RapidStream: Parallel Physical Implementation of FPGA HLS Designs

Licheng Guo[1], Pongstorn Maidee[2], Yun Zhou[3], Chris Lavin[2], Jie Wang[1], Yuze Chi[1], Weikang Qiao[1], Alireza Kaviani[2], Zhiru Zhang[4], and Jason Cong[1]
[1]University of California, Los Angeles [2]Xilinx, Inc. [3]Ghent University [4]Cornell University
{lcguo,cong}@cs.ucla.edu

ABSTRACT

FPGAs require a much longer compilation cycle than conventional computing platforms like CPUs. In this paper, we shorten the overall compilation time by co-optimizing the HLS compilation (C-to-RTL) and the back-end physical implementation (RTL-to-bitstream). We propose a split compilation approach based on the pipelining flexibility at the HLS level, which allows us to partition designs for parallel placement and routing then stitch the separate partitions together. We outline a number of technical challenges and address them by breaking the conventional boundaries between different stages of the traditional FPGA tool flow and reorganizing them to achieve a fast end-to-end compilation.

Our research produces RapidStream, a parallelized and physical-integrated compilation framework that takes in an HLS dataflow program in C/C++ and generates a fully placed and routed implementation. When tested on the Xilinx U250 FPGA with a set of realistic HLS designs, RapidStream achieves a 5-7× reduction in compile time and up to 1.3× increase in frequency when compared to a commercial-off-the-shelf toolchain. In addition, we provide preliminary results using a customized open-source router to reduce the compile time up to an order of magnitude in the cases with lower performance requirements. The tool is open-sourced at `github.com/Licheng-Guo/RapidStream`.

KEYWORDS

Parallel, Placement, Routing, FPGA, HLS, Dataflow

ACM Reference Format:
Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong, 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *2022 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '22), February 27–March 1, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490422.3502361

1 INTRODUCTION

FPGA compilation techniques have traditionally been adopted from the EDA industry, where designers have higher tolerance of a long turn-around time. However, this significantly impedes the adoption of FPGAs by the computing industry, where software programmers are used to a much shorter compile cycle [38].

One general approach to speeding up FPGA compilation is to utilize multi-core CPUs or GPUs to parallelize the CAD algorithms, such as logic synthesis [18, 19], placement [1, 15, 20, 42–44], and routing [26, 27, 30, 54, 56, 59, 80]. However, many important algorithms used in the FPGA CAD toolflow are inherently sequential. Moreover, the slowest steps of the FPGA physical compilation extensively involve timing optimizations. Since optimizing timing typically requires global knowledge of the designs, it further increases the difficulty of parallelization. In Figure 1, we profile the CPU utilization of a 14-hour FPGA compilation task by the commercial Xilinx Vivado tool suite. As the figure shows, Vivado only uses 2.1 cores on average when attempting to close timing.

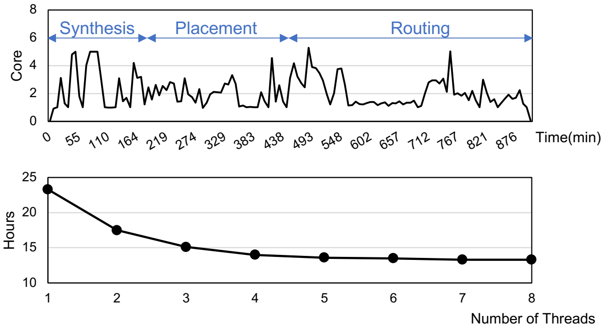


Figure 1: The upper figure shows the number of active CPU cores when implementing a CNN benchmark by Vivado (8 threads) on a 56-core server. The total implementation process takes about 14 hours, and with an average CPU utilization of 2.1 cores. The lower figure displays the runtime as we increase the number of threads.

Another approach to fast FPGA compilation is splitting the whole application into several partitions and then compiling different parts in parallel. A new challenge naturally arises here — *how to achieve timing closure with many inter-partition nets*? Given an RTL design or a netlist, it is relatively easy to partition the design and achieve timing closure within each partition, but it is difficult to achieve good timing on the inter-partition nets. Either we perform global cross-partition optimizations iteratively at the cost of high runtime overhead, or we sacrifice the timing quality of inter-partition nets for runtime efficiency, rendering the acceleration less meaningful.

Despite these challenges, the maturity of FPGA HLS tools in recent years brings new opportunities to address the timing problem of inter-partition nets. Since the input design for HLS is written in *untimed* high-level languages, the compiler has the flexibility to introduce additional pipelining if needed before generating the optimized RTL. Therefore, one may ask if we can couple the pipelining

FPGA '22, February 27-March 1, 2022, Virtual Event, CA, USA

ACM ISBN 978-1-4503-9149-8/22/02.
https://doi.org/10.1145/3490422.3502361

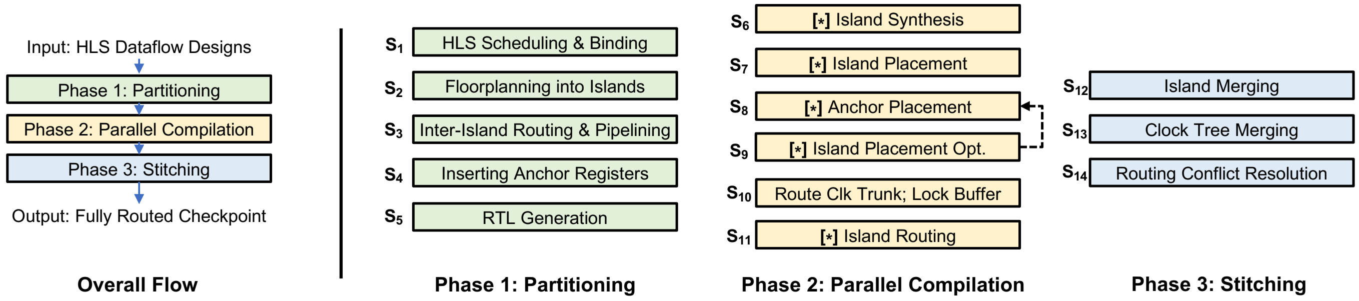

Input: HLS Dataflow Designs
Phase 1: Partitioning
Phase 2: Parallel Compilation
Phase 3: Stitching
Output: Fully Routed Checkpoint
Overall Flow
S1 HLS Scheduling & Binding
S2 Floorplanning into Islands
S3 Inter-Island Routing & Pipelining
S4 Inserting Anchor Registers
S5 RTL Generation
Phase 1: Partitioning
S6 [*] Island Synthesis
S7 [*] Island Placement
S8 [*] Anchor Placement
S9 [*] Island Placement Opt.
S10 Route Clk Trunk; Lock Buffer
S11 [*] Island Routing
Phase 2: Parallel Compilation
S12 Island Merging
S13 Clock Tree Merging
S14 Routing Conflict Resolution
Phase 3: Stitching

Figure 2: An overview of our RapidStream workflow. We use [*] to denote a parallelized step.

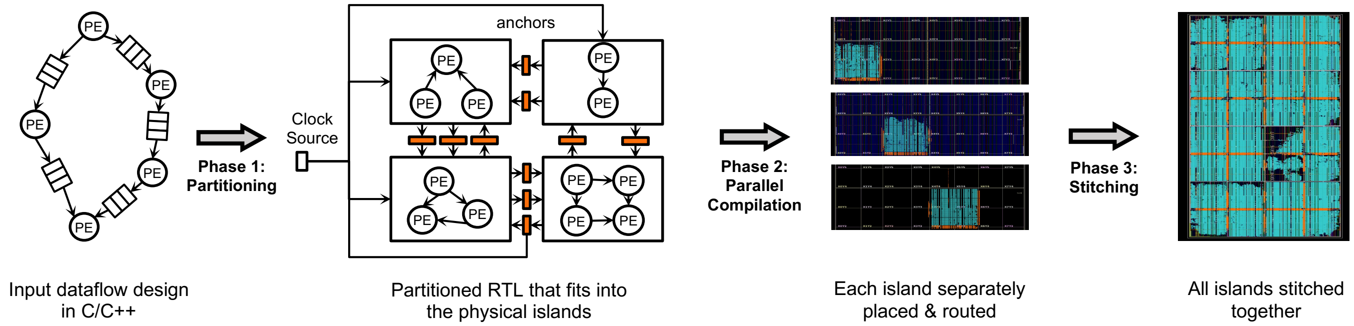

PE
Phase 1: Partitioning
anchors
Clock Source
Phase 2: Parallel Compilation
Phase 3: Stitching
Input dataflow design in C/C++
Partitioned RTL that fits into the physical islands
Each island separately placed & routed
All islands stitched together

Figure 3: Illustration of results obtained in different phases. In the final output, the orange part shows the anchor registers, the cyan part shows the implemented partitions.

flexibility of HLS with the split compilation approach, and if we can first partition an untimed HLS design for parallel implementation, then pipeline the inter-partition nets for timing closure?

