RWRoute: An Open-source Timing-driven Router for Commercial FPGAs

YUN ZHOU, Ghent University, Belgium
PONGSTORN MAIDEE, Xilinx Research Labs, USA
CHRIS LAVIN, Xilinx Research Labs, USA
ALIREZA KAVIANI, Xilinx Research Labs, USA
DIRK STROOBANDT, Ghent University, Belgium

One of the key obstacles to pervasive deployment of FPGA accelerators in data centers, is their cumbersome programming model. Open source tooling is suggested as a way to develop alternative EDA tools to remedy this issue. Open source FPGA CAD tools have traditionally targeted academic hypothetical architectures, making them impractical for commercial devices. Recently, there have been efforts to develop open source back-end tools targeting commercial devices. These tools claim to follow an alternate data-driven approach that allows them to be more adaptable to the domain requirements such as faster compile time. In this paper, we present RWRoute, the first open source timing-driven router for UltraScale+ devices. RWRoute is built on the RapidWright framework and includes the essential and pragmatic features found in commercial FPGA routers that are often missing from open source tools. Another valuable contribution of this work is an open-source lightweight timing model with high fidelity timing approximations. 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. Compared to Vivado, RWRoute results in a 4.9× compile time improvement at the expense of 10% Quality of Results (QoR) loss for 665 synthetic and 6 real designs. A main benefit of our router is enabling fast partial routing at the back-end of a domain-specific flow. Our initial results indicate that more than 9× compile time improvement is achievable for partial routing. The results of this paper show how such a router can be beneficial for a *low touch* flow to reduce dependency on commercial tools.

CCS Concepts: • **Hardware → Electronic design automation**; **Physical design (EDA)**; **Wire routing**; **Software tools for EDA**.

Additional Key Words and Phrases: FPGA routing, timing-driven, commercial FPGAs, RapidWright, timing model, Ultrascale+

ACM Reference Format:
Yun Zhou, Pongstorn Maidee, Chris Lavin, Alireza Kaviani, and Dirk Stroobandt. 2021. RWRoute: An Open-source Timing-driven Router for Commercial FPGAs . *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (October 2021), 28 pages. https://doi.org/10.1145/3491236

1 INTRODUCTION

FPGAs have made their way into data centers as compute, network, and storage accelerators to keep up with the insatiable demand of a data-driven digital world. This new paradigm increases demand for more software-like FPGA design experience similar to that of traditional compute engines such as CPUs. Innovation on the front-end of the

Authors' addresses: Yun Zhou, Ghent University, Technologiepark-Zwijnaarde 126, Ghent, Flanders, 9052, Belgium, Yun.Zhou@ugent.be; Pongstorn Maidee, Xilinx Research Labs, 2100 Logic Drive, San Jose, CA, 95124, USA, Pongstorn.Maidee@xilinx.com; Chris Lavin, Xilinx Research Labs, 3100 Logic Drive, Longmont, CO, 80503, USA, Chris.Lavin@xilinx.com; Alireza Kaviani, Xilinx Research Labs, 2100 Logic Drive, San Jose, CA, 95124, USA, Alireza.Kaviani@xilinx.com; Dirk Stroobandt, Ghent University, Technologiepark-Zwijnaarde 126, Ghent, Flanders, 9052, Belgium, Dirk.Stroobandt@ugent.be.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).
Manuscript submitted to ACM

hardware development flow has helped raise the abstraction of design entry to software languages. However, the back-end implementation tools such as placement and routing have largely been limited to FPGA vendor compilers with long compilation times. In order to approach software-like compilation times, back-end FPGA implementation will need a different strategy from what is currently employed in conventional FPGA flows.

One way to explore these strategies is to rely on back-end open source tools similar to the way traditional temporal computing software has evolved. This will help in two ways: first, enabling innovation by tapping into a large open source community, and second, making customized flows for specific target domains possible. Commercial vendors must optimize for generalized solutions that maximize availability across a wide range of customer design scenarios. However, as data center workloads become more domain-specific, a customized approach for back-end implementation is needed to achieve both compute efficiency of the solution and fast compile time. Older work on routing primarily focused on minimizing wirelength and resolving congestion and resulted in wirelength-driven routing tools. However, more recent timing-driven tools emphasize on meeting timing requirements. In this work, we have built RWRoute, an open source timing-driven router for UltraScale+ FPGAs. This router leverages the RapidWright framework and its lightweight open timing model [15] and is derived from CRoute [25], an open-source academic router. There are three main contributions of this work:

(1) RWRoute, a tool that routes designs in both wirelength-driven and timing-driven modes, enabling the open source ecosystem to innovate towards new algorithms. The open source aspect allows the community to develop domain-specific algorithms such as bundle routing in addition to adjusting the cost functions for the desired domain figure of merit.
(2) An additional routing mode that allows partial routing—an essential capability for a library-based customized flow. The libraries are often built out-of-context using commercial tools. The proposed router enables stitching the libraries together by quickly routing the inter-module nets.
(3) An open source, lightweight timing model for commercial devices that is extensible, data-driven and optimized for routing.

RWRoute can be adapted to target other FPGA architectures when sufficient connectivity and timing information is available. Experimental results obtained through routing of a number of synthetic designs and real applications show that the timing-driven router achieves a runtime speedup of 4.9× over Vivado, at the expense of a 10% penalty in performance on average.

The next section provides a summary of the relevant background followed by a section describing the proposed wirelength-driven routing and results. Section 4 describes the open source timing model along with its validation methodology. Section 5 summarizes the list of features for our proposed timing-driven router and the corresponding results, compared to those of the Vivado router. Section 6 summarizes previous related work and how our work differentiates with that described in the earlier literature, followed by concluding remarks in Section 7.

2 BACKGROUND

This Section summarizes the relevant background such as the framework, the target UltraScale+ architecture and routing fundamentals.

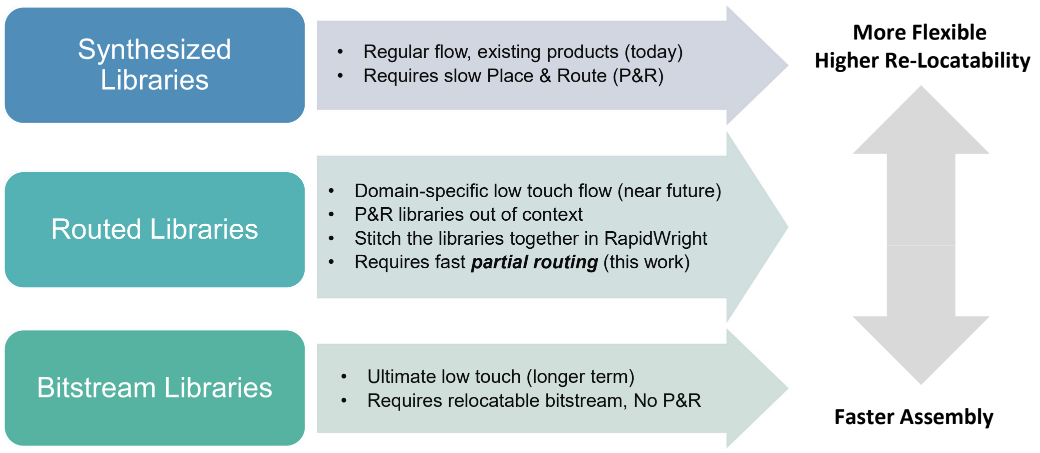

Synthesized Libraries
Regular flow, existing products (today)
Requires slow Place & Route (P&R)
Routed Libraries
Domain-specific low touch flow (near future)
P&R libraries out of context
Stitch the libraries together in RapidWright
Requires fast partial routing (this work)
Bitstream Libraries
Ultimate low touch (longer term)
Requires relocatable bitstream, No P&R
More Flexible
Higher Re-Locatability
Faster Assembly

Fig. 1. Library-based low touch flow.

2.1 RapidWright Open-source Framework

RapidWright [10] is an open source framework for back-end FPGA implementation and provides sufficient architectural detail about Xilinx devices to perform detailed placement and routing tasks. Although RapidWright provides a very basic Manhattan distance-based router, it does not resolve congestion and does not route clocks. Due to the lack of availability of commercial FPGA timing data, RapidWright did not have any timing-driven algorithms, which limited its usefulness in certain situations. This work enables the RapidWright ecosystem with an open timing model and demonstrates how a routing toolset for Xilinx FPGAs can use it.

In addition to helping with building experimental algorithms, RapidWright enables replicating pre-implemented circuits and allows exploration of domain-specific tool flows. An example of a customized domain-specific flow would be a *low touch* flow that stitches pre-implemented libraries together to assemble a complete design. Figure 1 depicts the main value proposition of a low touch flow towards the goal of an alternative programming model for FPGAs that is similar to that of traditional compute platforms. The top row in the figure shows today's regular flows, which often come with high compile times similar to traditional EDA tools. The middle row involves placing and routing modules of the design out of context to create reusable libraries. Assembling these libraries together will require routing inter-module nets and at times rerouting the library net to resolve conflicts or congestion. We refer to this step as partial routing, which can be done using commercial tools such as Vivado. However, the Vivado router is not designed for this purpose and we will show that RWRoute can achieve significant speedup over commercial counterparts. The bottom row of the figure points to an ultimate low touch flow, which might require architectural changes and is beyond the scope of this paper.

2.2 UltraScale+ Architecture

Traditional FPGAs include Configurable Logic Blocks (CLBs) that are arranged in a regular array. Each CLB includes a switch matrix for access to the general routing resources, which run vertically and horizontally between the rows and columns. UltraScale+ devices are arranged in a column-and-grid layout. Figure 2 depicts relevant architectural factors. Columns of CLBs and hard blocks, such as DSPs and block RAMs (BRAMs), are combined in different ratios to provide

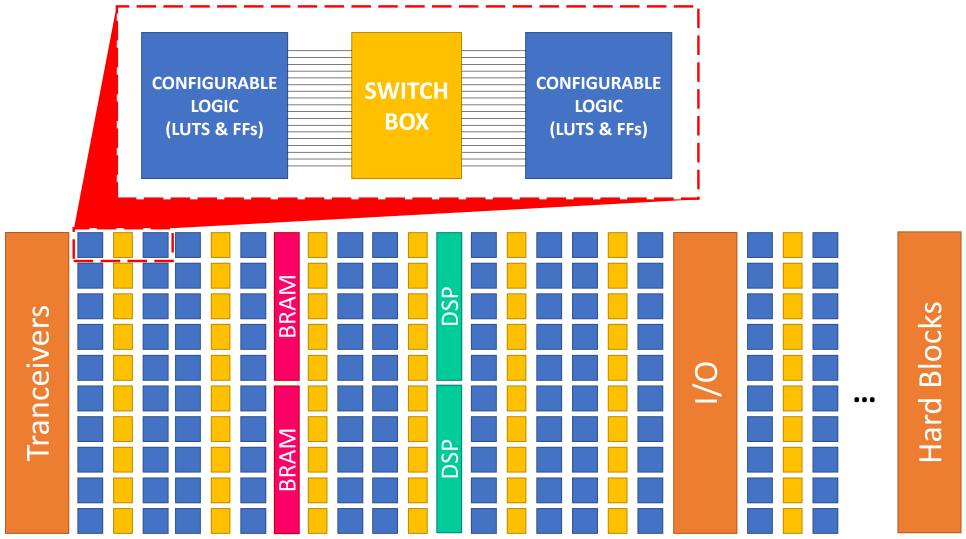

CONFIGURABLE LOGIC (LUTS & FFs)
SWITCH BOX
CONFIGURABLE LOGIC (LUTS & FFs)
Tranceivers
BRAM
BRAM
DSP
DSP
I/O
...
Hard Blocks

Fig. 2. FPGA with columnar resources.

the optimum capability for the device density, cost, and target market. This columnar architecture enables different devices to have a mix of varying features that are optimized for different application domains [7]. 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. Therefore, our proposed delay model requires a discontinuity factor to adjust for such columnar architecture heterogeneity, as described in section 4.

