcvu3p-ffvc1517-2-e) using Vivado 2018.3 for each target clock period. The data path delay of the critical path of each design is noted. Using our model, we computed the estimated delay for each sink for every net in the design. Then, the existing net delays are replaced by the computed delays before running the `report_timing` command again. The data path delay of the critical path as a result of using our delay model was computed.

The comparison is summarized in Table 4. Among all 440 runs, the model is pessimistic in predicting achieved periods by an average of 33 ps, which is within 2% of the data path delay of the most critical path reported by Vivado, for the typical frequency. The reported average was computed as an average of the absolute errors for each run. This is higher than the observed error per nets basis reported in the previous subsection. One factor is the compounding effect of multiple nets along the path. Another factor would be the contention among nets. For near-spec frequency, tighter clock period constraint reduces the delay budget, forcing the place and route tools to use resources with smaller delays when possible. As a result, high error can be observed when the target period was reduced to 1.3 ns. The table also highlights the averages for the extreme optimistic and pessimistic (Min and Max) cases.

## V. DEPLOYING TIMING MODEL WITHIN RAPIDWRIGHT

This section describes deployment of the proposed timing model in the RapidWright framework. Adding a timing package to RapidWright makes it possible to get feedback on timing readily and to estimate the quality of the implementation quickly. Without such a timing package, Vivado must be used to determine the critical path, which will significantly increase the design turn-around time. This new timing package is designed to be data-driven with a nimble implementation that leverages the proposed lightweight timing model.

Table 5. Routing simple nets in RapidWright.

Distance	Timing-driven	Non-timing driven	Timing Improved
dx = 8, dy = 0	444 ps	503 ps	11.7%
dx = 4, dy = 16	704 ps	813 ps	13.4%

The delay model parameters described in Section III are read from a data file, which can be easily modified for those wishing to improve or extend the timing package. The  $d$  values corresponding to the columnar architecture are precomputed in an array to identify the third term of Equation 1 for each timing group at runtime. We demonstrate the RapidWright timing package using two case studies. The first case study shows how routing can be done with timing information, which is targeted at users who want to create their own routers using this timing package. The second case study shows how to report timing in RapidWright, which is targeted at users who want to know the critical path in their routed design.

### A. Enabling timing-driven algorithms

A set of APIs and a prototype router was implemented to use the delay of timing groups as part of the cost function. We use this example to illustrate the use of our timing model; the implementation details are beyond the scope of this paper but can be found in the open source code. Table 5 shows two example nets routed within RapidWright in both timing-driven and non-timing driven modes. The two example nets are different in both distance and direction. The “dx” and “dy” in the table describe the distance in terms of tile column and row coordinates, respectively. We can observe more than 11% improvement in the data path delay of these example nets. For the first net “dx=8, dy=0” the timing-driven RapidWright router selects two double TGs and one pin bounce TG. In contrast, the RapidWright built-in router without timing information selects one quad TG, one global TG, and one pin bounce TG. Although both routes use the same number of TGs, the non-timing-driven router selected a global TG, unaware that it has very high delay.

Table 6 shows the same two example nets, but the timing is reported by Vivado. The purpose of this table is to examine the accuracy of route delays reported in RapidWright. The numbers in parentheses show the error (in %) when comparing to Table 5. The data in Table 6 shows that RapidWright timing report error is 3% and 0.6% for the two example nets. A second take away is that the error is very small for non-timing-driven routing in RapidWright, validating the accuracy of our model for that route. Delay numbers in the right column of Table 6 show the results when we route the same nets with Vivado. The Vivado router selects a similar path to that of the RapidWright timing-driven router, with less than 1% delay error, indicating that our router has chosen a high-quality path. We believe the simple timing APIs provided in the open source timing package will allow reasonably accurate implementations of timing-driven algorithms.

Table 6. Vivado timing results for simple nets routed in RapidWright.

Distance	Timing-driven	Non-timing driven	Vivado router
dx = 8 dy = 0	431 ps (3%)	504 ps (0.2%)	429 ps (0.5%)
dx = 4 dy = 16	700 ps (0.6%)	813 ps (0%)	706 ps (-0.9%)

## B. Reporting timing on a routed design

The second case study demonstrates using the RapidWright timing package to report timing of a realistic design that has already been routed by Vivado. Our first example design is based on the "five-tuple" parser, which is part of Xilinx SDNet 2018.2 compiler [12]. SDNet is Xilinx's tool for accelerating network packet processing applications. The parser has a 512b input port and a 512b output port for packets, and an additional output port for the extracted 112b five-tuple key. This version of the example parser design contains 2.9k LUTs and 7.7k FFs on a Xilinx VU3P.