In this work, we propose *RapidStream*, a split compilation flow featuring tight integration of HLS-level pipelining and physical design to accelerate the end-to-end FPGA compilation. As illustrated in Figure 2, our method includes three major phases. During the partitioning phase, we organize the FPGA device as a mesh of disjoint *islands* and floorplan a dataflow design into the islands; we then utilize the flexibility of HLS to insert pipeline registers into the inter-island nets, which we call *anchor* registers. The anchor registers provide crucial timing isolation between islands to enable parallel implementation. Finally, we stitch together the layout results of each island to generate the complete implementation.

Compared to the prior arts that also employ a split compilation approach [62], RapidStream has several distinct characteristics. First, we achieve full automation while [62] relies excessively on manual inputs, including design modification, floorplanning, pin assignment, etc. Second, we achieve a clock frequency close to 400 MHz but [62] only reports a frequency of 187 MHz. One of the key differences is that [62] relies on a fixed pre-routed overlay structure to isolate the islands, but at the expense of flexibility and timing quality. In contrast, RapidStream can exploit design-specific optimizations without using a pre-configured overlay, which helps improve timing. We will provide a detailed comparison in Section 8.

Our key technical contributions are summarized as follows:

- To the best of our knowledge, we are the first to propose an automated, parallelized, and physically-integrated flow to map HLS dataflow designs into a fully placed and routed FPGA implementation while achieving fast timing closure.
- We identify and address several technical challenges for a practical split compilation flow. Specifically, we propose new and effective methods for (1) inserting pipeline registers and optimizing their placement at the latency-tolerant borders of partitions, (2) clock management in parallel routing, and (3) efficient island stitching and routing of inter-island nets.
- Our evaluation shows that the proposed approach significantly increases the degree of parallelism of FPGA-targeted split compilation. RapidStream uses ~26 cores on average, whereas a commercial CAD tool only utilizes about two cores on average. As a result, we achieve an end-to-end speedup of 5-7× over the commercial tool. Additionally, we achieve an improvement in frequency by up to 1.3×.

2 PRELIMINARIES

2.1 Problem Scope

RapidStream focuses on HLS dataflow designs. By our definition, a dataflow design consists of (1) a collection of *processing elements (PE)* working in parallel and (2) a set of FIFOs that connect the communicating PEs. Each PE can be arbitrarily complex internally, but it must send or receive data through FIFO interfaces.

2.2 Organization of the FPGA Fabric

To facilitate the split compilation, we divide the FPGA fabric into two types of regions. As illustrated in Figure 4, these regions include (1) large disjoint *islands* (in blue) that are equally sized and (2) thin columns/rows of *anchor regions* (in green) between adjacent islands. Here we define an island as a square-shaped region reserved for (a subset of) the user logic; we further require that different islands are non-overlapping. Meanwhile, the anchor regions are reserved to place the anchor registers (in orange) needed for inter-island communications; each inter-island connection is equipped with one anchor register, which isolates the inter-island timing paths.

Note that we need to distinguish the anchor regions located at die boundaries. The Xilinx multi-die FPGAs have discrete channels for die-crossing signals. To facilitate timing closure, the anchor registers will be placed in the die-crossing channels to bridge the islands that are on different sides of the die boundary (see Figure 4).

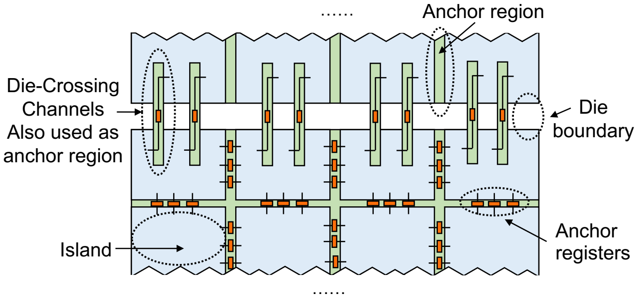


Figure 4: Organization of the FPGA device.

2.3 Flow Overview

Figure 3 shows the input and output of each phase of our proposed workflow. In Phase 1, we take in an HLS dataflow design and floorplan it to the disjoint islands (steps S_1 and S_2 in Figure 2). We take advantage of the elasticity of dataflow designs to ensure that every inter-island connection is pipelined with an *anchor register* (S_3 and S_4). This provides timing isolation that is crucial in the later parallel placement and routing.

Phase 2 performs parallel placement and routing of the disjoint islands and inserts the anchor registers. In the placement step (S_7-S_9), we propose to iteratively co-optimize the placement of anchors and islands since they are interdependent. In the routing step (S_{10}-S_{11}), we propose a clock management scheme to ensure that the clock skew is consistent when the islands are routed and later stitched together. Without this step, we will run into hold violations after stitching.

In Phase 3, we implement a stitcher using the RapidWright framework [39] to stitch the physical netlists of post-routing islands together (S_{12}, S_{13}). Although the nets inside each island remain legal after stitching, conflicts may arise among the inter-island anchor nets. This is a routing problem unique to our flow, and we propose a lightweight method to resolve the potential routing conflicts (S_{14}). Compared to the full-fledged commercial router, we achieve a 4× speedup on average while retaining nearly the same setup slacks.

3 PARTITIONING

This section describes steps S_1-S_5 of the partitioning phase of RapidStream, as shown in Figure 2.

3.1 Problem Description

In this phase, we exploit the pipelining flexibility of HLS to transform the design into a parallelization-friendly structure. We first discuss what features are needed in later phases that parallelize the physical implementation of islands.

Objective 1: Non-overlapping partitioning – Since we aim to parallelize the physical implementations of different islands, each island is required to host a unique and non-overlapping partition of the original design.

Objective 2: Pipelined inter-island connections – To facilitate the timing closure on the inter-island nets, we want each inter-island connection to be pipelined with an *anchor* register.

Objective 3: Direct neighbor connections – We further enforce that each island only has direct connections with adjacent islands. This property is key to parallelizing the placement and routing process.

3.2 Approaches

Next, we introduce how RapidStream partitions and transforms the original dataflow design to satisfy the above-mentioned objectives.

Mapping PEs to Islands (S_2). To achieve objective 1, we exclusively assign each PE to one island. The assignment problem is formulated as follows:

The input dataflow design is represented as a graph $G(V, E)$, where each vertex $v \in V$ represents one PE; each edge $e_{ij} \in E$ represents an inter-PE FIFO connection between v_i and v_j. Given an array of islands that has N rows and M columns, the goal is to map each $v \in V$ to one unique island such that the resource of each island is not overused and the total wirelength is minimized. We use the weighted Manhattan distance to calculate the total wirelength:

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|) \quad (1)$$

where $e_{ij}.width$ is the bitwidth of the FIFO between v_i and v_j and each v is assigned to the $v.col$-th column and the $v.row$-th row.

The rationale behind the formulation is that a shorter wirelength results in a lower latency overhead. Our problem is typically small in size since an HLS design usually only instantiates up to a few thousand PEs. Hence we use integer linear programming (ILP) to formulate and solve a top-down partitioning-based placement problem in an iterative manner. Notably, the placement problem is similar to the ones described in several prior works [2, 22, 28, 47].

