RapidRoute: Fast Assembly of Communication Structures for FPGA Overlays

Leo Liu, Jay Weng, Nachiket Kapre
School of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada
{l276liu, j7weng, nachiket}@uwaterloo.ca

***Abstract*—We can implement relocatable, bus-based communication structures on Xilinx FPGAs using RapidWright while delivering competitive frequency, single digit speedups in execution time, and orders of magnitude reduction in memory usage over Xilinx Vivado 2017.2. We develop RapidRoute, a custom router that exploits symmetry in placement and routing of bus endpoints, caching of reusable route segments, selective multi-threading of the router engine, and abutment-friendly tiling heuristics. The key idea is to reduce the amount of work necessary to generate these communication structures through the use of search heuristics, parallelism, and reuse. We are able to outperform Vivado router by as much as 8× for topologies ranging from 1D rings, torii, and meshes, while taking 1000× lower memory footprint, and delivering timing within 0.2 ns of Vivado. RapidRoute opens the door to building a family of custom routing tools for constructing FPGA overlays for various application domains.**

Source → https://git.uwaterloo.ca/watcag-public/rapidroute

I. Introduction

*In the Japanese Edo period, Ukiyo-e genre of woodblock printing was popular. With this technique, artists could produce complex colorful prints of intricate scenes from multiple woodblocks, each block assigned to an individual color. The final painting was **assembled** by sequential overlay of multiple woodcut prints. RapidRoute can assemble communication structures for FPGA overlay architectures in a similar manner.*

The pursuit of high-level, high-performance programming languages, and tools for FPGA design has long been the holy grail of the FPGA community. Traditionally, FPGAs have been programmed using low-level, register-transfer level (RTL), hardware description languages like VHDL, or Verilog and these continue to stubbornly remain the language of choice. The re-emergence of High-Level Synthesis [2] (HLS) on one hand, and the popularity of FPGA overlay architectures [3] on the other hand gives us two different directions for overcoming the entrapment of the RTL environment. HLS programmings flows provide the allure of coding in high-level languages such as C/C++ but still requires a full FPGA backend pass that can be time consuming. FPGA overlay architectures often require low-level RTL engineering to build the intermediate hardware layer, but provide a potential dual benefit of high-level programming in C/C++ while eliminating the runtime overheads of FPGA placement and routing. This alternative overlay approach still requires an RTL design phase and an unwelcome optimization loop involving the fussy FPGA CAD tool flows.

RapidWright is a new tool that can help us build FPGA overlays quickly using a high-level Java-based programming environment while simultaneously exposing the low-level architecture details of the FPGA for direct manipulation. It is possible to entirely bypass RTL as well as traditional FPGA CAD and enjoy direct access to the FPGA architecture. Some prior work [1], [6] has performed rapid overlay construction through a careful floorplanning, reliance on ISE CAD toolflow for module placement and routing, and the now-outdated XDL [1] environment for relocation. In this paper, we build RapidRoute, a tool that uses RapidWright to generate high-performance layouts of communication structures directly in Java while bypassing the FPGA CAD tools for placement and routing. Vivado is still needed for timing analysis, and bitstream generation. These communication blocks can then be integrated into an overlay design that can be tiled across the entire FPGA in Lego-like fashion.

RapidRoute constructs a range of communication structures by exploiting symmetry and reducing the amount of wasted effort by reusing the partial layouts. It embodies a custom FPGA router that implements a lightweight congestion-aware routing algorithm running significantly faster than the Vivado router that is designed to handle arbitrary circuits. Furthermore, RapidRoute uses caching and natural physical isolation of the communication network structures for multi-threaded parallelization the routing algorithm.

The key contributions of this work are listed below:

- Development of a custom routing algorithm using RapidWright for mapping communication structures on FPGAs.
- Use of symmetry, lightweight congestion-aware design, route caching, and parallelization to reduce work and time required to route the structures.
- Quantification of runtime, memory use, and timing slack for a range of structures like rings, tori, and meshes.

II. Background

A. RapidWright

RapidWright [4] is an open-source tool for Xilinx FPGAs that allows direct construction of FPGA mapped designs in the Java programming environment. Unlike High-Level Synthesis, there is no *compilation* of the code to hardware blocks. Here, Java is used as a high-level language for describing low-level FPGA design. The programmer is able to generate FPGA layouts by direct manipulation of hardware blocks like LUTs,

as well as routing resources. A developer can construct circuits geometrically and exploit relocation and reuse opportunities for generating large FPGA designs from small repeating components. The Jython frontend adds a Pythonic flavor to construction of designs that further enhances user experience. With RapidWright it is possible to generate high frequency implementations that exploit regular patterns in the design that Vivado's general-purpose CAD tools may be unable to.