The inputs and outputs of functional blocks within the device, such as IOBs, CLBs, DSPs, and BRAMs, can be connected through the programmable interconnect network, also called routing resources. Interconnect (INT) is segmented for optimal speed-performance. Segmented routing resources in the UltraScale+ architecture span 1, 2, 4, or 12 INT tiles to ensure that all signals can be transported from source to destination efficiently. We refer to these resources as SINGLE (S), DOUBLE (D), QUAD (Q), and LONG (L) nodes for length 1, 2, 4, and 12, respectively. A node is a collection of electrically connected wires that spans one or more tiles [11]. This *segmentation* is the key notion behind the simplified equation-based delay model that will be explained further in Section 4.

To gain better insight into architectural features, consider Figure 3 including one back to back INT tile (i.e. the switch box shown in Figure 2), and two Configurable Logic Element (CLE) tiles on the right and left side. Each back to back INT tile includes a set of long and global resources that are shared for both CLEs and two similar sets of resources for the west and east CLE tiles. INT tiles connect to each other through S/D/Q/L nodes in the four cardinal directions. The name of a node is encoded with these details. For example, a Q node traveling north on the west CLE is labeled NN4_W.

Although most of the interconnect features are transparent to designers, device interconnect resources 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 node. 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

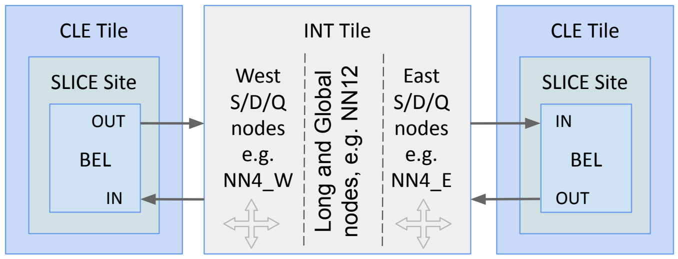

CLE Tile
SLICE Site
OUT
BEL
IN
INT Tile
West S/D/Q nodes e.g. NN4_W
Long and Global nodes, e.g. NN12
East S/D/Q nodes e.g. NN4_E
CLE Tile
SLICE Site
IN
BEL
OUT

Fig. 3. Two CLE tiles and their connectivity with the INT tile.

files. When automatic place and route tools fail to meet these requirements, manual examination of critical path delays is undertaken. This generally leads to design changes such as using fewer logic levels or leveraging faster paths.

2.3 CRoute: Connection-based FPGA Routing

The routing resources and the connectivity for a target FPGA are usually represented by a directed graph $G = (V, E)$, which is also called routing resource graph (RRG). The vertex set V and edge set E correspond to the routing resources in the FPGA architecture and the switches that connect these resources, respectively. In this way, the routing problem of each net is reduced to finding a sub-graph of the RRG. The sub-graph of each net is called a routing tree. The routing trees of nets should be disjoint so that there is no illegal resource sharing to avoid short circuits. The main algorithm in our router is driven from a connection-based FPGA router [24], which is based on the negotiated congestion router PathFinder [16].

The Pathfinder algorithm balances the competing goals of eliminating congestion and minimizing the delay of critical paths in an iterative framework. PathFinder allows nets to share resources initially and then gradually increases the penalty for sharing routing resources during subsequent iterations. The nets sharing resources, which are termed congested nets, are ripped up and rerouted every iteration until no resources are used by multiple nets. Historically, Pathfinder-based routers rip up and reroute all congested nets in every iteration, even when parts of the nets are legally routed. This results in redundant expansions and slow convergence. Since each net can be regarded as a set of source-sink connections, the routing problem can be simplified to find a single path in the RRG for each connection of the circuit [24]. These paths are allowed to share resources only when the corresponding connections are within the same net. Only congested connections are needed to be ripped up and rerouted in subsequent iterations. This connection-based routing strategy is much more time-efficient than traditional net-based routing, while retaining high quality of results (QoR), as shown in the state-of-the-art connection router CRoute [25, 29].

3 WIRELENGTH-DRIVEN ROUTING FOR COMMERCIAL ARCHITECTURES

Although our final target is a timing-driven routing tool, congestion avoidance is still a requirement. In this section, we start by explaining the changes needed in the wirelength optimization mode and cost functions that focus on features required for commercial devices. We will compare our results with the non-timing mode of Vivado as a reference to demonstrate the quality of the router. At the end of this section, we demonstrate how RWRoute can be used in this mode to improve compile time for partial routing in a low touch flow. In section 5, we will then add the timing-driven aspects.

3.1 General Signal Routing

General signal nets connect pins of different sites, also called inter-site nets. A general signal net is decomposed into a set of source-sink connections. Each connection of net N is routed through the expansion-based searching of resources traversing from its source to the sink. A bounding box is computed up-front for each connection to exclude expansion of routing resources that are less promising for the optimization goal. In CRoute [29], the bounding box of each connection at least contains its two pins and the geometric center of the entire net, with an extension on each side of the smallest bounding box.

To provide a regular coordinate system of routing resources in commercial architectures, the routing expansion is per INT tile. In each expansion step from a specific routing resource, each downstream neighbor of it, i.e. vertex n in the RRG, is evaluated with a total path cost. The total path cost $f(n)$ of the vertex n is computed as follows.

$$f(n) = c_{prev}(n) + c_{wld}(n) + c_{exp,wld}(n) \tag{1}$$

The upstream path cost $c_{prev}(n)$ is the sum of $c_{wld}(n)$ of all vertices from the vertex n (exclusive) to the source. The wirelength-driven cost $c_{wld}(n)$ of a vertex is shown in Equation (2), including a congestion cost and a bias cost $c_{bias}(b(n))$. The congestion cost depends on its base cost $b(n)$, the present congestion penalty $p(n)$, and a historical congestion penalty $h(n)$. The base cost indicates the cost of using the vertex n. The penalty factors $p(n)$ and $h(n)$ are related to the number of other nets presently using n and its history of congestion in previous iterations, respectively. The sharing factor $share(n)$ is introduced to facilitate resource sharing among connections within the same net [24]. It is equal to the number of connections from the same net that are using the vertex n.

$$c_{wld}(n) = \frac{b(n) \cdot p(n) \cdot h(n)}{1 + share(n)} + c_{bias}(b(n)) \tag{2}$$

$$b(n) = b_0 \cdot L(n) \tag{3}$$

$$c_{exp,wld}(n) = \alpha \cdot \left[\frac{\delta_{same} \cdot \bar{c}_{same}}{1 + share(n)} + \frac{\delta_{ortho} \cdot \bar{c}_{ortho}}{1 + share(n)} + b_{ipin} + b_{sink} \right] \tag{4}$$

The expected wirelength cost $c_{exp,wld}(n)$ (Equation (4)) is the estimated downstream cost from the vertex n to the sink, controlled by the wirelength-driven weighting factor α that determines how aggressively the router explores towards the sink. It is based on the Manhattan distance from vertex n to the sink. The distance is split up into two parts: δ_{same} and δ_{ortho} for the distances in the same and orthogonal directions as the vertex n, respectively. The distance is multiplied with an average cost per distance in each direction ($\bar{c}_{same}$ and $\bar{c}_{ortho}$). The estimated cost is divided by the sharing factor to keep the expected cost heuristic admissible for an A* search in the router [24]. The small constants b_{ipin} and b_{sink} indicate the cost of the input pin and the sink, respectively.

The bias cost is introduced to help the router choose a good path from many equivalent shortest paths [25, 29], improving the negotiated sharing mechanism. It depends on the base cost $b(n)$ of vertex n and the Manhattan distance ($\delta_{m,c}$) from vertex n to the geometric center of the net N. It is shown in Equation (5), where N_{fo} and N_{HPWL} are the fanout and half perimeter wire-length of net N, respectively.

$$c_{bias}(b(n)) = \frac{b(n)}{2 \cdot N_{fo}} \cdot \frac{\delta_{m,c}}{N_{HPWL}} \tag{5}$$

It should be noted that the base cost $b(n)$ of vertex n, shown in Equation (3) is scaled to be proportional to its actual length $L(n)$ in CRoute [25, 29], for dealing with routing networks of the academic FPGA architecture model that has multiple routing resource types. The constant base cost coefficient b_0 is the same for all the wiring routing resources, meaning that the base cost is linear to the wirelength. This results in a reduced use of long routing resources, because the cost of using long resources is higher. However, an unnecessarily large cost for using long resources limits the timing optimization for commercial architectures, as it makes the router reluctant to use the available long routing resources.