Global Routing & Pipelining Inter-Island Connections (S_3). Before we pipeline the connections between non-adjacent islands, we need to first determine which intermediate islands the connections will go through. Essentially, we need to first solve a routing problem at the island level. Next, we insert pipeline registers in the islands that the connection passes through. As an example, Figure 5(A) shows the potential routes (P_1, P_2, P_3) for an connection between two non-adjacent islands.

The main constraint in this routing problem is the number of available flip-flops (FFs) in the anchor regions. Recall in Figure 4 that we reserve a thin region between islands to hold the anchor registers for inter-island nets and each inter-island net has an anchor register. Therefore, when routing the connections at the island level, we

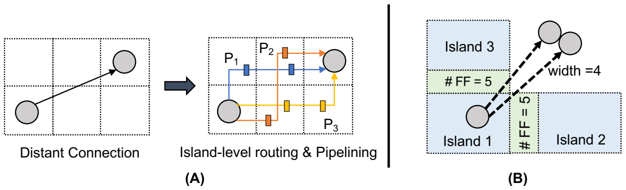


Figure 5: (A) three potential routes for a connection. (B) Each anchor region (in green) only has 5 Flip-Flops, so the two connections (both of width 4) cannot go through the same anchor region.

must ensure the participating anchor regions have sufficient FFs for pipelining all the nets passing through, as illustrated in Figure 5(B).

Since the number of islands being mapped to is typically small, we again formulate the problem in ILP. For each connection, we generate all potential routes with the shortest Manhattan distance that have at most two bends. For each anchor region between a pair of adjacent islands, we add a constraint to ensure that the number of passing-through nets is no greater than the available FFs. We also assign a cost to each route based on the average resource utilization of the passing islands. The ILP is set up to minimize the total cost in this path selection problem.

Inserting Anchor Registers (S_4). To facilitate timing closure and inter-island routing, each island will register all input/output signals. Figure 6 shows how we insert anchor registers into the inter-island nets between adjacent islands. We leverage an *almost-full FIFO* which asserts the `full` signal before the FIFO is actually full. This signal increases tolerance of the round-trip latency between adjacent islands, which allows us to add a pipeline register without causing an overflow.

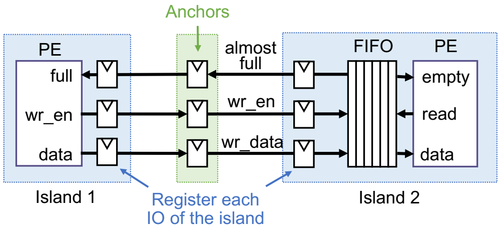


Figure 6: Inserting anchor registers.

Note that we choose to use the ILP formulations because they are sufficiently fast and scalable for today's HLS designs and FPGA devices. This is validated by our experiments in Section 7. For future FPGA designs that may become much larger, we can incorporate other well-known techniques such as multi-level placement [6] and hierarchical routing [68] to handle the increased complexity.

4 PARALLEL PLACEMENT

Phase 1 produces an optimized version of the RTL that is floorplanned to the island regions and anchor regions (Fig. 4). In step S_2, we determine which PEs are assigned to each island region; and in step S_3 we compute which anchor registers that each anchor region accommodates.

In Phase 2, we first synthesize the RTL of each island into the netlist representation (S_6). As all islands are non-overlapping, we are able to run logic synthesis for all islands in parallel.

Next, we place all island regions and anchor regions in parallel based on the previous floorplanning (S_7-S_9).

4.1 Iterative Placement of Anchors and Islands

Compared to logic synthesis, it is more challenging to parallelize the placement step. Two neighbor islands that are independently placed should have their interface properly aligned. This requires the separate placer processes to properly synchronize on inter-island connections.

We adopt an iterative approach to gradually align the interfaces of separately-placed islands by utilizing the anchor regions between islands. Figure 7 sketches the main ideas of our approach. The intuition is that we lock the placement of all islands and then incrementally re-place the anchor regions, then alternate their roles in the next iteration.

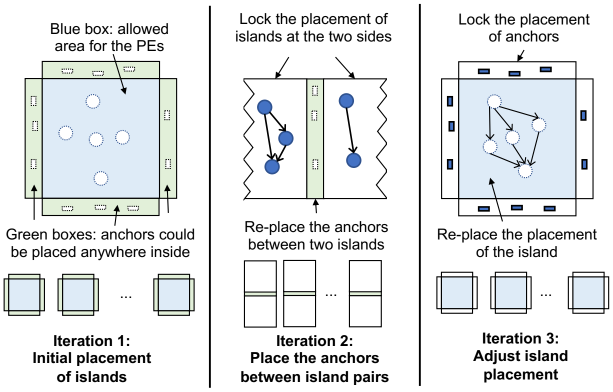

Blue box: allowed area for the PEs
Green boxes: anchors could be placed anywhere inside
Iteration 1: Initial placement of islands
Lock the placement of islands at the two sides
Re-place the anchors between two islands
Iteration 2: Place the anchors between island pairs
Lock the placement of anchors
Re-place the placement of the island
Iteration 3: Adjust island placement

Figure 7: Demonstration of the iterative placement.

Iteration 1 (S_7). In the first iteration, we determine an initial placement of the islands. To place an island by itself, the placer needs the locations of all anchors around the island, which are unknown at the time. So we only impose a partial constraint that each anchor should be within the anchor region on its corresponding side of the island.

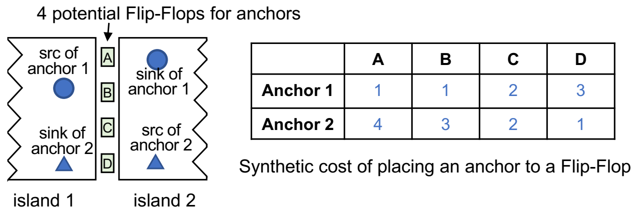
Iteration 2 (S_8). With the initial placement of each island, we compute the exact locations of the anchors between the islands to connect the inter-island nets. This step is also carried out by parallel placer processes. Each process handles a pair of adjacent islands and places the anchors in between to best connect both sides. We further elaborate this step in Section 4.2.

Iteration 3 (S_9). We fine-tune the placement of islands based on the exact anchor locations. Since the resulting anchor locations from the first two iterations may differ, iteration 3 further refines the placements of the islands to best match the latest anchor locations from iteration 2.

Through the three iterations, all islands are placed in a parallel manner. It is possible to repeat iteration 2 (S_8) and iteration 3 (S_9) to further improve the overall timing quality. However, our experiments indicate that applying them just once already achieves a post-placement frequency of 400 MHz.

4.2 Anchor Placement by Min-Cost Matching

Motivation. While we use the standard placer for iterations 1 and 3, we formulate the anchor placement problem (iteration 2) as a min-cost matching problem. Iteration 2 places the anchors based on the placement of the islands on the two sides. First, since the anchor region is very thin[1], it is effectively a 1-D placement problem and the solution space tends to be small. Second, using the standard placer would incur unnecessary overhead in compile time as it is optimized towards general situations. Finally, we need control in a finer granularity to make sure that all anchors are exactly inside the feasible regions.


4 potential Flip-Flops for anchors
src of anchor 1
sink of anchor 2
A
B
C
D
sink of anchor 1
src of anchor 2
island 1
island 2
Synthetic cost of placing an anchor to a Flip-Flop

	A	B	C	D
Anchor 1	1	1	2	3
Anchor 2	4	3	2	1

Figure 8: Illustration of the anchor placement formulation

Method. We propose a simple yet effective distance-driven placement formulation specifically for iteration 2 (S_8), which can achieve a similar timing quality compared to a standard placer but with a much shorter running time. Given an anchor, we assign a heuristic value for each FF in the anchor region representing the cost to place the anchor onto that FF. Then we minimize the total cost of placing all anchors. This formulation is a min-cost matching problem that can be solved in polynomial time [7]. Specifically, we formulate the problem in linear programming (LP), which in this case guarantees integer solutions because the constraint matrix is totally unimodular [34].