The RapidWright timing package builds a timing graph [10] for this task as it is a convenient data structure for organizing the net and logic delays of designs. In our timing graph, vertices represent pins in the design and edges represent either: (a) physical nets connecting each source pin to a sink pin or (b) logic delays from LUTs. The paths within the graph start from the source pins of registers and inputs. The paths terminate at the sink pins of registers and outputs. The timing graph for this example parser contains ~43k vertices and ~45k edges. Some of the vertices and edges are logical rather than physical and are used for keeping track of cell hierarchy in the design. We use the Java graph library JGraphT library [13] to help implement our timing graph.

Vivado computes the critical path with a total delay of 2742 ps. The path in the RapidWright graph having the maximum delay has an estimated total delay of 2672 ps (2.55% error). The critical path for this example contains the four edges in the graph, representing two levels of logic and net delay and some design hierarchy. The logic and net delays along the critical path for the example parser design is shown in the table within Figure 9, which compares these edges to the actual delays reported by Vivado. The total delays for this timing path were similar, within 70 ps.

The second example design is based on the Xilinx PicoBlaze microcontroller [14]. The timing graph for this example design contains 1.5k vertices and 1.8k edges. Vivado computes the critical path with a total delay of 1609 ps. The path in the RapidWright graph having the maximum delay has an estimated total delay of 1645 ps (2.24% error). The logic and net delays along the critical path for the example processor design is shown in the table within Figure 10, which compares these edges to the reference delays reported by Vivado.

Delay	RW	Vivado
Logic	78 ps	77 ps
Net	311 ps	341 ps
Logic	150 ps	150 ps
Net	2133 ps	2174 ps
<b>Total</b>	<b>2672 ps</b>	<b>2742 ps</b>

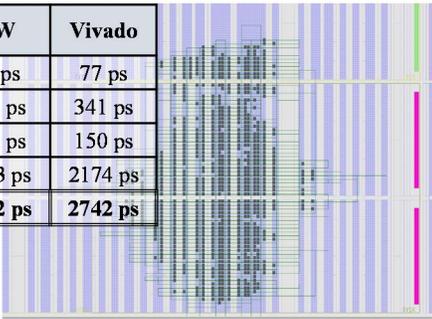


Fig 9. Example 1: Parser design for RapidWright report-timing, including critical path details.

## VI. RELATED WORK

One of the earliest openly available delay models for FPGA architecture was made in VPR [15] using an Elmore delay model. As the architecture can be artificially created in VPR, specific attributes can be chosen and made readily available. In contrast, commercial FPGA vendors do not provide such fine-grained architectural detail and thus this approach would not work for commercial architectures.

Recent work on a timing model for use within the RapidWright framework has been made for a domain specific router called RapidRoute [5]. RapidRoute is a router targeted specifically at routing networks on FPGAs. As their specific routing problem is well-formed due to the regular nature of routing network busses, they were able to achieve an accurate timing model (<1% error) after only a few dozen calibration runs. In contrast, this work aims to be a more general solution for many kinds of routing scenarios. The work in [5] relies on regularity of FPGA structure, which is no longer true for general use cases and latest devices such as UltraScale+. For example, Figure 7 shows that the delays of Quads crossing different columns vary significantly due to different wire length and delay characteristics. Our results indicate that a model with an acceptable level of accuracy needs to take into consideration the columnar structure of FPGAs.

In GROK-INT [6] and GROK-LAB [7], the authors described a technique to extract delays of interconnect and logic blocks, respectively. Their goal is to characterize on-chip delay variation of each resource to be used in routing. Measuring delay at various points must be done in hardware for such characterization. The concept of Logical Component Nodes (LC nodes) containing a set of resources was introduced by necessity as they are the smallest measurable units. Any path can be represented using LC nodes. Our goal is to abstract and represent delay information in Vivado using a small memory footprint. Therefore, we only need to perform software experiments through Vivado interface. Our proposed timing groups are coarser than those in [6,7] and they are not always the smallest measurable units. We argue that there is not much benefit of providing delay estimate for each component of a TG, due to architecture connectivity patterns. It is not our goal to attribute estimated delay proportionally to the real delay as long as we can estimate the path delay accurately.

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
<b>Total</b>	<b>1645 ps</b>	<b>1609 ps</b>

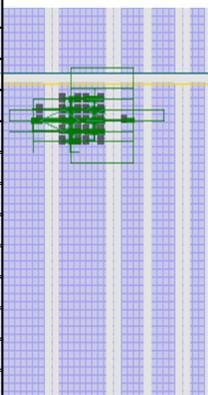


Fig 10. Example 2: PicoBlaze design for RapidWright report-timing, including critical path details.