For our router, we determine different base cost coefficients for different resource types. It should be noted that the vertical resources have smaller base costs than horizontal resources for the same resource type, as there are more vertical resources per INT tile and they have smaller delays. For the purpose of wirelength optimization, the wirelength $L(n)$ of n is included in the wirelength-driven cost $c_{wld}(n)$, controlled by the wirelength-driven weighting factor. The modified wirelength-driven cost $c'_{wld}(n)$ and base cost $b'(n)$ in our router are shown in Equation (6) and Equation (7), where $type(n)$ indicates the resource type of n.

$$c'_{wld}(n) = \frac{b'(n) \cdot p(n) \cdot h(n)}{1 + share(n)} + c_{bias}(b'(n)) + (1 - \alpha) \cdot \frac{L(n)}{1 + share(n)} \tag{6}$$

$$b'(n) = b_0(type(n)) \cdot L(n) \tag{7}$$

$$c'_{exp,wld}(n) = \alpha \cdot \frac{\delta_x + \delta_y}{1 + share(n)} \tag{8}$$

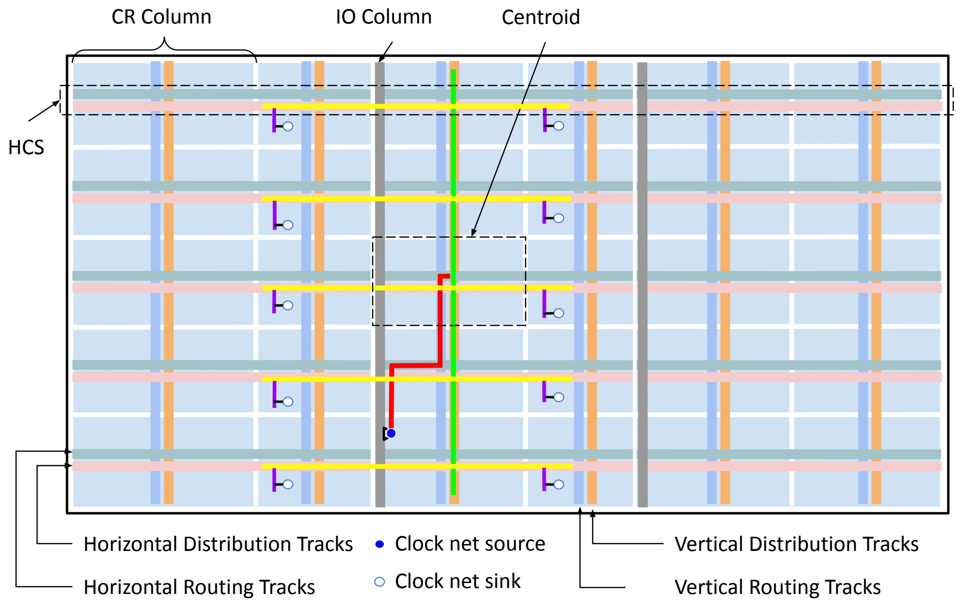
$$f'(n) = c'_{prev}(n) + c'_{wld}(n) + c'_{exp,wld}(n) \tag{9}$$

The b_{ipin} and b_{sink} are not expressed explicitly, because the input pin and the sink are not part of the routing expansion. We also determine that it is not necessary to rely on the direction of a routing resource to estimate the downstream wirelength cost to the target. Therefore, we simplify the expected cost shown in Equation (4) to use the Manhattan distance in horizontal δ_x and vertical δ_y directions from vertex n to the target. Equation (8) indicates the expected wirelength cost $c'_{exp,wld}(n)$ used in our router. Based on the modified costs, the total path cost used in our router is expressed in Equation (9). The $c'_{prev}(n)$ is the sum of $c'_{wld}(n)$ of all the vertices along the upstream path from n to the source.

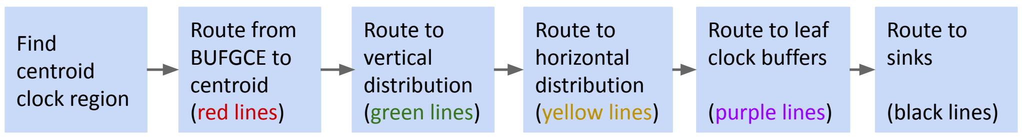
3.2 Global Signal Routing

Routing of global signals, such as the clock and static nets (VCC/GND), is essential for routers targeting real devices. The UltraScale+ device is subdivided into columns of clock regions (CRs), which is illustrated in Figure 4a. Each CR contains columns of CLBs, DSPs, BRAMs, and INT tiles. A clock net usually spans multiple CRs driven by a global clock buffer (e.g. BUFGCE). A horizontal clock spine (HCS) in the center of each CR runs through the device. The HCS contains the horizontal routing and distribution tracks as well as leaf clock buffers and clock network interconnects between horizontal/vertical routing and distribution tracks [8]. Vertical tracks of routing and distribution connect all CRs in a column.

The purpose of the clock routing is to route a clock signal from the global clock buffer to a central point from where it is connected to the loads via the distribution resources. Routing a clock net is done in a number of steps shown in Figure 4b. The clock net routing starts from identifying a centroid CR and routing from the source BUFGCE to the centroid CR. Once the route reaches the central point in the centroid CR, a transition is needed to route from the central routing track to vertical distribution tracks towards different target CRs. For each CR, the remaining routing is to route from the vertical distribution track to horizontal distribution tracks, followed by routing from the horizontal distribution tracks to leaf clock buffers. The last step of clock routing is to connect the leaf clock buffers to sinks.



(a) Clock architecture and routing example


Find centroid clock region
Route from BUFGCE to centroid (red lines)
Route to vertical distribution (green lines)
Route to horizontal distribution (yellow lines)
Route to leaf clock buffers (purple lines)
Route to sinks (black lines)

(b) Clock routing steps

Fig. 4. Clock architecture and routing process.

Other important global signals are static nets, which are different from the clock net with a unique source. These VCC/GND static nets have only input pins, i.e., sinks. Routing of VCC and GND means to connect their sinks to physical sources—hard wired points in the architecture that supply the necessary static polarity. Since multiple physical sources exist for static signals, these two nets can have many disjoint networks. These VCC/GND static nets are routed backwards one by one from each sink to find one of the potential sources. When routing of VCC/GND is completed, the utilized routing resources are preserved to ensure that there are no resource conflicts.

3.3 Wirelength-driven Routing Results

To evaluate RWRoute wirelength-driven routing, we used a number of designs, including 665 synthetic designs and 6 Rosetta benchmarks [28]. The benchmarks' attributes are shown in Table 1. Each design was placed on a Virtex UltraScale+ VU3P (xcvu3p-ffvc1517-2-e, whose maximum operating frequency is 775 MHz). The placement of all the benchmarks was constrained by using PBlocks (area constraints). The LUT utilization of synthetic designs is between 50%-88%. The placement was then preserved for both Vivado and RWRoute routing. In addition, RWRoute bounding box of connections ensures that the placement constraints are also reflected in the routing. To cover a wide range of

Table 1. An overview of the post-place benchmarks.

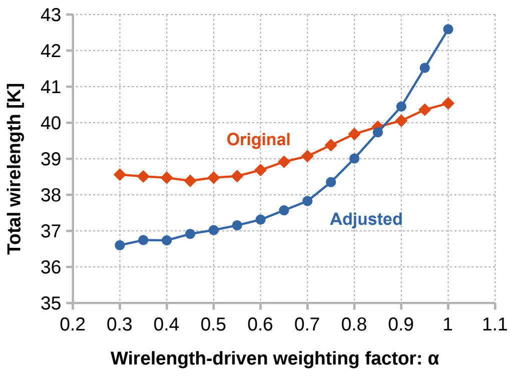
	LUTs	**FFs**	**DSPs**	**BRAMs**	**Total Nets**
benchmark	**[K]**	**[K]**	**[#]**	**[#]**	**[K]**
665 Synthetic designs	2~8	1~5	0	0	2~9
3d-rendering	2.3	1.8	13	39	4.8
BNN	3.1	2.8	4	29	5.9
Spam-filter	3.5	2.5	224	101	18.4
Face-detection	3.5	1.7	224	101	17.6
Optical-flow	28.6	25.8	124	69	58.9
Digit-recognition	24.8	6.4	1	336	42.5

practical design characteristics, we also varied the Rent's exponents of synthetic designs between 0.3 to 0.8 and logic depth from 3 to 7.

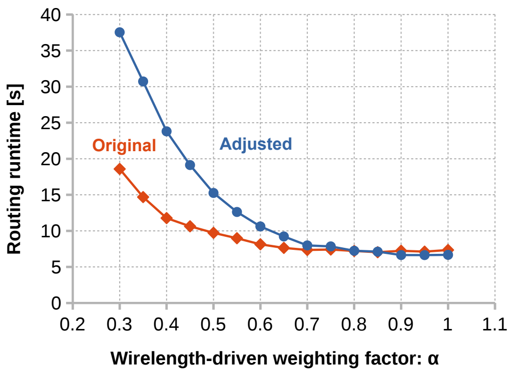
We first verify the efficacy of the modified cost function of RWRoute. Thirteen representative benchmarks of the synthetic designs were used to tune and compare RWRoute with different cost functions. These 13 representative designs have a Rent exponent of 0.7 with the number of LUTs ranging from 2K to 8K. The determined parameter was applied to the whole set of benchmarks in order to quantitatively evaluate the performance of RWRoute with Vivado as the baseline. The comparisons focus on runtime and the total wirelength of benchmarks. The wirelength of a routed design is defined as the total number of INT tiles that all the routing resources span. Resources that reside within an INT tile do not contribute to the total wirelength.

3.3.1 Efficacy of the Modified Cost Function. Figure 5 shows the overall wirelength and runtime of RWRoute with the original cost function $f(n)$ of CRoute, denoted as "Original" and with the adjusted cost function $f'(n)$, denoted as "Adjusted", for different wirelength weighting factors. With an increasing weighting factor α, both cost functions allow the router to expand more aggressively towards the targets, resulting in a shorter runtime at the expense of a longer wirelength. Using the modified costs with explicit wirelength in the cost function entitles the router to achieve 5% smaller wirelength at the expense of more runtime. The modified cost function yields a smaller wirelength than the original cost function when α is smaller than 0.85. When α reaches 0.75, the runtimes obtained from both cost functions converge. As a result, the adjusted cost function yields smaller wirelength while requiring similar runtimes when the weighting factor is between 0.75 and 0.85.

Figure 6 indicates the wirelength breakdown of RWRoute with the original cost function $f(n)$ and the adjusted cost function $f'(n)$, in terms of the total wirelength of all used long routing resources (i.e., L nodes) and other short routing resources (e.g., S/D/Q nodes). More than 90% of the total wirelength is attributed to short routing resources regardless of the cost function used, due to the fact that those 13 designs are small designs. The adjusted cost function encourages the usage of long routing resources. However, the original one discourages the router to use such resources because of their unnecessarily large base costs, which is aforementioned in section 3.1. In conclusion, the adjusted cost function of RWRoute is suitable for UltraScale+ devices, because it encourages using long routing resources to connect long-distance connections and thus results in a more efficient routing.

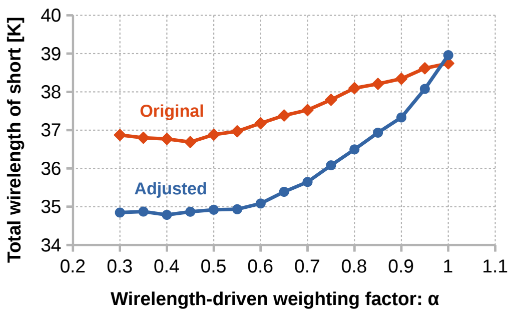


(a) Wirelength.