We use a heuristic method to determine the cost function. To place an anchor onto an FF, the cost consists of two parts: (1) the total wirelength from the anchor to the source and sink cells; (2) the wirelength difference between the longest and the shortest net of the anchor. We sum the two parts with empirical weights. This distance-based heuristic will push the anchors close to their source and sink cells and avoid being too close to one cell but far away from the other.

Consider the example in Figure 8, where we need to place two anchors to four potential FFs (A, B, C, and D) between the islands. Since the source and sink of anchor 1 are at the top, A has a smaller cost than others. Likewise, D has the smallest cost for anchor 2.

Our LP placement scheme for the anchors is on average 20× faster than the commercial placer and the timing quality is similar.

5 CLOCK ROUTING

5.1 Problem Description

After we finalize the placement of the islands and anchors, we next aim to route the islands in parallel. Since all inter-island connections are anchored, we only need to route each island to connect to its surrounding anchors. However, we need to take special care of the clock signal because it is a global net that fan-outs to all islands.

5.2 Challenges and Previous Approaches

Clock routing and data signal routing are interdependent. In a general non-split routing process, the router will first generate an initial clock tree and then route all the data signals. Later, the router may adjust the clock tree for timing optimization.

However, when we route standalone islands separately, the router is unaware of the final clock tree for the entire design. If the island is routed under a different clock tree compared to the final clock tree, the variation in the clock skew will cause timing degradation as well as hold violations. Consider a simple example where the clock signal may enter an island either from the left side or the right side. If the island is routed assuming the clock is from the left, but the actual clock signal arrives from the right in the final stitched design, then the variation in clock skew will cause timing degradation.

A common solution is to first route each island using estimated clock delays and skews; after all islands are combined, the router will globally finalize the clock and re-route the islands to deal with clock skew variations [65]. However, this approach requires an additional global routing step that compromises the compile time.

To address this challenge, we propose dedicated clock management steps to ensure a consistent clock skew before and after the stitching process. Our clock routing flow consists of three steps, which are elaborated in the following subsections. Figure 9 visualizes the key concepts in our clock management scheme.

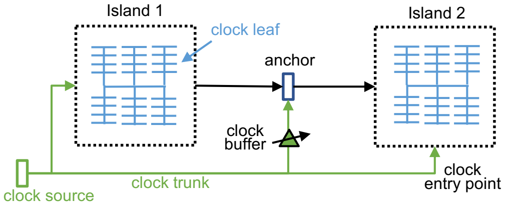


Figure 9: Route different segments of the clock separately and maintain a stable clock skew in one pass. Step 1: route the clock trunk. Step 2: lock the delay level of the clock buffers for anchors. Step 3: route each island and merge with the clock trunk.

5.3 Routing the Clock Trunk (S_{10})

The goal of this step is to route from the clock source to the clock entry points of each island. We refer to this route segment as the *clock trunk*. Here we aim to minimize the clock skew among those entry points. To do so, we first route the clock signal from the clock source to the geometry center of all islands. From there, we fan-out the clock to reach all islands while minimizing the skew. The obtained clock trunk will be used to constrain the clock routing of each island.

5.4 Locking the Clock Buffers for Anchors (S_{10})

With the clock trunk, we have determined the clock entry points for each island. Since two adjacent islands will route to the same set of anchors in between, we need to *disable* the time-borrowing optimization [21, 23, 69] on the anchor registers to prevent clock skew variations of inter-island paths.

In modern FPGAs, the clock network is equipped with buffers that have configurable delay levels to fine-tune the clock skews [36,

[1] Typically, an anchor region requires 1-3 FF columns, about 1/25 the width of an island.

66]. The time-borrowing optimization can utilize the configurable buffers to redistribute the timing slack between consecutive pipeline stages, as demonstrated by Figure 10.

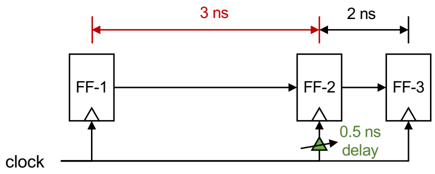


Figure 10: By introducing an artificial clock delay of 0.5 ns to FF-2, the critical path is reduced from 3 ns to 2.5 ns.

In our flow, we separately route two adjacent islands that connect to the anchors between them. The two independent router processes may result in different time-borrowing schemes and thus different clock buffer configurations for the shared anchors. Such potential inconsistency on the clock delay levels for the shared anchors will cause unpredictable timing degradation when the islands are stitched together in the final phase.

To prevent this potential issue, we lock the delay level to the default value for all clock buffers associated with anchor registers.[2] To mitigate the negative impact of this disabled optimization, two aforementioned techniques are beneficial: (1) the source and sink of each anchor net are both pipelined; (2) the local placement optimization performed after fixing the anchor locations (S_8).

5.5 Routing and Merging the Local Clocks (S_{11})

With the setup from the previous steps, we are ready to route each island (S_{11}). We enforce the constraint that the local clock net starts from the pre-determined entry point and prevent the clock buffers for anchors from being adjusted. A routed island will contain a complete clock route, including the clock trunk. During the final island stitching, redundant clock trunks are unified (S_{13}).

Summary. The clock management steps (S_{10}, S_{11}) ensure that the clock skew remains consistent before/after we stitch the islands together. Since the clock entry points within an island are the same before and after the stitching, the clock skew for intra-island timing paths will remain unchanged. In addition, since we lock the delay level for the anchor registers, the clock skew for inter-island timing paths is also stable. Section 7 shows that without the clock management, we will run into severe hold violations; meanwhile, the measured impact of this method on the achievable frequency is negligible.

6 STITCHING AND INTER-ISLAND ROUTING

6.1 Island Merging (S_{12}, S_{13})

In the previous sections, we present how to place and route the islands in parallel. As a result, we will obtain separate post-routing checkpoints, each for one island. Next, we need to assemble them together into the complete physical implementation. While this step is conceptually simple, it is not supported by the off-the-shelf commercial tools. We utilize the open-source RapidWright framework [39] to edit the netlists and assemble the physical information of the island checkpoints.

The checkpoint of each island also includes its surrounding anchor registers. Thus when we stitch the netlists together, we need to unify (or merge) the duplicated anchor registers, as the same anchor is included in the checkpoints of both islands on its two sides. Since the physical information of the duplicated anchors are consistent after the parallel placement (Section 4), we can safely merge them without causing conflicts in anchor locations. Further, our clock routing scheme (Section 5) ensures that different islands are routed under the same clock trunk, thus the clock net can also be merged without conflicts (S_{13}).

6.2 Inter-Island Routing (S_{14})

After the individual checkpoints are assembled together, we need to resolve the routing conflicts in the anchor regions. This is the last step of the RapidStream flow.

Problem Description:
Figure 11 shows the low-level routing resources in the anchor region and why routing conflicts may arise. Since the switch boxes in the anchor region are shared, the two router processes may both exploit the same physical wire segments when they separately route islands 1 and 2. According to our profiling, the conflicting nets in the anchor region amount to 5-10% of all the nets. Those conflicts will be exposed after we glue the post-routing checkpoints of islands together.

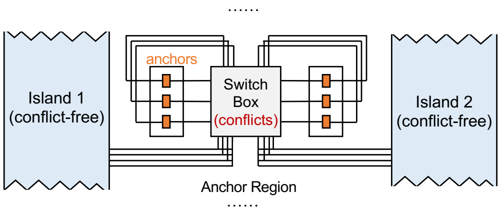


Figure 11: Detailed view of anchor region. Only 1 switch box shown.

One potential solution is to resolve the inter-island conflicts pair by pair. Figure 12 illustrates why this will not work. In Figure 12 we could try to separately re-route the conflict nets between islands (1, 2) and between islands (2, 3). However, while a pairwise re-routing resolves the anchor region conflicts, it will lead to new conflicts within the islands. In Figure 12, assume the black and the yellow net are separately routed by two router processes, conflicts may show up inside the islands (the red segment).

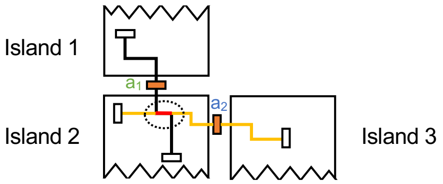


Figure 12: Pairwise inter-island routing will not work because it may cause conflicts inside the island.