B. Xilinx Fabric

RapidWright allows a developer to directly access logic resources such as LUTs, and FFs, and also routing resources such as PIPs (programmable interconnect points). In Xilinx terminology, the BEL (Basic Element of Logic) is the simplest building blocks in the hardware. Logic BELs like LUTs represent compute resource of the FPGA while Routing BELs represent data communication resources like switches. In addition, BELs are grouped into Sites with accompanying Site Pins for top-level connectivity, and Site Wires for internal site connectivity. A key feature of RapidWright is the ability to specify placement information for the Site elements. For Logic BELs this is not different than supplying Tcl location constraints to Vivado placer. The relocatability of the placements makes it possible to *copy-paste* layouts far more easily in RapidWright than Vivado. Since we can also choose to occupy routing BELs, we are able to perform our own routing in RapidWright. This is a crucial capability that opens to door to the development of tools like RapidRoute presented here.

III. RapidRoute

In this section, we present the architecture of RapidRoute and the underlying algorithms, search heuristics and associated performance optimizations.

RapidRoute targets fast, high-performance assembly of communication structures. A communication structure is organized as a netlist of bussed register pairs connected in a specified topology. Placement information for the register endpoints is provided as per the geometry of the topology. RapidRoute is seeded with an EDIF file for a bus with input and output registers. Topology information associated with placement of registers is encoded directly in the Java design.

RapidRoute's algorithm core consists of two main elements: (1) the bus router, which routes a bundle of identical wires with register endpoints using a customized negotiation-based algorithm, and (2) the multi-threaded global router, which selectively assigns jobs to the bus router, caching its results, and cloning them wherever possible. Since buses are routed independently from each other, the global router also enacts a conflict resolution procedure, placing emphasis on keeping resolution changes localized within the conflicted tile.

A. Bus Routing

Our bus router is inspired by PathFinder [5] but extensively customized to solve the specific problem of routing regular registered busses.

In the first pass, the bus router operates by only determining wire "hops" between source and sink ignoring resource conflicts. This is done by ignoring PIP configuration at switchboxes along the path, and use of caching of the hop fanouts to avoid querying the RapidWright database repeatedly. By ignoring the PIPs needed within interconnect tiles, we effectively treat each tile as a black box connecting an incoming wire to its appropriate fan-outs. In this stage we are also routing the connections obliviously to each other and the solution may have resource conflicts.

In the second pass, to resolve conflicts (2a) we use an approximate delay based on which wires are part of the path (ignoring PIP delays). This is needed to determine which route deserves higher priority service. (2b) To resolve congestion when filling-in PIPs, we pick one switchbox along the path. (2c) Then we select the most expensive connection and identify the PIP resource needed to connect the two hop wires in that connection. (2d) These resources are then locked down to prevent their use by subsequent connections. Then, we select the next most expensive connection and repeat the steps (2c) onwards. If a resource needed has been previously locked down, we reroute the less-critical connection avoiding the locked resources. Once all connections traversing a switchbox have a home, we proceed to the next switchbox, and repeat the process from (2b) onwards. RapidRoute is designed to complete routing within a single pass even if the outcomes are marginally suboptimal than a full-blown PathFinder.

B. Global Routing

The global router achieves three objectives: (1) assign signal buses for the bus router to compute, (2) copy and paste bus router results to congruent configurations, and (3) resolve all remaining congestion issues, such as the abutment between the entrance and exit of a set of flip-flops. This additional hierarchy provides the key advantage that routes produced by the bus router can be infinitely reusable with negligible cost in routing runtime. The global router identifies symmetrical routes by checking if the sources and sink pairs are identical and offset by the same x-y coordinates. This symmetry in naturally possible when constructing communication structures *i.e.* ring segment i can be copied over to create segment $i+1$.

The global router begins by running an analysis on all bus connections, identifying and temporarily ignoring connections which can be potentially routed by the copy-paste method. Then, the remaining connections are dispatched to the bus router, which will perform routing on separate threads and independent from each other. After all connections are routed, the router attempts to copy and relocate routes to symmetrical connections.

We may encounter conflicts in routing fabric due to copy-paste procedures coinciding over already-used resources. Therefore, a resolution phase is also required to relieve all congestion and conflicts. To prevent any resolution changes from affecting the rest of the design, the global router first attempts to localize changes to within the tile containing the conflicted resource. Thus, only signals passing through the

interconnect will be affected, and only within the specific tile. A similar negotiation-based approach from the bus router is employed, with both the searching and conflict algorithms are constrained to within only the PIPs of the tile. Effectively, it solves an identical problem as the bus router, except sources and sinks are now, respectively, the incoming and exiting wires of each signal. In cases where there is no cheap solution within the tile, the least expensive signal(s) connected to the violated node are completed rerouted, with the node's intrinsic cost adjusted to infinity.