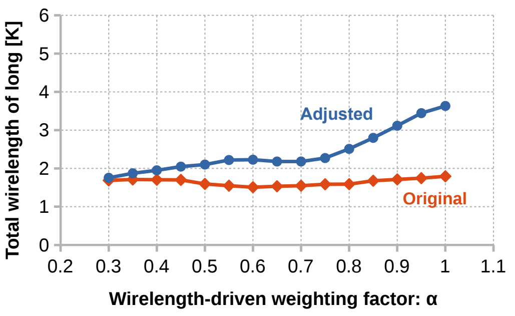


(b) Runtime.

Fig. 5. Wirelength and Runtime of RWRoute with different cost functions for different wirelength-driven weighting factors.



(a) Total wirelength of short resources.



(b) Total wirelength of long resources.

Fig. 6. Wirelength breakdown of RWRoute with different cost functions for different wirelength-driven weighting factors.

3.3.2 Comparison with Vivado. Figure 7 shows the empirical flow for the comparison between the RWRoute in its wirelength-driven mode and the Vivado 2020.2 router in its non-timing-driven mode. The Vivado router was set to run with a single thread, while other settings of it were left to default values. Vivado was running with a single thread because it is faster than running with multiple threads for these design sizes. Designs routed by Vivado were loaded into RapidWright for wirelength statistics using the same wirelength calculation utility as RWRoute. Wirelength statistics of designs routed by RWRoute were collected right after routing and stored in the routing data. The resulting post-route DCPs from RWRoute were loaded into Vivado for route status verification. RWRoute with the adjusted cost function was evaluated over all designs shown in Table 1. The wirelength weighting factor α of RWRoute is set to 0.8, which provides the minimum wirelength-runtime product.

All the designs are successfully routed by both Vivado and RWRoute. The total routing flow runtime (RT) includes the time for reading a design checkpoint (ReadDCP) and the time for routing the design (Routing). Figure 8 shows the geometric mean runtime and wirelength comparison between Vivado and RWRoute. Since RWRoute leverages RapidWright to open design checkpoints, it is able to load designs up to 13.2× faster. Because of its targeted scope

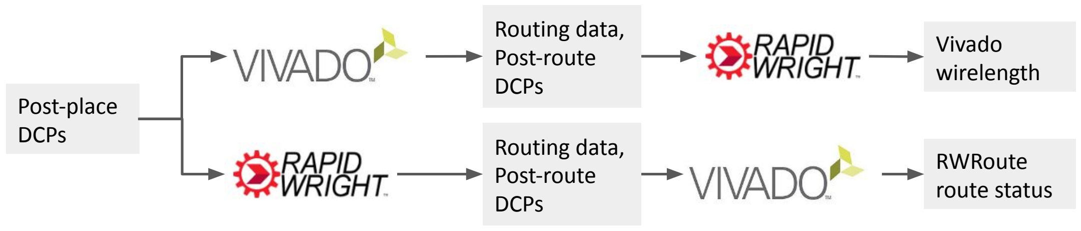
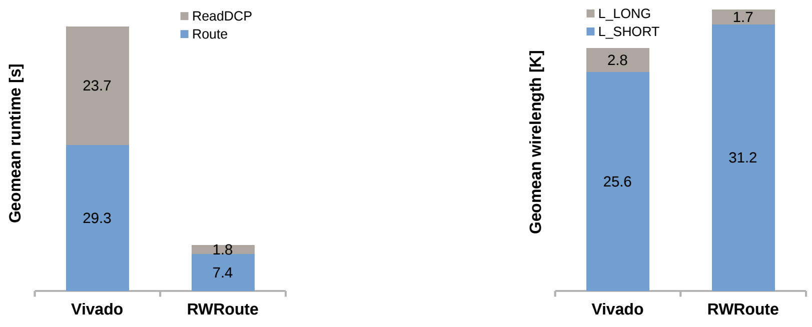


Fig. 7. Validation and comparison with Vivado.



(a) Runtime (b) Wirelength

Fig. 8. Comparison of RWRoute and Vivado for wirelength-driven routing.

of capabilities, RapidWright can safely avoid runtime-intensive interpretation of XDC constraints within the design checkpoint which Vivado must evaluate each time a design is loaded. This 13.2× reduction in design load time coupled with RWRoute's 4.9× improvement in routing runtime, resulting in a total speedup of 5.8× over Vivado. The total wirelength of RWRoute is 16% longer than Vivado, due to more resources used.

3.4 Partial Routing Using RWRoute

We explained in section 2.1 that compile time is one of the main obstacles for FPGAs to entry into computing era. A number of solutions such as split compilation or library-based approach are being explored to remedy this issue. These divide-and-conquer strategies divide the design into smaller modules such that each module can be implemented in parallel or offline. The complete design needs to be assembled where all nets within each module (intra-module nets) are already routed and only the nets between modules (inter-module nets) need to be routed. We refer to this routing use case as partial routing and RWRoute can support this mode.

The intra-module nets that were previously routed are preserved when inter-module nets are routed. As a result, routability is hindered by congestion and conflicts. A divide-and-conquer flow may mitigate this issue by ensuring that the modules have FFs at the edge. In effect, these techniques make the inter-module nets easy to be routed and less timing critical. As a result, wirelength-driven routing is generally enough for most inter-module nets routing.

We have evaluated our router in the partial routing mode on three designs from previous literature. Rendering is obtained through PRFlow that accelerates the FPGA compile time with the split compilation technique [26]. IntegerCoder is the design from Reference [23] that offers a massively parallel streaming model, and PicoBlaze is a processor array

Table 2. Comparison of RWRoute and Vivado for wirelength-driven partial routing, in terms of reading design checkpoint (ReadDCP), routing the design (Routing), total runtime (RT), Wirelength (WL) and critical data path delay (CPD).

				Vivado					RWRoute				
Design	**LUTs [#]**	**Total Nets [#]**	**Unrouted Nets [#]**	**ReadDCP [s]**	**Routing [s]**	**RT [s]**	**WL [K]**	**CPD [ns]**	**ReadDCP [s] (rel)**	**Routing [s] (rel)**	**RT [s] (rel)**	**WL [K] (rel)**	**CPD [ns] (rel)**
Rendering	28386	25288	10306	36	50	86	293	4.151	6 (0.17)	13 (0.26)	19 (0.22)	290 (0.99)	4.151 (1.00)
PicoBlaze	45540	88575	12144	51	104	155	583	2.545	7 (0.14)	16 (0.15)	23 (0.15)	577 (0.99)	2.545 (1.00)
IntegerCoder	145763	192519	8280	71	269	340	1863	7.117	14 (0.20)	9 (0.03)	23 (0.07)	1863 (1.00)	7.117 (1.00)
Geomean	**57330**	**75550**	**10120**	**51**	**112**	**165**	**683**	**4.221**	**8 (0.17)**	**12 (0.11)**	**22 (0.13)**	**678 (0.99)**	**4.221 (1.00)**

design available from RapidWright tutorials. In each test case, the nets between computing kernels are not routed. All the nets within each kernel were routed and will be preserved. Table 2 shows the runtime and QoR comparison between RWRoute and Vivado for partial routing. On average, RWRoute is 6.4× faster in reading design checkpoints and 9.3× faster in routing designs, resulting in a total runtime speedup of 7.5× over Vivado. The total wirelength of RWRoute is slightly better than Vivado. For each of the three designs, their critical paths do not involve inter-module nets. Thus, both RWRoute and Vivado yield the same critical path delay.

4 LIGHTWEIGHT OPEN-SOURCE TIMING MODEL

In order to extend the wirelength-driven router to a timing-driven router, we need to introduce delay estimates in the cost function to guide the router for delay optimization. Therefore, an accurate enough timing model for the architecture is required. We present the proposed equation-based lightweight timing model in this section. The model contains both the delay of logic elements and routing resources. The logic delays of LUT, FF, CARRY, DSP, BRAM and URAM are included in the model. The model stores delays covering all the configurations of each logic element. However, a DSP can be used in many configurations. Therefore, its delay is generated by Vivado in a case by case basis. The script is provided with the release. As more configurations are generated and archived, more designs can use the stored data without relying on the script. Because the logic delays can be straightforwardly extracted from Vivado, they are omitted from this paper.

Segmentation routing and a columnar floorplan are first order architectural factors that affect the delay of the nets in a design. However, silicon process technology miniaturization has increased the relative impact of wire delay on total delay and hence the influence of implementation and layout. Examples include nonuniform 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 bound by the same guarantee and is free to explore implementation trade-offs 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.

4.1 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 *Node Group* (NG), which is a lumped set of inter-site switches

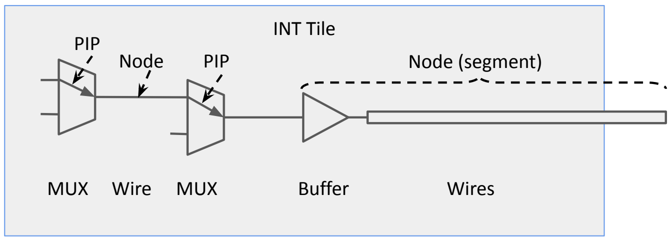


Fig. 9. Node Group example.

and nodes to minimize memory requirements. NGs represent distinct decision points in the inter-site routing graph, and finer timing granularity is unnecessary in many cases.

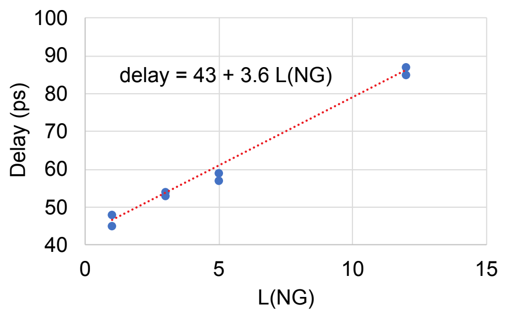
Figure 9 shows an example of a NG that represents a routing segment and its corresponding switch multiplexers and driving buffer. A MUX is used to provide routing flexibility to the segment. Connections between a MUX input and output are called programmable interconnect points (PIPs). A connection between MUXes is called a node. A node may contain several electrically connected wires, depending on the tiles it spans. Based on the ending node and the direction of the ending node, NGs are classified into different types in horizontal and vertical directions, such as BOUNCE, SINGLE, DOUBLE, QUAD, LONG in both directions and GLOBAL in the horizontal direction. The upstream PIPs, buffer, and the segment starting point of a NG reside inside an INT tile. Equation 10 calculates the delay of a NG,

$$delay(NG) = k_0 + k_1 \cdot L(NG) + k_2 \cdot d(NG) \tag{10}$$

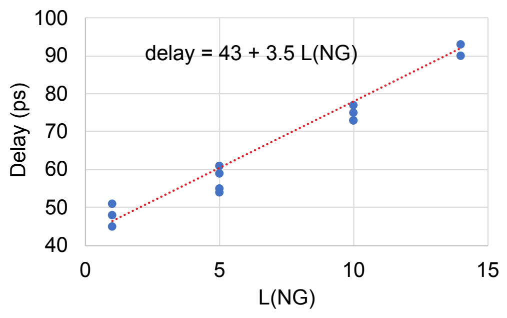
where $L(NG)$ represents the length of the wire segments, and $d(NG)$ captures the additional distance a NG travels beyond the typical logic resource stride in the architecture. $L(NG)$ is defined for each NG type, but $d(NG)$ is the sum of d for all the tiles that nodes of the NG pass through. The constant k_0 is the intrinsic delay and k_1 is the coefficient constant corresponding to the segmented routing architecture. The coefficient constant k_2 is needed to adjust the delay for columnar architecture discontinuities as explained in section 2.2. These three constants, which capture the first order architectural factors in the delay model, will be obtained through curve fitting.