Therefore, we have to do a global routing pass to fix the inter-island conflicts. We present two solutions for this routing task. One set of experiments uses the Vivado router in order to maintain the best performance, while the other solution relies on a customized open-source routing solution for the best compile time.

[2] In Vivado, this can be achieved by setting the FIXED_ROUTE property of the clock net.

Solution 1: with Commercial Routers

Commercial routers can resolve the inter-island conflicts at the expense of some runtime overhead because they are optimized for general purpose routing. The Vivado router spends about 1/4 of the time for initialization; 1/4 of the time for the actual routing and timing closure; and 1/2 of the time to loop through a set of optimization steps even after timing closure.

However, our routing problem has two unique features. First, 90-95% of the nets (intra-island) are fully routed and have been well optimized for timing. Second, the conflicts are clustered in the anchor region between islands. In this case, we can potentially utilize the special properties of the problem for further speed-up.

Solution 2: with Customized Partial Router

For this unique problem, we build a lightweight *partial router* that only rips up and reroutes the conflicting nets from/to the anchor regions. Meanwhile, the partial router preserves other fully routed nets, i.e., masking the routing resources used by those nets and skipping any processing on those nets.

One challenge of preserving the non-conflicting nets is how to determine suitable sizes for the *bounding boxes*. During the routing process, the bounding boxes restrict the accessible routing resources for the net. Usually, their sizes are determined based on the pin locations of a net. A large bounding box allows more flexibility for the net but will incur extra runtime; while a small bounding box limits the routability but also reduce the route time. In a typical routing process with no preserved nets, the effective bounding boxes for all nets could be determined in advance and will remain fixed during routing [27, 30, 45, 59, 80]. However, the conventional approach does not work in our situation due to the reduced routing flexibility after we preserve all the intra-island nets.

Figure 13 shows a case where a net needs long horizontal routing detours outside of its bounding box. This is because there is resource blockage within the initial bounding box resulting from the preserved nets. Without expanding the bounding box, the net cannot be routed. There are also cases where vertical long routing detours are needed for successful routing. Therefore, it is difficult to determine suitable bounding boxes for all the target nets before routing.

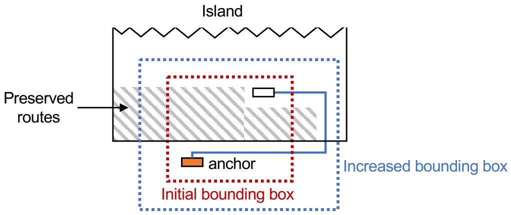


Figure 13: Required long routing detours outside of the initial net bounding box.

To address this issue, we use a simple heuristic to start with small bounding boxes and incrementally increase the box size. Starting from the second iteration, our router expands the four sides of the bounding box for each net that will be be ripped up and rerouted.

We achieve the goal by customizing an open-source router called RWRoute [79]. We upgrade its partial routing function to be timing-driven and enable the tool to expand bounding boxes at runtime. With a single thread, our customized router achieves 4× speed-up compared to the Vivado router.

As of now, RWRoute relies on an open timing model [48] to achieve timing-driven routing. However, this model provides only the *slow path* delay estimation of routing resources. As a result, RWRoute could not resolve hold violations which require the *fast path* delay estimation of the routing resources. We present a temporary workaround in the next section to eliminate hold time requirements at the expense of some performance.

Workaround for Hold Violation in Solution 2

Since the customized RWRoute will only route the nets to/from the anchor registers, we make all anchor registers to be triggered by the negative clock edge, e.g., in Figure 6, modify the registers in the green box to be triggered by the *negative* clock edge while keep everything else triggered by the *positive* clock edge.

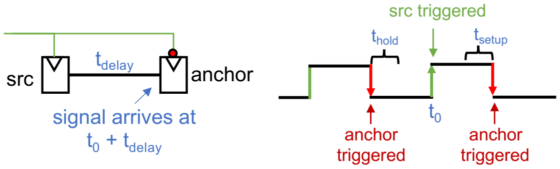


Figure 14: Make anchors trigger on negative clock edges.

Figure 14 depicts the idea when the anchor is the signal sink. The same reasoning applies when the anchor is the signal source. Assuming a zero clock skew, the source FF is triggered at t_0 and the anchor FF is triggered at $t_0 + t_{period}/2$ to transfer Signal i. The signal will arrive at the anchor at $t_0 + t_{delay}$. For Signal i to be properly captured at the anchor FF while still not interfering with the capturing of Signal $i - 1$, both Equation (2) and (3) must be satisfied.

$$t_0 + t_{slow\_delay} < t_0 + t_{period}/2 - t_{setup} \tag{2}$$

$$t_0 + t_{fast\_delay} > t_0 - t_{period}/2 + t_{hold} \tag{3}$$

Equation (2) and (3) can be reduced to (4) and (5):

$$t_{period} > 2(t_{setup} + t_{slow\_delay}) \tag{4}$$

$$t_{period} > 2(t_{hold} - t_{fast\_delay}) \tag{5}$$

Therefore, with negatively-triggered anchors, we can always increase the clock period to satisfy the conditions and thus avoids any setup/hold violation on the anchor nets when RWRoute reroutes them to fix conflicts in the anchor region. Meanwhile, the intra-island nets are routed by Vivado and are free of hold-violation.

Note that this technique of clock phase shifting is a temporary measure, which will no longer be needed if an open fast-path timing model is provided. This experiment shows us the potential for the best runtime and advantages of an open source partial router.

7 EXPERIMENT

7.1 Implementation Details

The core of RapidStream is implemented in Python (about 8K LoC). We first summarize the tools used in the RapidStream flow. The timing target is 400 MHz (2.5 ns period).

Phase 1: we use Vivado HLS 2020.1 to generate the initial RTL, then floorplan the HLS dataflow design (S_1, S_2). Based on the floorplanning results, RapidStream will post-process the RTL generated by Vivado HLS to insert the inter-island pipelines (anchor registers) and rebuild the RTL hierarchy for each island (S_3-S_5).

Phase 2: we use Vivado 2021.1 to synthesize each island (S_6). For island placement (S_7, S_8, S_9), we use Vivado (`place_design`) for initial island placement (iteration 1, S_7); then use our ILP-based method to place the anchors (iteration 2, S_8); finally we switch back to Vivado (`phys_opt_design`) to incrementally optimize the placement of islands (iteration 3, S_9). In island routing (S_{10}, S_{11}), we pre-build the clock trunk (set_property `FIXED_ROUTE`) and lock the clock buffer[3] for anchors (S_{10}), which are passed as constraints to the Vivado router (S_{11}). We use the "Explore" directive in Vivado.

Phase 3: we build a stitcher based on RapidWright to edit the netlist of islands and put them together (S_{12}, S_{13}). Then we use Vivado to resolve for inter-island routing (S_{14}). We separately compare Vivado and our timing-driven partial router RWRoute on S_{14}.

Island Organization: In our experiments, we target the Xilinx UltraScale+ U250 FPGA, which consists of 4 dies stacked vertically. At present, we employ an empirical scheme to organize the FPGA fabric as 32 islands in 8 rows (4 islands per row), with each island being 120 CLBs[4] in height[5]. Between adjacent islands, we reserve 3 empty columns (or 10 rows for vertically adjacent islands) of CLBs as the anchor region to accommodate the anchor registers. The width of the anchor region is approximately 1/25 as that of an island. At die boundaries, we use all Laguna columns as the anchor region (Figure 4).

Two-Level Stitching: Specifically for Xilinx UltraScale+ devices, we employ a two-level method in Phase 3. We first stitch the island-level checkpoints into die-level checkpoints and route the inter-island nets; then we stitch together all the die-level checkpoints into the final checkpoint. Note that in the second stitching step, the die-level checkpoints could be readily assembled without any re-routing. As shown in Figure 4, the anchor regions at the die boundary of the Xilinx UltraScale+ FPGAs are different, where the islands on the two sides of the die boundary rely on the dedicated Laguna channels for cross-die signals. Since the actual wires within the channel are point-to-point and separated from each other [64], there will be no conflicts when merging die-level checkpoints.