## VII. CONCLUDING REMARKS

In this paper, we have described a high fidelity timing model approximation for Xilinx UltraScale+ devices that is on average within 2% accuracy of Vivado timing reports with a  $\rho$  value of 0.99. Specific case studies of this timing model achieving high quality of results comparable with Vivado have been shown. We have also demonstrated that exhaustive, detailed timing information is not necessary to achieve high quality results and that a prudent, lightweight course-grained model is sufficient in all but the most extreme cases. The implications of such a lightweight model (with less than a kilobyte memory footprint) instills excitement and promise for embedded efforts where place and route algorithms must run in highly constrained memory environments such as the Zynq platform [11].

Referring to the initial question of reducing compile time while maintaining the performance, we envision this work to set a foundation for exploring fast timing-driven FPGA tools. Emerging application domains and use cases require unique CAD solutions and we believe an extensible open source timing model can help satisfy the growing needs. By open sourcing this timing model for RapidWright, we hope to drive new research in the areas of faster place and route, memory-constrained CAD algorithms and FPGA productivity.

In this work we focused on modeling interconnect and a few basic logic blocks. Extracting the delay of other logic blocks from Vivado is straightforward and can incrementally be added to this data-driven model. One main value proposition of this paper is providing an equation-based delay model with a smaller memory footprint. We have implemented this concept, which is finding the coefficient constants and the data, for UltraScale+ family. However, the equation and all the architectural concepts can be extended to other family members. We envision this work to guide the open source community to add the data for other families incrementally. The reported results assumed zero clock skew for both reference Vivado results and ours for consistency. The setup is to measure the accuracy of the model without interference from clock skew, which is an orthogonal factor. Clocking modeling can be added to improve accuracy as part of future work by the community. Future work will also tackle other challenges such as net fan out compensation or improving accuracy.

## DISCLAIMER

The opinions expressed by the authors are theirs alone and do not represent the opinions of Xilinx and are not indications of any future policy on FPGA software or hardware held by Xilinx.

## REFERENCES

- [1] Xilinx, "Accelerating DNNs with Xilinx Alveo Accelerator Cards," White Paper, WP504 (v1.0.1), October 14, 2018
- [2] C. Lavin and A. Kaviani, "RapidWright: Enabling Custom Crafted Implementations for FPGAs," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 2018, pp. 133-140.
- [3] Xilinx, "UltraScale Architecture and Product Data Sheet: Overview," DS890 (v3.9) June 27, 2019
- [4] Leo Liu, Jay Weng, Nachiket Kapre, "RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 2019, pp. 61-64
- [5] Leo Liu and Nachiket Kapre, "Timing-aware routing in the RapidWright framework," 2019 29th International Conference on Field-Programmable Logic and Applications (FPL), Barcelona, Spain, 2019.
- [6] B. Gojman and A. DeHon, "GROK-INT: Generating Real On-chip Knowledge for Interconnect Delays Using Timing Extraction," 2014 IEEE 22th International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boston, MA, USA, 2014, pp. 88-95
- [7] B. Gojman, et al., "GROK-LAB: Generating Real On-chip Knowledge for Intra-cluster Delays Using Timing Extraction," ACM Transactions on Reconfigurable Technology and Systems, Vol. 7, No. 4, December 2014.
- [8] J.T. Russell and M.F. Jacome, "Architecture-level performance evaluation of component-based embedded systems," 40<sup>th</sup> ACM/IEEE Design Automation Conference (DAC), Anaheim, CA, USA, 2003.
- [9] H. Javaid, A. Ignjatovic and S. Parameswaran, "Fidelity Metrics for Estimation Models," 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 2010.
- [10] A. Marquardt, et al., "Timing-Driven Placement for FPGAs," ACM Transactions on Reconfigurable Technology and Systems, Vol. 7, No. 4, December 2014.
- [11] D. Glick, J. Grigg, B. Nelson and M. Wirthlin, "Maverick: A Stand-Alone CAD Flow for Partially Reconfigurable FPGA Modules," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 2019, pp. 9-16.
- [12] Xilinx SDNet, "SDNet Packet Processor User Guide," [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2018\\_2/ug1012-sdnet-packet-processor.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf).
- [13] JGraphT, "Homepage," <https://jgrapht.org/>.
- [14] Xilinx PicoBlaze Microcontroller, "PicoBlaze 8-bit Microcontroller," <https://www.xilinx.com/products/intellectual-property/picoblaze.html>.
- [15] V. Betz, "Architecture and CAD for speed and area optimizations of FPGAs," Ph.D. Thesis, University of Toronto, National Library of Canada, 1998.