Figure 10a shows the results of fitting the delay for vertical NGs. There are two data points, representing vertical NGs of each resource type. L and d are independent variables that can be selected during curve fitting. We often start with the nominal length of a NG (i.e. a DOUBLE NG spans two INT 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 10b shows a similar linear approximation for horizontal resources. Keep in mind that both Figures 10a and 10b 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 [3, 4, 13]. The extent of delay increase can be different for different types of NGs even when they cross the same hard block. Therefore, d in Equation 10 is a function of NGs. Delay adjustment of NGs is obtained by subtracting the value of the first two terms in the equation



(a) Vertical node groups.



(b) Horizontal node groups.

Fig. 10. Curve fitting for k_1 of vertical and horizontal node groups.

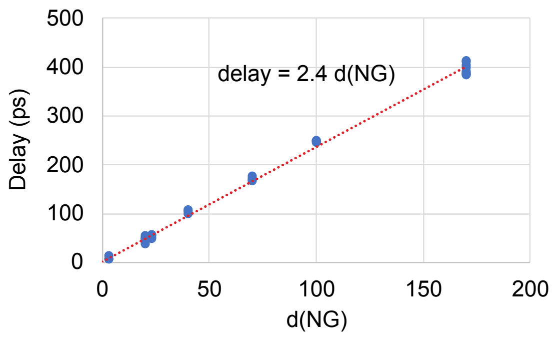


Fig. 11. Curve fitting for k_2 of horizontal QUAD NGs

from the net delay. Figure 11 shows an example plot for the resulting delays of horizontal QUAD NGs versus $d(NG)$. The horizontal axis $d(NG)$ does not directly correspond with the physical width of the columns. We can indirectly adjust $d(NG)$ by adjusting d of hard block tiles the NG pass through (see Table 4). QUAD NGs can cross many types of hard block columns, but we could still obtain a highly correlated fit as seen in Figure 11. 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 third term of vertical NGs is similar, but simpler because only the RCLK tile has a non-zero $d(NG)$ that requires adjustment.

4.2 Segmentation and Discontinuity Parameters

We adopt a successive difference approach to determine the delays of NGs. 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 NGs from the net delay reported by Vivado. Next, the delay of SINGLE and DOUBLE NGs will be determined by placing source and sink at appropriate distances. The delay of SINGLE and DOUBLE NGs is found by subtracting an appropriate input site pin delay from the obtained net delay. The delay of QUAD and LONG NGs can be derived in a similar way.

Table 3. Segmentation parameters for delay model.

Coefficient	Hor.	Ver.
k_0	43	43
k_1	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 4. Discontinuity parameters for delay model.

k_2 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	SINGLE / DOUBLE	QUAD	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

Table 3 summarizes the constants for all the NGs in the interconnect that correspond to the first and second term of Equation 10. As expected, k_0 and k_1 of horizontal and vertical NGs are similar as the architecture is designed to be symmetric. It is important to note that the independent variable L does not always correspond to the length of a target NG. Fitting the delay to Equation 10 yields coefficients as shown in Table 3. It should be noted that a GLOBAL NG has no physical L associated with it and its length to calculate the delay is merely virtual.

Table 4 summarizes the constant k_2 and corresponding d for NGs that need adjustment to cross over architectural discontinuities. In the vertical direction, there is only one discontinuity regarding RCLK rows. A RCLK row is in the middle of each clock region, dedicated to some clocking resources. All other hard blocks listed in the table are for the discontinuity in the horizontal dimension. To find d of a horizontal NG, we only need to compute and store d values for one row because the UltraScale+ architecture is columnar. The cumulative distances are computed 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 NG. Due to the fact that NGs straddle between INT tiles, we need to store the cumulative d only on INT tiles. As a result, the size of a d array of a device architecture 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 a 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 NGs in our model. This implies a more than two orders of magnitude reduction in the timing model size for each tile. Moreover, we extend the concept of equivalent NGs to all the device tiles to reduce the memory footprint even further. Such equivalency is based on the same type of the ending node the NGs contain and is determined by the ending node direction, length, and the hard block columns on its path. Our proposed model will provide one delay value for each equivalent NG, which results in an implementation consuming less than one kilobyte of memory.

4.3 Corner Cases and Exceptions

The equation with parameters explained previously captures the 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.

4.3.1 Pessimistic Delay Approach. When there are multiple delays for similar NGs 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.

4.3.2 Masking Resources Approach. When an exceptional case occurs, we disallow the use of such resources without impacting routability in RWRoute. For example, horizontal long NGs on rows adjacent to a RCLK tile may have large delay differences. Such rare cases constitute roughly 0.2% of all long NGs ending at a given column. Therefore, we can easily mask those NGs from routing resources in RWRoute 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.

4.4 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.

4.4.1 Estimating Point-to-point Net Delay. In this evaluation, we randomly placed the source and sink of a two-terminal 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 12 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 5, where a positive error value indicates that the model is pessimistic. On average, the model is pessimistic by 0.7%.

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 [19]. 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 value, computed over the data shown in Figure 12, is 0.99, where the value of 1.0 indicates perfect fidelity [9].

Table 5. 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

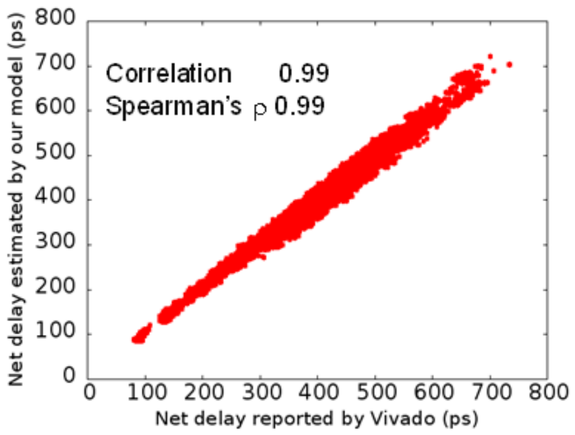


Fig. 12. Correlation between our model net delay estimates and Vivado.

Table 6. 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

4.4.2 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 used the synthetic designs as mentioned in section 3.3. Each design was routed with both 2 ns (500 MHz) and 1.3 ns (769 MHz, near the device spec) target clock period. The setup is to measure the critical path delay 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. 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 6. Among all 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 net 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, a 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.

5 TIMING-DRIVEN ROUTING FOR COMMERCIAL ARCHITECTURES

In this section, we present our timing-driven routing tool for commercial architectures, based on the open timing model presented in the previous section.

5.1 Timing-driven Signal Routing

The timing-driven general signal routing of RWRoute follows the same connection-based approach as that of the wirelength-driven routing presented in section 3.1. In addition to wirelength reduction and congestion removal, it also focuses on optimizing the critical path delay of a design, which impacts the maximum frequency of the circuit. A static timing analysis task is performed in each routing iteration to compute and update the delay. The total path cost function used during the expansion is extended from the wirelength-driven version shown in Equation (9) to be delay-aware, which is expressed in Equation (11). In addition to the wirelength-driven costs $c'_{wld}(n)$ (Equation (6)) and $c'_{exp,wld}(n)$ (Equation (8)), the total path cost $f_{td}(n)$ includes the delay cost $T_{del}(n)$ of the routing resource n and the estimated remaining delay cost $c_{exp,del}(n)$ to the sink.

$$f_{td}(n) = c_{prev,td}(n) + c_{td}(n) + c_{exp,td}(n) \tag{11}$$

$$c_{td}(n) = (1 - f_{criti,con}) \cdot c'_{wld}(n) + f_{criti,con} \cdot (1-\beta) \cdot T_{del}(n) \tag{12}$$

$$c_{exp,td}(n) = (1 - f_{criti,con}) \cdot c'_{exp,wld}(n) + f_{criti,con} \cdot \beta \cdot c_{exp,del}(n) \tag{13}$$

The delay cost $T_{del}(n)$ of n is computed according to the timing model equation, while the estimated delay cost $c_{exp,del}$ is based on the remaining distance to the target sink (Equation (14)). The distance is split into the horizontal (δ_x) and vertical (δ_y) parts. The two estimated delay constant factors $T_{del,x}$ and $T_{del,y}$ are the average horizontal and vertical delay per unit distance of an INT tile, respectively. Each of them is computed according to the timing model equation coefficients from Table 3, with SINGLE, DOUBLE, QUAD, and LONG NGs taken into account.