Distributed Execution: Each step will be launched as soon as its input is ready. For example, the placer process for an island will start immediately after the corresponding synthesis process has finished, and there is no synchronization to wait for all synthesis processes to finish. Likewise, the process to optimize the island placement will start as soon as the dependent anchor placement processes have exited and all surrounding anchors have been placed.

Environment: We test RapidStream using 4 servers, each with the 56-core Inter Xeon E5-2680 v4 CPU at 2.40GHz and 128 GB of memory. All servers run under the Ubuntu 18.04 operating system.

7.2 Benchmarks

We test six large-scale dataflow designs as shown in Table 1. We denote the number of PEs as "#V" and the number of FIFO connections between PEs as "#E". The matrix multiplication (MM), CNN, L/U decomposition (LU) and MTTKRP are from the AutoSA project [60]; the 2-D and 3-D stencil accelerators are from the SODA project [11].

The benchmarks are mapped onto our target U250 FPGA, which has 5376 BRAMs, 12288 DSPs, 3456K FFs and 1728K LUTs. The mapped benchmarks use 60% - 70% of the available resources.

Table 1: Benchmarks.

Name	# V	# E	Topology	DSP %	BRAM %	FF %	LUT %
MM	463	854	2-D Mesh	62	23	34	69
CNN	439	813	2-D Mesh	59	33	32	50
LU	1691	4483	Triangular	20	41	26	66
MTTKRP	360	760	2-D Mesh	66	33	30	48
2-D Stencil	266	1562	Irregular DAG	52	21	27	45
3-D Stencil	1314	2866	Irregular DAG	64	39	35	53

7.3 Runtime Reduction

Figure 15 shows the comparison of runtime and the achievable frequency between the vanilla Vivado flow and RapidStream. Since RapidStream will insert additional pipelining to the RTL, we consider two Vivado baselines: (1) the original RTL generated by HLS and (2) the version that has been pipelined by RapidStream.

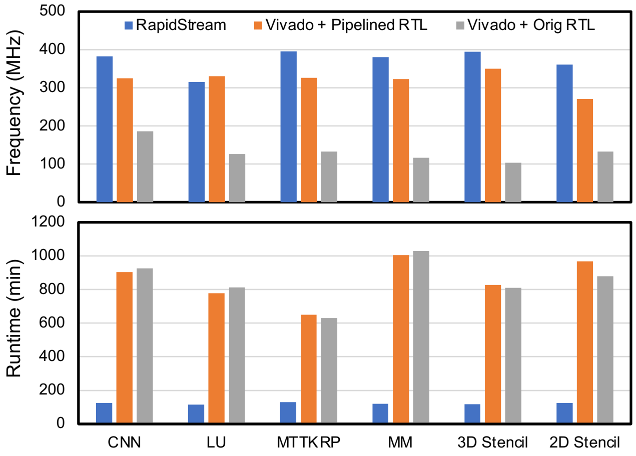


Figure 15: Comparison of the runtime and achievable frequency between RapidStream and Vivado.

By default, we use Vivado for inter-island routing (S_{13}) to pursue the best timing quality. In this case, we achieve 5-7× speed-up and reduce the otherwise >10-hour process to around 2 hours.

In terms of frequency, we achieve better results than both baselines. Since each island is much smaller than the entire design, Vivado could better optimize the timing of each island. The only exception is the LU benchmark, which has lots of division operations that become the critical paths in both flows.

Figure 16 shows the CPU and memory utilization when we use RapidStream to compile the same CNN design as in Figure 1. While Vivado uses 2.1 cores on average and runs for about 14 hours, we use 26 cores on average and runs for about 2 hours.

[3] Refer to our code for more details.

[4] Each CLB in Xilinx FPGAs contain 16 FFs.

[5] Meanwhile, the width of islands may vary slightly based on clock region boundaries.

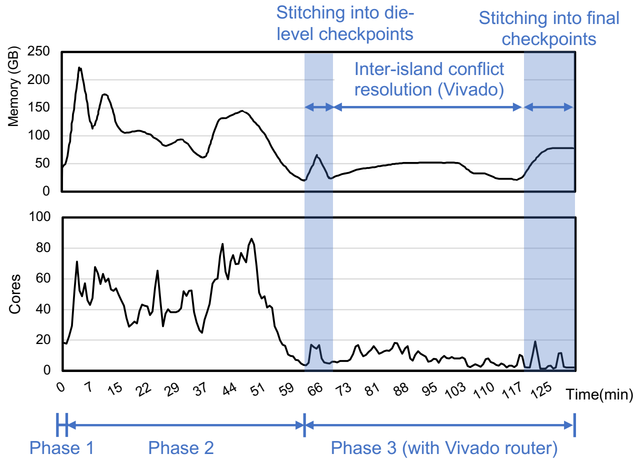


Figure 16: CPU and memory usage of the RapidStream run on the CNN design. No re-route needed after die-level stitching (Sec. 7.1)

Figure 17 shows in detail the time break down of Phase 2 in Figure 16. We plot how many islands are in each step. This asynchronous execution alleviates the long tail issue within each step.

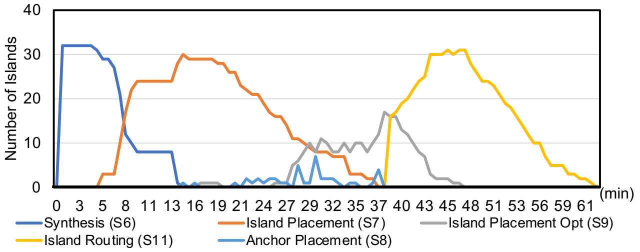


Figure 17: Number of active jobs in Phase 2. E.g., in minute 11 there are 24 islands in synthesis while 8 islands have started placement.

7.4 Fast Inter-island Routing

In Figure 16 we have a long tail issue, which corresponds to Phase 3, where we use Vivado to resolve the inter-island routing conflicts. As mentioned in section 6.2, we customize the open-source RWRoute to further accelerate this step. Figure 18 shows the comparison between using the customized RWRoute and using Vivado for S_{14}.

On average we achieve 4× speedup over the Vivado router, reducing the conflict resolution time from about 25 minutes to about 6 minutes. The RWRoute flow achieves lower Fmax because it relies on negatively-triggered anchors (Section 6.2) to prevent hold violation, which sacrifices the setup slack. This performance loss will be remedied once an open timing model with fast-path is available.

In addition to the routing time reduction, iterating between Vivado and RapidWright through checkpoint read/write is reduced, since our routing solution is also implemented under the RapidWright framework and the output of the stitcher could be passed in memory to RWRoute. This will further alleviate the long tail in Phase 3. By projection, we could reduce the end-to-end time reported in Section 7.3 down to around 80 minutes, which is 7-10× speed-up over the traditional Vivado flow.

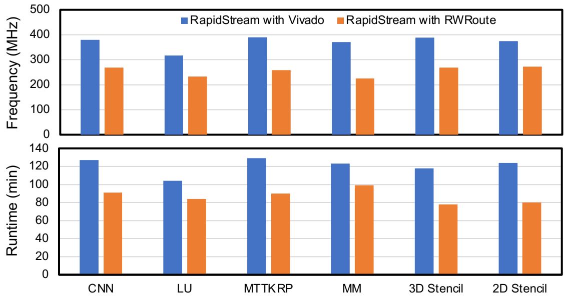


Figure 18: Runtime comparison in conflict resolution.

7.5 Anchor Placement

In our 3-iteration approach to place the islands and anchors (S_7-S_9), we propose a concise min-cost matching formulation for the anchor placement (iteration 2, S_8). We use the MM benchmark to compare our lightweight placer with the Vivado placer. With 32 islands, there will be 52 island pairs and we will have 52 placer processes, each for one pair of islands.

On the runtime side, the min-cost matching placer takes less than 1 minute to place the anchors between pairs of islands; while it takes Vivado 21 minutes on average (including the time to read the checkpoints). On the timing quality side, both placement schemes will achieve above the 2.5 ns target period after the three iterations, as shown in Figure 19. Note that the timing report is based on Vivado's placement-level timing estimation.