C. Optimizations

To further reduce routing runtime, we use three main optimizations to speed up the routing operations.

Caching: To begin with, enumeration of outgoing paths in a BFS search is an inherently slow process requiring multiple queries of the RapidWright database. However, intrinsic symmetry of Xilinx interconnect tiles allow the fan-outs of incoming wires to be cached and repurposed for corresponding identical wires connected to another tile. Caching is done on-the-fly as the search is conducted, so that new types of wires will only have its fan-outs calculated once. The benefits of caching to the BFS algorithm are two-fold: (a) future route searching become faster by substituting device database queries with cache lookups, and (b) traversal for long-distance wires now takes dramatically reduced time. Accessing the cache requires less than 1% of the runtime needed compared to an un-cached scan of an interconnect tile.

Fast-Forward Search: By identifying the pattern of long wire composition in the Xilinx architecture, we develop a fast forward mechanism to further reduce search duration for long routes. Once a signal search has reached and identified a reasonably long wire (distances of which are specific to the FPGA device family), the traversal will be restricted to connected long wires until the search reaches close to the sink. This fast forward technique prunes the router's ability to exit from a long hop track down to slower wires, allowing the search algorithm to ignore pointless, slow alternatives.

Multi-threading: Lastly, the global router also performs selective multithreading to benefit from better hardware. In general, parallel FPGA routing is tricky due to the challenges of maintaining resource consistency across threads. We identify all unique routing patterns in our communication topology that are guaranteed to not interfere with each other. Each pattern is assigned to a bus routing thread and can proceed in isolation safely. Note that this is different from the copy-paste relocation of the routing structures that is already used by RapidRoute – in this case, we identify distinct routes within a bus bundle that are needed to generate the relocatable layout that can then be copy-pasted. Multithreading is also used during the final congestion resolution phase, since the resolution procedure is localized to within the conflicted tile. Thus, a separate thread is used for resolving each tile conflict in complete physical isolation.

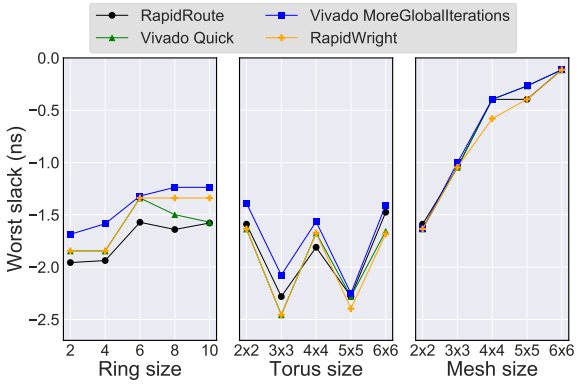

Fig. 1. Timing slack of RapidRoute and Vivado for various system sizes with chip-spanning layout.

IV. Evaluation

In this section, we measure execution time, memory usage, and timing slack performance of RapidRoute against Xilinx Vivado. We evaluate RapidRoute with Xilinx Vivado 2017.2 running on a 32-core 2.6GHz Intel Xeon CPU. We use the UltraScale XCKU115 device for our experiments but have also validated against Kintex UltraScale+ XCKU5P part. We generate various communication topologies such as rings, torii, meshes of different sizes and 8b widths. Compared designs are provided with the same placement constraints, and an identical 1 ns timing constraint. The placement constraints are designed to stress the router by forcing them to use the full device extents and uniformly spacing the registers across the entire device. For Vivado, we provide a placed DCP generated by RapidWright to ensure identical placement starting points for both routers. We restrict both RapidRoute and Vivado to the same number of threads. We compute timing slack of the resulting design in Vivado due to lack of timing analysis support in RapidWright. We configure the Vivado router in *resource-based* mode to optimize for execution time rather than quality due to the simplicity of the layout.

A. Timing Slack

A fundamental requirement of RapidRoute is that it is able to match the quality of Vivado routing output. In Figure 1, we show timing slack observed for an impossible-to-meet 1 ns timing target. RapidRoute is able to demonstrate its ability to remain competitive with Vivado across our experiments. Specifically, in the worst case, we observed that RapidRoute produced designs which are no more than 0.2 ns slower than Vivado. At larger system sizes we observed a faster clock rate due to reductions in distance between register hops.