The timing-driven weighting factor β is to enable a better trade-off between runtime and critical path delay, which is demonstrated in section 5.3.1. Different from CRoute, we assign a weighting factor $(1-\beta)$ to T_{del}(n), in order to balance the impact of the real delay $T_{del}(n)$ and the estimated delay $c_{exp,del}$ on the overall timing-driven cost.

$$c_{exp,del} = T_{del,x} \cdot \delta_x + T_{del,y} \cdot \delta_y \tag{14}$$

In order to determine if a connection should be routed for delay or wirelength optimization, a criticality factor is assigned to each connection based on the static timing analysis at the beginning of each routing iteration. The relative importance of the wirelength-driven and the timing-driven cost is determined by the criticality of the connection. The criticality of a connection is computed as follows:

$$f_{crit,con} = min\left[\left(1 - \frac{slack_{con}}{D_{max}}\right)^{\phi}, f_{crit,max}\right] \tag{15}$$

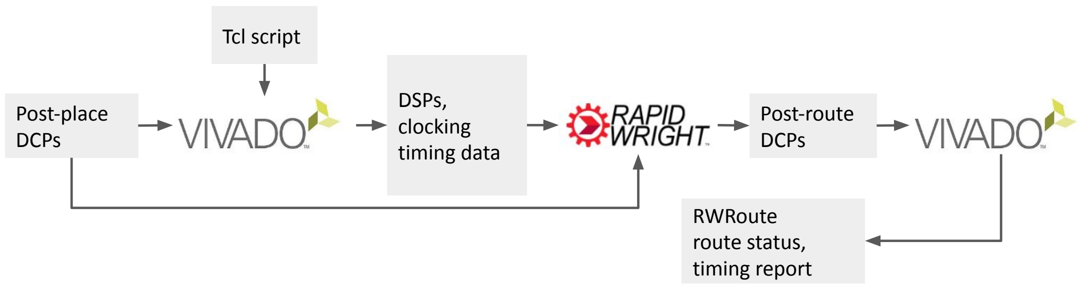


Fig. 13. Timing-driven routing and validation flow for designs with DSPs and clocks.

where D_{max} is the maximum arrival time and $slack_{con}$ is the slack of the connection based on its delay, the arrival time of its source and the required time of its sink. The criticality exponent ϕ, always greater than 1, is used to spread non-critical connections and critical connections apart. In this way, only most critical connections are routed with the maximum efforts in delay optimization. The criticality of a connection is capped at $f_{crit,max}$. The criticality exponent ϕ and the $f_{crit,max}$ are set to 3 and 0.99 respectively, according to CRoute [25, 29].

As aforementioned in section 3, the sharing factor is introduced to the wirelength-driven costs to drive the resource sharing among connections within the same net. It is defined as the number of connections that are routed through the routing resource n, which is clueless about a connection's criticality. Aggressively encouraging resource sharing can harm the delay optimization of timing-driven routing, especially for large designs. Therefore, the original sharing factor of CRoute has been updated with each connection's criticality, resulting in $share_{criti}(n)$ shown in Equation 16.

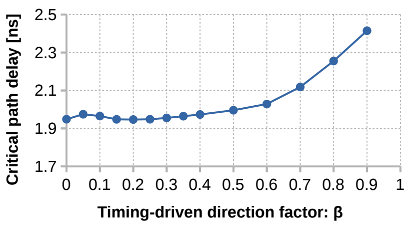
$$share_{criti}(n) = (1 - f_{criti,con})^{\gamma} \cdot share(n) \tag{16}$$

The customizable sharing exponent γ empowers the router to be aware of each connection's criticality when facilitating resource sharing of subsequent connections of a net. The new sharing factor $share_{criti}(n)$ of a critical connection should be smaller than that of a non-critical connection so that the sharing mechanism focuses less on the resource sharing for the critical connection. Therefore, γ should be greater than 0. This modification has no impact on the wirelength-driven routing, as the criticality of each connection is 0 in the wirelength-driven mode.

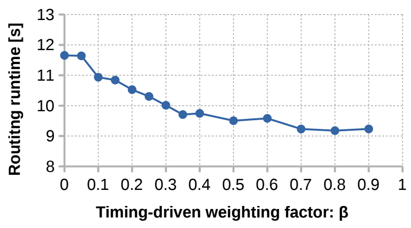
5.2 Global Clock Routing

Although the clock routing seems straightforward by following the steps in Figure 4, effectively routing clock signals in timing-driven routing requires detailed delay information. The detailed delay information is only available in Vivado. It is virtually impossible to extract the delays of clock routing resources in a similar way to section 4. There are two main reasons: (i) the clock network has limited observability, i.e., its strict hierarchical structure provides little flexibility to setup a test circuit, and (ii) the network is not composable, i.e., a delay of a clock routing resource depends on its load. As a result, we leverage Vivado to build a critical part of the clock network.

A sample design is built to cover a set of CRs and is later to be routed by Vivado. The information of the routed clock network from the center point to the last horizontal distribution track of each CR is stored as a template. The template can be used for any design whose footprint is within the set of CRs. A tcl script is provided with the open source release that can be modified for other templates. The Tcl script is used to build the clock routing template, as illustrated in Figure 13. The partial routing tree from the central point to different horizontal distributions is imported



(a) Critical path delay.



(b) Routing runtime.

Fig. 14. Critical path delay and routing runtime of RWRoute with different timing-driven weighting factors.

from the template into the clock routing, constituting green and yellow routes shown in Figure 4a. The dependency on the script reduces as more templates are accumulated.

Timing-driven clock routing needs to complete the route from the global clock buffer (BUFGCE) to the central point (red routes shown in Figure 4a) and the route from the horizontal distributions to the sinks following the last two steps shown in Figure 4. This routing strategy is also used to route a high fanout net (e.g. a clock enable or reset net) using the clock routing network. As for the global static signal routing, VCC and GND nets do not have timing requirements, so they are routed in the wirelength-driven mode, following the same approach as mentioned in section 3.2.

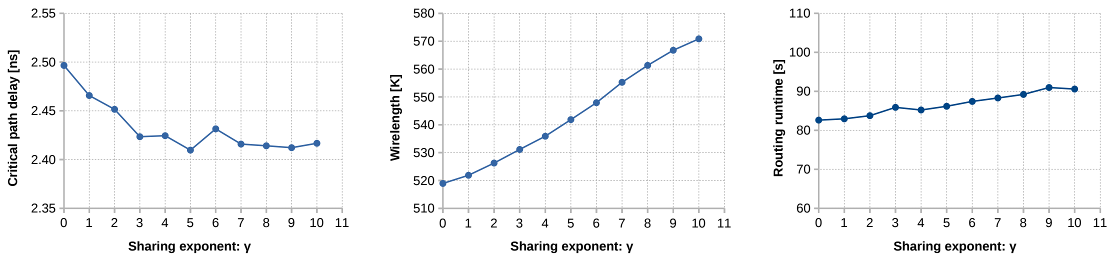
5.3 Timing-driven Routing Results

We first present the effect of customizing the routing parameters of RWRoute that are available for users to control the router performance, followed by its timing accuracy evaluation. Next, we introduce the critical path delay pessimism factors to guarantee the timing closure. The last part of the experimental study focuses on the comparison between RWRoute and Vivado in terms of the runtime and QoR.

5.3.1 Customizable Routing Parameters. A major benefit of an open-source routing tool is that it offers users the flexibility to control the router performance, such as the runtime and QoR trade-off, through customizable routing parameters. We have presented how the wirelength-driven weighting factor α impacts the routing runtime and achievable total wirelength in section 3. The runtime and wirelength of RWRoute timing-driven routing shows the same trend as illustrated in Figure 5, with different values of α. The value of α has insignificant influence on the critical path delay, as it varies within 2% when α increases from 0.3 to 1. We set α to 0.8 for RWRoute in the timing-driven mode to enable aggressive search towards connection targets for fast runtime. We will hereafter focus on the two timing-related parameters, i.e., the timing-driven weighting factor β and the sharing exponent γ.

Figure 14 shows the critical path delay and runtime plots with different timing-driven weighting factor values with α set to 0.8. When β increases from 0 to 0.9, the critical path delay gets a 24% increase, while the runtime reduces by 23%. When β is 0, there is no estimated timing cost included in the total cost function. RWRoute relies on the exact delay from the source to the current resource under consideration for timing optimization. While that certainly yields the smallest critical path delay, the router is oblivious to the more promising path and massively explores the accessible routing resources, resulting in the largest runtime.

With a greater β, RWRoute focuses more on optimizing the estimated delay, i.e., pruning away less promising resources, and hence the runtime decreases. The critical path delay remains virtually the same when β increases from 0



(a) Critical path delay. (b) Wirelength. (c) Routing runtime.

Fig. 15. Critical path delay and wirelength of RWRoute with different sharing exponent values.

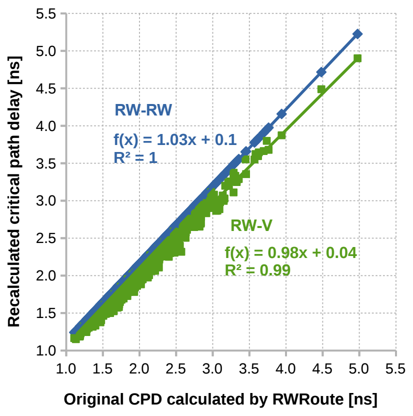
to 0.35, while the runtime decreases. Therefore, the value of β is set to 0.35 in our later experiments. If a more accurate estimate could be obtained with small runtime overhead, the runtime could be further reduced with small critical path delay degradation.

The original sharing factor has negative effects on long and critical nets. Figure 15 shows the impact of the introduced sharing exponent on the critical path delay, total wirelength, and runtime of the router. When γ is 0, the sharing factor is unaware of the criticality of a connection. Consequently, it drives the router to share resources even when the connection is long and timing critical. Such resource sharing generally causes long detours of connections, resulting in a large critical path delay. With the increasing sharing exponent, the router is discouraged to share resources for very critical connections, allowing a suitable route for the critical connection. Therefore, the critical path delay becomes smaller, at the expense of wirelength increase. Usually, critical path delay is the dominating metric for practical use cases as long as the design is routable, while the total wirelength is not a constraint. The runtime is not sensitive to the sharing exponent. As shown in Figure 15c, it is only increased by 10% when the sharing exponent increases from 0 to 10. The sharing exponent of RWRoute is set to 2 for an optimal trade-off between the wirelength-runtime product increase and critical path delay reduction.

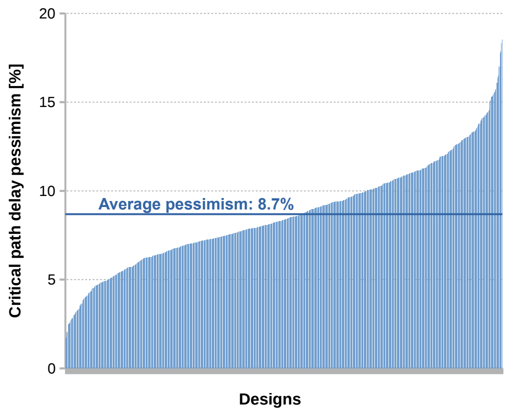
5.3.2 Timing Accuracy and Timing Closure Guarantee. The open timing model used has been validated by Vivado as explained in section 4.4.2. We evaluate the collective effects of the timing model accuracy and the router in order to provide a timing closure guarantee without relying on Vivado verification. Since the ultimate guarantee can only be provided with accurate timing data from commercial vendors, we adopted a strategy to be pessimistic on reporting the critical path delay of designs. If the timing requirements are met in RWRoute, they are highly likely to be met in Vivado as well.

The green scatter correlation plot in Figure 16a shows critical path delays of designs routed by RWRoute and timed by Vivado (RW-V) against those originally reported by RWRoute. The plot visually shows high correlation (0.98) between the two measures, which is consistent with that of Figure 12. Similar to the optimistic cases that Table 6 indicates, for some designs, their delays reported by RWRoute are optimistic, i.e., RWRoute reports smaller delays than Vivado for them.