In some cases, our min-cost matching placement even achieves higher setup slacks than Vivado. This is because for the anchors at die boundaries, our min-cost matching formulation will always place them onto the die-crossing channels to balance the signal delays on two sides. However, Vivado often places the anchors outside the die-crossing channels as the timing target is still met.

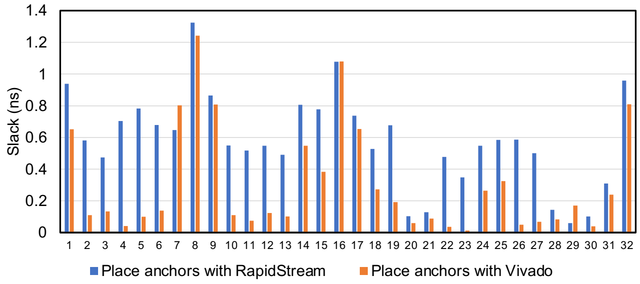


Figure 19: Post-placement slack between using the Vivado placer or the min-cost matching placer for anchor placement.

After we place all the anchors (iteration 2), we will perform local optimization of the island placement (iteration 3). We measure the setup slack of all nets from/to anchors to check the placement quality of our min-cost matching placement formulation. Based on Vivado's timing report at placement level, the average setup slack of anchor nets after iteration 2 is 0.55 ns (when targeting 2.5 ns or 400 MHz); and iteration 3 improves the average slack to 0.69 ns.

7.6 Clock Management

In this subsection, we demonstrate the advantages of preserving the clocking trunk using a number of experiments with the MM benchmark. Figure 20 shows the timing degradation when we stitch the islands together and route the clock net afterward. In this case, we route each island without preserving the clock trunk. The router relies on an estimation of the clock skew when routing the data signals. As a result, the actual clock skew after stitching may be different. As seen in the figure, all islands will run into hold violation after stitching, and setup/hold slack times deteriorate by about 0.25 ns for the islands, which will almost always cause hold violations.

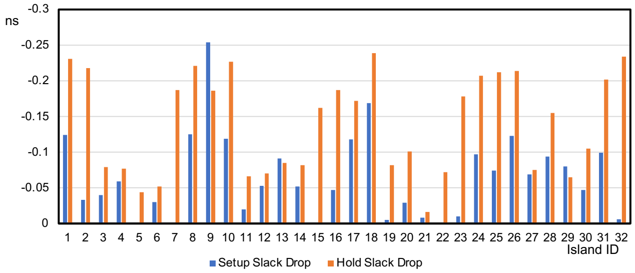


Figure 20: Timing loss after stitching w/o clock management.

Figure 21 shows the setup slack differences when an island is routed with preserved clock trunk. This is compared to the reference case used in Figure 20 without any clocking constraints. The setup slack drop is at most 0.15 ns, which is much smaller than that of Figure 20. The key is that we will not suffer from setup/hold loss in the stitching process as we keep the clock consistent.

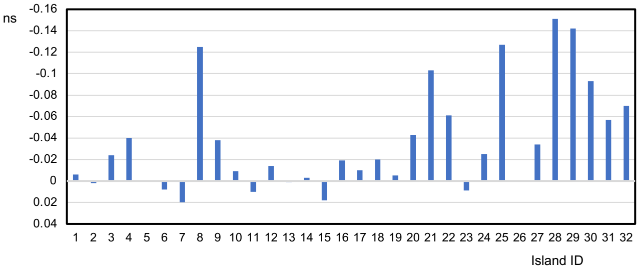


Figure 21: Clock preservation reduces timing degradation.

8 RELATED WORK

Split Compilation for HLS Designs enjoys the flexibility to add additional pipelining as needed, which contrasts split compilation methods for *RTL* designs (Section 1).

Previous efforts on HLS-level split compilation are based on pre-building a fixed *static region* to divide the FPGA into islands. The static region includes pre-placed and pre-routed logic that remains unchanged. Then, a design is divided and mapped onto those disjoint islands. [51, 63] pre-build an NoC based on partial reconfiguration [67], as shown in Figure 22. However, they suffer from the area overhead and the limited bandwidth of the NoC. Recently, researchers propose FPGA virtualization [71–73], which also relies on pre-building a static region to form disjoint islands.

DW [62] reduces the area overhead and their static region only consists of a set of *partition pins*, which are pre-routed wire segments at the boundary of two adjacent islands. However, DW needs users to manually change the design and map inter-island nets to partition pins. Meanwhile, as the number and distribution of partition pins are fixed in advance, they will affect timing closure. DW reports a Fmax of 187 MHz, while we achieve close to 400 MHz.

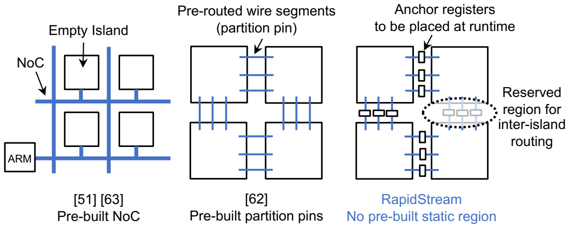


Figure 22: Comparison between previous works and RapidStream.

Soft Cores with NoC. Many other flows implement an array of soft processors on the FPGA [3, 4, 16, 32, 33, 61, 70], then connect the processors by programmable NoC [24, 31, 37, 50, 58]. In comparison, we target high-performance application-specific implementation.

Acceleration Based on Hard Macros. Researchers have explored acceleration utilizing pre-implemented hard macros [17, 25, 40, 46, 49, 74]. Hard macros consist of pre-built circuitry and can be reused. However, this approach may only cover a very limited part, if any, of an arbitrary input design. In addition, the fixed shapes of pre-determined macros may result in area waste and designers need to customize the macros for different devices.

Co-optimizing HLS and Physical Design. Guo et. al. [28] couples floorplanning with HLS synthesis to pipeline the global data transfer logic. RapidStream (in S_2) also adopts the iterative partitioning floorplan algorithm. AutoBridge and other works that co-optimize physical design process and HLS compilation [14, 29, 57, 75–77] rely on the conventional RTL-to-bitstream tool chain.

Dataflow Designs. RapidStream targets the dataflow design pattern, which has been well studied in theory [5, 41] and has been applied in a rich set of application domains, including linear algebra [55, 60], graph processing [8, 12, 13], image processing [11, 78], sorting [52, 53] and many more. Recently, HLS tools with dynamic scheduling [9, 10, 35] are gaining popularity. They introduces elastic components like FIFOs to enable a dataflow-style execution, which could potentially be utilized by RapidStream as well.

9 CONCLUSION

RapidStream is an automated split compilation flow for HLS dataflow designs. It features tight integration of HLS-level pipelining and physical design automation to enable split compilation while maintaining a high timing quality. Compared to a commercial tool chain, RapidStream achieves about 5-7 × reduction in compile time and up to 1.3× increase in frequency for HLS dataflow designs. In addition, our results show potential for up to an order of magnitude speed-up by leveraging customized open-source routers.

ACKNOWLEDGMENTS

We thank Dina G. Mahmoud for help in the artifact evaluation. This work is partially supported by the CRISP Program, the CDSC Industrial Partnership Program, the Xilinx Adaptive Compute Clusters (XACC) Program. We thank Gurobi and GNU Parallel for academic licenses. The opinions expressed by the authors are theirs alone and do not represent future Xilinx policies.

REFERENCES

[1] Matthew An, J Gregory Steffan, and Vaughn Betz. 2014. Speeding up FPGA placement: Parallel algorithms and methods. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.

[2] Melvin A Breuer. 1977. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*. 284–290.

[3] Davor Capalija and Tarek S Abdelrahman. 2011. Towards synthesis-free JIT compilation to commodity FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 202–205.

[4] Davor Capalija and Tarek S Abdelrahman. 2013. A high-performance overlay architecture for pipelined execution of data flow graphs. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 1–8.

[5] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on computer-aided design of integrated circuits and systems* 20, 9 (2001), 1059–1076.