B. Routing Runtime

In Figure 2, we first show the execution time (averaged across 3 runs) comparison of both RapidRoute and Vivado. For Vivado we **only** measure the execution time of the routing phase and do not include DCP loading times. For RapidRoute, we measure the **complete** runtime including DCP loading times as it is an external tool. It is clear that RapidRoute is consistently 4-8× faster than Vivado for various topologies

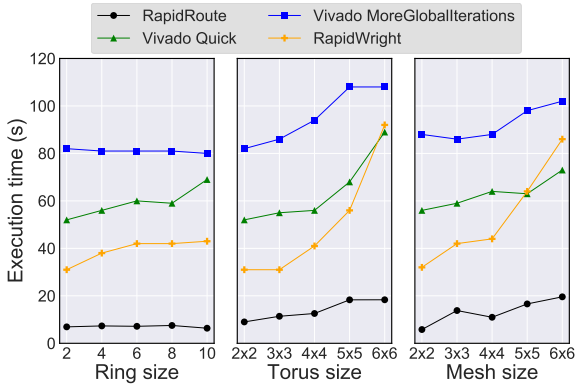

Fig. 2. Average wall clock (real) runtime of RapidRoute and Vivado, routing various network topologies and sizes.

Fig. 3. Average time usage of RapidRoute and Vivado for the each network topology with largest sizes tested from Figure 2.

and system sizes. For RapidRoute we see stable runtimes even as we increase system size due to the use of caching optimizations in the routing algorithm. Vivado is slower, but also relatively stable at larger system sizes as well. A key placement insight we inferred was to pack at most 8 endpoint registers out of 16 in a CLB site to eliminates entry/exit congestion at that CLB. The other registers can be used locally for pipelining.

Next, in Figure 3, we show a breakdown of router runtime for the largest sizes networks (1×10 ring, 6×6 torus and mesh). For RapidRoute, it is clear that the traversal operations during BFS is the key bottleneck. The resources access RapidWright APIs limit our ability to support even faster BFS. We also spend quite a bit of time in our customized congestion resolution phase, and are looking for further avenues for improvements as future work.

C. Memory Usage

Since both RapidRoute and its underlying library, RapidWright, are designed to be lightweight, we expect memory consumption for routing to be orders of magnitude less than that of Vivado. We collect peak memory usage over the lifetime of router execution using the Unix **time** tool. In the worst case, RapidRoute uses only 2–3 MB of RAM during execution, while greedy Vivado allocates 2.7 GB of memory in contrast.

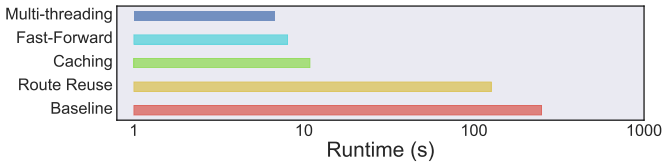

Fig. 4. Cumulative runtime improvements of RapidRoute optimizations for 1×6 ring with 150-tile separation.

D. Effects of Optimizations

We now quantify the impact of each optimization technique on execution time such as reuse, caching, fast-forwarding, and multithreading. For a 1D ring, shown in Figure 4, route caching is the dominant cause of performance improvement. Route reuse is also important as it allows copying the results from one XY location on the chip to another. Fast forwarding and multi-threading have limited but noticeable impact. For a 6×6 mesh, multi-threading gives us a near-linear 8× speedup for 8 threads and diminishing returns beyond.

V. Conclusions

We present RapidRoute, a custom router built around RapidWright to implement relocatable, bus-based communication structures on Xilinx FPGAs. We are able to deliver comparable frequency and single digit speedups over Vivado 2017.2 routing. RapidRoute exploits the natural symmetry available in communication structures like rings, torii, and meshes to reduce the amount of work needed in the Breadth-First Search phase of routing. We also develop fast congestion resolution techniques that avoid the cost of full-blown Pathfinder routing to generate abutment friendly copy-pasteable layouts. RapidRoute outperforms Vivado by as much as 8× while staying with 0.2 ns of Vivado timing slack while requiring 1000× lower memory footprint for chip-spanning rings, torii, and mesh structures.

Source → *https://git.uwaterloo.ca/watcag-public/rapidroute*

References

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

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

[3] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on dsp blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, May 2015.

[4] C. Lavin and A. Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, volume 00, pages 133–140, Apr 2018.

[5] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for fpgas. In *Third International ACM Symposium on Field-Programmable Gate Arrays*, pages 111–117, Feb 1995.

[6] M. X. Yue, D. Koch, and G. G. F. Lemieux. Rapid overlay builder for xilinx fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20, May 2015.