In some custom flows, relying on RapidWright (instead of Vivado) for timing requirements helps with compile time. Providing a timing guarantee at the expense of some critical path delay is acceptable for such flows. In order to guarantee that there is no timing violations in Vivado when RWRoute reports the timing target is met for a design, we introduced critical path delay pessimism factors to the RWRoute timing analysis. By applying linear regression to the



(a) Recalculated critical path delay.



(b) Critical path delay pessimism.

Fig. 16. Critical path delay of RWRoute.

Table 7. Comparison of RWRoute and Vivado for timing-driven routing of all benchmarks, in terms of reading design checkpoint (ReadDCP), routing the design (Routing), total runtime (RT), total wirelength (WL) and critical data path delay (CPD).

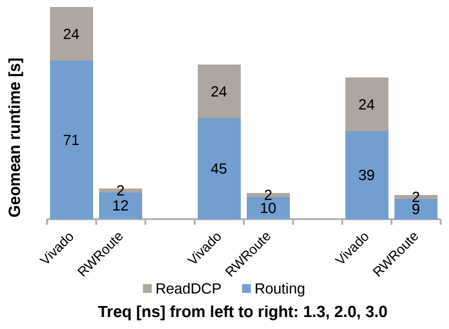
Router	ReadDCP [s]	Routing [s]	RT [s]	WL [K]	CPD [ns]
Vivado	23.7	50.0	73.7	31	1.80
RWRoute	1.8	10.3	12.1	34	1.98
Ratio	**0.07**	**0.21**	**0.16**	**1.10**	**1.10**

RW-V data points, we obtained a linear equation of RW-V, as illustrated by the green line in Figure 16a. The final delay reported by RWRoute timing analysis (RW-RW) is adjusted to scale following the blue trend line, which has a slightly greater slope (1.03) than that of RW-V (0.98) and a constant delay factor of 100 ps. As a result, RWRoute timing analysis is pessimistic in calculating the critical path delays of designs, as shown in Figure 16b. On average, the critical path delay reported by RWRoute is 8.7% greater than that of the Vivado timing analysis tool.

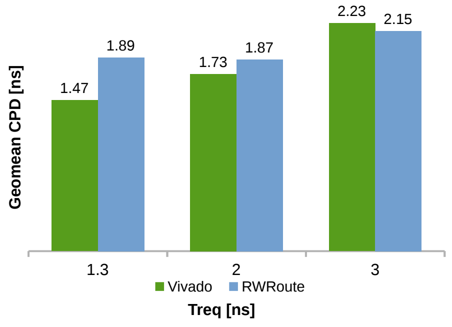
5.3.3 *Comparison with Vivado.*

Table 7 shows the comparison between RWRoute and Vivado for timing-driven routing with results averaged (geometric mean) across all the benchmarks shown in Table 1. The critical path delay of Vivado and RWRoute was obtained through the Vivado timing analysis tool. RWRoute is 4.9× faster in routing designs and 13.2× faster in loading design checkpoints, resulting in an overall speedup of 6.1×. It achieves 10% larger critical path delay and 10% larger total wirelength than Vivado. We further categorize the synthetic designs placed by Vivado with different timing targets into three groups according to the timing requirement for a more detailed comparative analysis, followed by a comparison regarding routing Rosetta benchmarks and RWRoute speedup analysis.

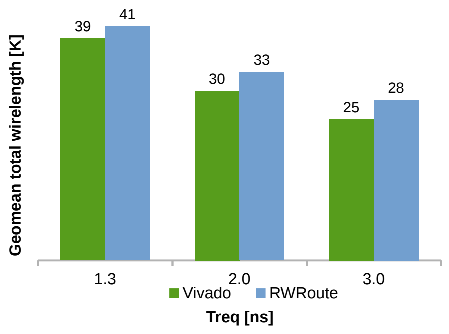
Figure 17 shows the comparison between RWRoute and Vivado in terms of runtime, critical path delay, total wirelength, and the percentage of routed designs that have timing closure for different timing targets. The time spent in reading DCPs of both Vivado and RWRoute remains the same for different timing targets, as it mainly depends



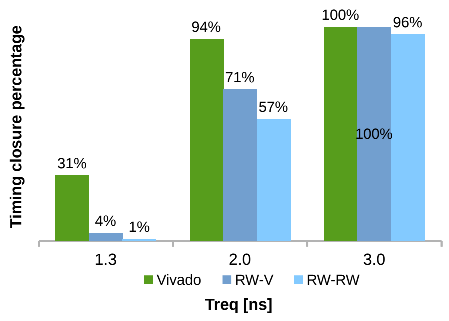
(a) Runtime.



(b) Critical path delay.



(c) Total wirelength.



(d) Percentage of routed designs with timing met.

Fig. 17. Timing-driven routing comparison of RWRoute and Vivado with different timing targets.

on the design size. With more relaxed timing constraints, the routing runtime and total wirelength of both RWRoute and Vivado get smaller. RWRoute maintains a 4.6× or larger routing runtime speedup for every timing requirement. RWRoute consistently uses larger total wirelength and the gap increases as the target frequencies decrease.

As for the timing-related comparison, we include the results obtained from the two different measurements for RWRoute routing results, i.e., the one timed by Vivado (RW-V) and the self-measured one with critical path delay pessimism factors (RW-RW). Figure 17b and Figure 17d verify the critical path delay guarantee of RWRoute. As the RW-RW is always greater than RW-V, the percentage of routed designs with timing met is less when relying on RW-RW than that of relying on RW-V. When aiming at the near-spec timing target (1.3 ns), RWRoute obtains a 29% lower maximum frequency. Consequently, only few designs have timing closure at such a high frequency. Relaxing the timing constraint leads to a greater number of designs that achieve timing closure for both RWRoute and Vivado. When the timing target is 3.0 ns, RWRoute is able to achieve 3.6% smaller critical path delay than Vivado and same percentage of designs that have timing closure, based on the timing analysis tool of Vivado. Even with the timing guarantee pessimism approach, RWRoute can close timing for up to 96% of the designs.

We further evaluate RWRoute scalability with additional 50 larger synthetics designs that have up to 110K LUTs with a Rent exponent of 0.7 and logic depth of 3 targeting the timing requirement of 3.0 ns. Figure 18 shows the routing runtime scalability of RWRoute and Vivado. RWRoute remains faster than Vivado for routing the larger synthetic designs. When the number of nets reaches 110K, the routing runtime reduction of RWRoute over Vivado is 20%. The

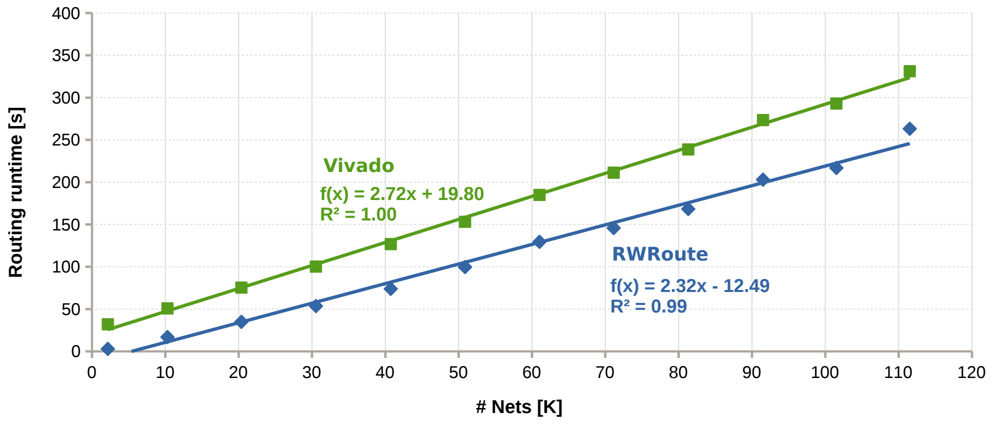


Fig. 18. Routing runtime scalability of RWRoute and Vivado.

Table 8. Comparison of RWRoute and Vivado with Rosetta benchmarks.

	Vivado					RapidWright				
Design	ReadDCP [s]	Routing [s]	RT [s]	WL [K]	CPD [ns]	ReadDCP [s] (rel)	Routing [s] (rel)	RT [s] (rel)	WL [K] (rel)	CPD [ns] (rel)
3d-rendering	24	42	66	30	3.566	2 (0.08)	9 (0.21)	11 (0.17)	29 (0.97)	3.377 (0.95)
BNN	24	43	67	39	3.357	2 (0.08)	12 (0.28)	14 (0.21)	40 (1.03)	3.432 (1.02)
Face-detection	29	92	121	159	4.315	4 (0.14)	37 (0.40)	41 (0.34)	158 (0.99)	4.264 (0.99)
Spam-filter	29	88	117	167	3.735	4 (0.14)	39 (0.44)	43 (0.37)	166 (0.99)	3.980 (1.07)
Digit-recognition	29	139	168	406	5.619	6 (0.21)	131 (0.94)	137 (0.82)	434 (1.07)	5.730 (1.02)
Optical-flow	32	245	277	407	4.045	6 (0.19)	141 (0.58)	147 (0.53)	447 (1.10)	3.722 (0.92)
Geomean	**28**	**89**	**119**	**131**	**4.046**	**4 (0.13)**	**38 (0.42)**	**42 (0.35)**	**134 (1.02)**	**4.016 (0.99)**

speedup of RWRoute over Vivado gets larger as the number of nets reduces. Both RWRoute and Vivado achieve larger critical path delay with the increasing number of nets. However, RWRoute timing performance deteriorates faster than Vivado. When there are 110K nets, the designs routed by RWRoute no longer meet the timing requirement.

Table 8 shows the results of RWRoute and Vivado for each Rosetta benchmark. Overall, RWRoute obtains roughly the same QoR as Vivado and its speedup varies from 1.1× to 4.7×. It is also observed that there is a speedup gap when comparing the speedups of RWRoute over Vivado for Rosetta benchmarks and synthetic designs. For instance, RWRoute has little speedup gain for the Digit-recognition design with 42.5K nets (Table 1), at which number of nets RWRoute achieves a speedup of 1.5× over Vivado seen from Figure 18.