[6] Tony Chan, Jason Cong, and Kenton Sze. 2005. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the 2005 international symposium on physical design*. 185–192.

[7] Chandra Chekuri. 2010. https://courses.engr.illinois.edu/cs598csc/sp2010/Lectures/Lecture11.pdf

[8] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[9] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 288–298.

[10] Jianyi Cheng, Lana Josipović, George A Constantinides, Paolo Ienne, and John Wickerson. 2021. DASS: Combining Dynamic and Static Scheduling in High-level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[11] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[12] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for Power-Law Graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[13] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 204–213.

[14] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality aware transformation for high-level synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 125–128.

[15] Jason Cong and Yi Zou. 2009. Parallel multi-level analytical global placement on graphics processing units. In *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*. IEEE, 681–688.

[16] James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 13–22.

[17] James Coole and Greg Stitt. 2012. BPR: fast FPGA placement and routing using macroblocks. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 275–284.

[18] Kaushik De and Prithviraj Banerjee. 1994. Parallel logic synthesis using partitioning. In *1994 International Conference on Parallel Processing Vol. 3*, Vol. 3. IEEE, 135–142.

[19] Kaushik De, LA Chandy, Sumit Roy, Steven Parkes, and Prithviraj Banerjee. 1995. Parallel algorithms for logic synthesis using the MIS approach. In *Proceedings of 9th International Parallel Processing Symposium*. IEEE, 579–585.

[20] Shounak Dhar, Love Singhal, Mahesh Iyer, and David Pan. 2019. FPGA Accelerated FPGA Placement. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 404–410.

[21] Xiao Dong and Guy GF Lemieux. 2009. PGR: Period and glitch reduction via clock skew scheduling, delay padding and GlitchLess. In *2009 International Conference on Field-Programmable Technology*. IEEE, 88–95.

[22] Alfred E Dunlop, Brian W Kernighan, et al. 1985. A procedure for placement of standard cell VLSI circuits. *IEEE Transactions on Computer-Aided Design* 4, 1 (1985), 92–98.

[23] John P. Fishburn. 1990. Clock skew optimization. *IEEE transactions on computers* 39, 7 (1990), 945–951.

[24] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

[25] Marcel Gort and Jason Anderson. 2014. Design re-use for compile time reduction in FPGA high-level synthesis flows. In *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, 4–11.

[26] Marcel Gort and Jason H Anderson. 2010. Deterministic multi-core parallel routing for FPGAs. In *2010 International Conference on Field-Programmable Technology*. IEEE, 78–86.

[27] Marcel Gort and Jason H Anderson. 2011. Accelerating FPGA routing through parallelization and engineering enhancements special section on PAR-CAD 2010. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 1 (2011), 61–74.

[28] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[29] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *57th ACM/IEEE Design Automation Conference*. https://doi.org/10.1109/DAC18072.2020.9218718

[30] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A Parallel Deterministic Router Based on Spatial Partitioning and Scheduling. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/3174243.3174246

[31] Yutian Huan and André DeHon. 2012. FPGA optimized packet-switched NoC using split and merge primitives. In *2012 International Conference on Field-Programmable Technology*. IEEE, 47–52.

[32] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[33] Abhishek Kumar Jain, Khoa Dang Pham, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. 2014. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems* 77, 1 (2014), 61–76.

[34] Wei Jiang, Zhiru Zhang, Miodrag Potkonjak, and Jason Cong. 2008. Scheduling with integer time budgeting for low-power optimization. In *2008 Asia and South Pacific Design Automation Conference*. IEEE, 22–27.

[35] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 127–136.

[36] Parivallal Kannan and Satish Sivaswamy. 2016. Performance driven routing for modern FPGAs. In *Proceedings of the 35th International Conference on Computer-Aided Design*. 1–6.

[37] Nachiket Kapre and Jan Gray. 2017. Hoplite: A deflection-routed directional torus noc for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 2 (2017), 1–24.

[38] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14, 4 (2021), 1–39.

[39] 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)*. IEEE, 133–140.

[40] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. 2011. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 117–124.

[41] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[42] Wuxi Li, Meng Li, Jiajun Wang, and David Z Pan. 2017. UTPlaceF 3.0: A parallelization framework for modern FPGA global placement. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 922–928.

[43] Tao Lin, Chris Chu, and Gang Wu. 2015. POLAR 3.0: An ultrafast global placement engine. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 520–527.

[44] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. 2008. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. 14–23.

[45] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. 7, 2 (2014). https://doi.org/10.1145/2617593

[46] Sen Ma, Zeyad Aklah, and David Andrews. 2016. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–178.

[47] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. 2003. Fast timing-driven partitioning-based placement for island style FPGAs. In *Proceedings of the 40th*

annual design automation conference. 598–603.

[48] Pongstorn Maidee, Chris Neely, Alireza Kaviani, and Chris Lavin. 2019. An Open-source Lightweight Timing Model for RapidWright. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 171–178.

[49] Fubing Mao, Wei Zhang, Bingsheng He, and Siew-Kei Lam. 2017. Dynamic module partitioning for library based placement on heterogeneous FPGAs. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1–6.

[50] Michael K Papamichael and James C Hoe. 2012. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. 37–46.

[51] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for fast FPGA compilation using partial reconfiguration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 235–2353.

[52] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-Accelerated Near-Storage Sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 106–114.

[53] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-Performance Adaptive Merge Tree Sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 282–294.

[54] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 118–125.

[55] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2021. Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication. *arXiv preprint arXiv:2111.12555* (2021).

[56] Mirjana Stojilović. 2017. Parallel FPGA routing: Survey and challenges. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[57] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. 2015. Mapping-aware constrained scheduling for LUT-based FPGAs. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 190–199.

[58] Kizhepatt Vipin, Jan Gray, and Nachiket Kapre. 2017. Enabling partial reconfiguration and low latency routing using segmented FPGA NoCs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[59] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. 2017. A runtime optimization approach for FPGA routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2017), 1706–1710.

[60] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*.

[61] David Wilson and Greg Stitt. 2019. Seiba: An FPGA Overlay-Based Approach to Rapid Application Development. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[62] 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)*. IEEE, 196–205.

[63] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon. 2019. Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 153–161.

[64] Xilinx. 2020. Xilinx UltraScale Plus Architecture. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[65] Xilinx. 2021. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug905-vivado-hierarchical-design.pdf

[66] Xilinx. 2021. https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf

[67] Xilinx. 2021. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf

[68] Zhen Yang, Anthony Vannelli, and Shawki Areibi. 2007. An ILP based hierarchical global routing approach for VLSI ASIC design. *Optimization Letters* 1, 3 (2007), 281–297.

[69] Chao-Yang Yeh and Malgorzata Marek-Sadowska. 2005. Skew-programmable clock design for FPGA and skew-aware placement. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. 33–40.

[70] Michael Xi Yue, Dirk Koch, and Guy GF Lemieux. 2015. Rapid overlay builder for xilinx fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 17–20.

[71] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–858.

[72] Yue Zha and Jing Li. 2021. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 470–483.

[73] Yue Zha and Jing Li. 2021. When application-specific ISA meets FPGAs: a multilayer virtualization framework for heterogeneous cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 123–134.

[74] Niansong Zhang, Xiang Chen, and Nachiket Kapre. 2020. Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithms. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 145–152.

[75] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1130–1135.

[76] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. 2015. Area-efficient pipelining for FPGA-targeted high-level synthesis. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.

[77] Hongbin Zheng, Swathi T Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and effective placement and routing directed high-level synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 1–10.

[78] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. 2018. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 269–278.

[79] Yun Zhou, Pongstorn Maidee, Chris Lavin, Alireza Kaviani, and Dirk Stroobandt. 2021. RWRoute: An Open-source Timing-driven Router for Commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 1 (2021), 1–27.

[80] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. 2020. Accelerating FPGA Routing Through Algorithmic Enhancements and Connection-Aware Parallelization. *ACM Trans. Reconfigurable Technol. Syst.* 13, 4, Article 18 (Aug. 2020), 26 pages. https://doi.org/10.1145/3406959