In order to reveal the factors that impact the speedup of RWRoute, we introduced a new measure, called average connection span (ACS). It is the average span of all the connections in a design. The span of a connection is defined as the Manhattan distance between its source and sink, in the unit of the number of INT tiles. Figure 19 shows the speedup of RWRoute v.s. ACS of different designs targeting the timing requirement of 3.0 ns. It is shown that the RWRoute speedup over Vivado decreases when ACS of designs increases. The longer ACS of the three Rosetta benchmarks results in the fact that RWRoute achieves smaller speedups for them than synthetic designs of a similar size. Their longer ACS is because they use hundreds of hard blocks leading to larger placement footprint. Variations of RWRoute speedup for a

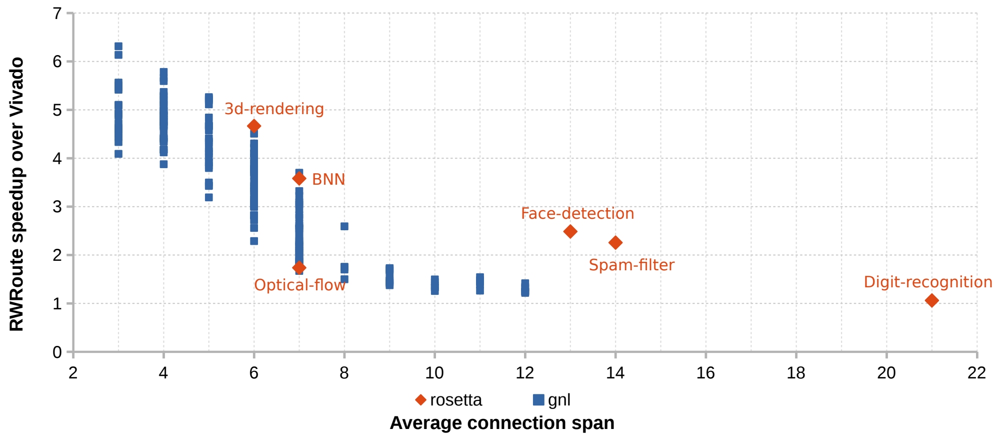


Fig. 19. RWRoute speedup over Vivado in terms of average connection span of designs.

given ACS is mostly due to design sizes. The speedup tends to be higher for smaller designs due to high fix startup runtime of Vivado. Therefore, the runtime speedup of RWRoute over Vivado depends on both the size and average connection span of a design.

The above experimental investigations emphasize a key benefit of RWRoute's partial routing capability, as explained in section 3.4. In a low touch flow with partial routing of large designs, it is reasonable to expect less than 100K unrouted inter-module (or library) nets to be completed by RWRoute. This will be within the range of the net count that RWRoute performs better than Vivado in terms of compile time.

6 RELATED WORK

Open-source routers have been under active investigation for over twenty years. Until recently, research in this area has mainly targeted hypothetical architectures, which tend to be much simpler than commercial architectures. VPR [1] has been a mainstay in this category since its inception, providing an important platform for researchers to investigate their hypotheses on FPGA architectures and CAD algorithms. Extensive work has been facilitated by VPR, including CRoute. While these efforts are important in FPGA research and advancement, their effectiveness has been challenging to quantify and directly translate for the use of commercial architectures due to the gap in architectural detail and complexity. For example, the UltraScale+ architecture has an enhanced and flexible clocking routing architecture that must be approached in a proper sequence in order to achieve a successful route. Previous academic architecture tools have not had the fidelity to capture this detail which is essential for targeting a commercial architecture. RWRoute has successfully demonstrated the adaptations needed when applying these techniques to commercial architectures to achieve a practical and usable flow.

There has been another class of open-source projects targeting commercial FPGAs, such as Torc [21], RapidSmith [5, 12], VTR-to-Bitstream [6], Maverick [2], Yosys+nextpnr [20], and SymbiFlow [17]. While these projects provide important platforms for CAD tool research targeting commercial FPGAs, most of them are in the unfortunate state of containing generic proof-of-concept routing algorithms (e.g., conflict unaware and/or non-timing driven). As a result, they are not adopted for real use cases. A notable exception is the tool nextpnr which supports timing-driven place and route on several Lattice and Gowin FPGAs with current development expanding to several other architectures [27].

Another exception is SymbiFlow which allows a complete open source implementation flow for a Xilinx Artix 7 commercial device with enhancements to VPR and a data-driven bitstream generator [17]. The entire flow supporting timing-driven routing for the target commercial device shows faster routing runtime with the penalty of 2.3× longer critical paths than Vivado 2017.2. Such high QoR penalty is often not acceptable for practical applications. In our work, RWRoute was shown to run 4.9× faster than the Vivado router for a number of placed designs, while maintaining competitive QoR.

Another related work is Reference [13], which extracts approximate timing information of routing resources of UltraScale FPGAs using the RapidWright framework. That work improves timing-aware routing of RapidRoute [14] that is specialized in routing communication overlays. The model was built based on a single INT tile and used for every INT tile across the device. This timing approximation lends itself well to the domain specific nature of RapidRoute. However for general use cases that take advantage of heterogeneous portions of a device, its effectiveness is limited. For example, Figure 11 shows that the delays of QUAD node groups vary significantly depending on where it is in the device. As a result, generalizing the timing information of one INT tile to the entire FPGA cannot provide accurate enough timing estimates for all scenarios. RWRoute in contrast provides a more comprehensive timing model and is better suited for a variety of routing scenarios beyond specific bus communication structures.

7 CONCLUDING REMARKS

In this work, we introduced RWRoute, the first open-source timing-driven router for UltraScale+ FPGAs. We demonstrated the possibility of open-source FPGA tools delivering QoR in the ball park of commercial tools. RWRoute includes global signal routing features such as clock and static signal routing that are often missing from earlier open-source tools, and hence preventing their usage for the existing commercial devices. A key benefit of the open-source router is to provide complete flexibility of internal cost function customization. This enables users to augment the router towards a specific domain of interest or their desired figure of merit. For example, our results show that 4.9× compile time improvement is possible at the expense of 10% loss in QoR. Additionally, through its partial routing capabilities, RWRoute enables a library-based low touch flow that can make FPGA development experience similar to that of traditional compute flows in terms of compile time. We believe the high level of productivity and flexibility such an open-source tool offers is essential in the era of FPGAs for computing and acceleration in order to cater to the wider market of software developers.

Additional optimization such as slack allocation techniques are possible to improve the performance. Support for multiple clock domains is another useful optimization and we welcome community collaboration to help improve the quality of results. Taking advantage of multi-threading and parallel routing will be part of future work to further reduce the compile time. RWRoute can eventually be adapted to leverage the FPGA Interchange format [22] which will allow a flexible Intermediate Representation (IR) to emerge long term. This will enable the portability among various architectures and legacy support, which is another desired attribute for FPGA deployment in the data center. Ultimately, a specific domain of interest can take advantage of this open-source router to adapt the cost function to the target domain and improving the results compared to the commercial counterparts. Future work should expand RWRoute timing-driven mode for partial routing in order to achieve this goal. Another clear step for the future work will be adding an open-source placer with similar capabilities.

ACKNOWLEDGMENTS

This work was supported by Xilinx Research Labs. RWRoute source code is publicly available in the RapidWright framework [18]. The opinions expressed by the authors are theirs alone and do not represent future Xilinx policies. We would like to thank Yuanlong Xiao for helping with running the PRFlow. The computational resources (Stevin Supercomputer Infrastructure) and services used in this work were provided by the VSC (Flemish Supercomputer Center), funded by Ghent University, FWO and the Flemish Government - department EWI.

REFERENCES

[1] Vaughn Betz and Jonathan Rose. 1997. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL '97)*. 213–222.

[2] D. Glick, J. Grigg, B. Nelson, and M. Wirthlin. 2019. Maverick: A Stand-Alone CAD Flow for Partially Reconfigurable FPGA Modules. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 9–16.

[3] Benjamin Gojman and André DeHon. 2014. 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*. 88–95.

[4] B. Gojman, S. Nalmela, N. Mehta, N. Howarth, and A. DeHon. 2014. GORK-LAB: Generating Real On-chip knowledge for Intra-cluster Delays Using Timing Extraction. *ACM Transactions on Reconfigurable Technology and Systems* 7, 4 (2014).

[5] Travis Haroldsen, Brent Nelson, and Brad Hutchings. 2015. RapidSmith 2: A Framework for BEL-Level CAD Exploration on Xilinx FPGAs. Association for Computing Machinery, New York, NY, USA.

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

[7] Xilinx Inc. 2019. UltraScale Architecture and Product Data Sheet: Overview (DS890).

[8] Xilinx Inc. 2020. UltraScale Architecture Clocking Resources User Guide (UG572).

[9] H. Javaid, A. Ignjatovic, and S. Parameswaran. 2010. Fidelity Metrics for Estimation Models. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[10] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–140.

[11] Chris Lavin and Alireza Kaviani. 2019. Build Your Own Domain-Specific Solutions with RapidWright: Invited Tutorial. Association for Computing Machinery, New York, NY, USA.

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

[13] Leo Liu and Nachiket Kapre. 2019. Timing-Aware Routing in the RapidWright Framework. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 24–30.

[14] Leo Liu, Jay Weng, and Nachiket Kapre. 2019. RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays. In *2019 27th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 61–64.

[15] P. Maidee, C. Neely, A. Kaviani, and C. Lavin. 2019. An Open-Source Lightweight Timing Model for RapidWright. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 171–178.

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

[17] K. E. Murray, M. A. Elgammal, V. Betz, T. Ansell, K. Rothman, and A. Comodi. 2020. SymbiFlow and VPR: An Open-Source Design Flow for Commercial and Novel FPGAs. *IEEE Micro* 40, 4 (2020), 49–57.

[18] RapidWright. 2021. https://github.com/Xilinx/RapidWright

[19] J. T. Russell and M. F. Jacome. 2003. Architecture-level performance evaluation of component-based embedded systems. In *2003 40th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[20] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–4.

[21] Neil Steiner, Aaron Wood, Hamid Shojaei, Jacob Couch, Peter Athanas, and Matthew French. 2011. Torc: Towards an Open-Source Tool Flow. Association for Computing Machinery, New York, NY, USA.

[22] SymbiFlow. 2021. FPGA interchange schema definitions. https://github.com/SymbiFlow/fpga-interchange-schema

[23] James Thomas, Chris Lavin, and Alireza Kaviani. 2021. Software-like Compilation for Data Center FPGA Accelerators. Association for Computing Machinery, New York, NY, USA.

[24] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. 2013. A connection-based router for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*. 326–329.

[25] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. 2019. CRoute: A Fast High-Quality Timing-Driven Connection-Based FPGA Router. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 53–60.

[26] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. 2020. Fast Linking of Separately-Compiled FPGA Blocks without a NoC. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 196–205.

[27] YosysHQ. 2021. nextpnr – a portable FPGA place and route tool. https://github.com/YosysHQ/nextpnr

[28] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. Association for Computing Machinery, New York, NY, USA.

[29] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. 2020. Accelerating FPGA Routing Through Algorithmic Enhancements and Connection-Aware Parallelization. 13, 4 (2020).