Appearing in *IEEE International Conference on Field-Programmable Technology* (FPT 2019), December 11–13, 2019

# Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks

Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han,
Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: ylxiao@seas.upenn.edu, dopark@seas.upenn.edu, giesen@seas.upenn.edu, zhhan@coe.neu.edu,
nmagnezi@ucsd.edu, andre@ieee.org

***Abstract*—Today's FPGA compilation is slow because it compiles and co-optimizes the entire design in one monolithic mapping flow. This achieves high quality results but also means a long edit-compile-debug loop that slows development and limits the scope of design-space exploration. We introduce *PRflow* that uses partial reconfiguration and an overlay packet-switched network to *separate* the HLS-to-bitstream compilation problem for individual components of the FPGA design. This separation allows both incremental compilation, where a single component can be recompiled without recompiling the entire design, and parallel compilation, where all the components are compiled in parallel. Both uses reduce the compilation time. Mapping the Rosetta Benchmarks to a Xilinx XCZU9EG, we show compilation times reduce from 42 minutes to 12 minutes (one case from 160 minutes to 18 minutes) when running on top of commercial tools from Xilinx. Using Symbiflow (Project X-Ray/Yosys/VPR), we show preliminary evidence we can further reduce most compile times under 5 minutes, with some components mapping in less than 2 minutes.**

## I. Introduction

The performance and energy benefits of FPGAs are propelling them into larger scale use for embedded, server, and cloud computing. Nonetheless, the exploitation of FPGAs significantly lags behind their potential benefits. Despite increasing opportunities to describe FPGA designs at higher-levels of abstraction with High-Level Synthesis (HLS), it can still take hours to compile a design from C to Verilog to gates and ultimately to a bitstream that can be loaded onto the device (HLS-to-bitstream compile). This long compile time serves as a deterrent and disincentive to adopt and embrace FPGAs. This compile time is off-putting for software designers accustomed to debugging turns of seconds to minutes, making multicore and GPU solutions attractive even when they are slower and burn more energy. Furthermore, to find good FPGA solutions, experts will need to carefully explore the design space for potential solutions. However, when faced with hour-long compilation times, even experts must limit the scope of their design-space exploration leading to sub-optimal solutions that do not fully exploit the benefits of the FPGA. *Slow compilation presents a barrier that is limiting the adoption and exploitation of FPGA computing.*

One of the key reasons FPGA compilation is slow is the monolithic compilation and global optimization of the entire design. With FPGAs that hold millions of 6-LUTs and thousands of independent DSP blocks and embedded RAMs, the sheer size of the compilation problem is large. Park [1] suggests it is possible to decompose this large compilation task into many, smaller pieces that can be compiled separately. Separate compilation allows an easy form of data-level parallelism. Furthermore, separate compilation isolates the impact of changes, preventing the need to recompile pieces of the design that have not been changed.

Specifically, Park uses partial reconfiguration and a packet-switched overlay network to isolate design components for separate compilation, dividing the FPGA capacity into a number of independent, partial reconfiguration regions. Components (IP blocks, computational operators) can be compiled to partial reconfiguration regions independently. The packet-switched network provides the communication linking that allows the independently-compiled blocks to interact.

We develop and evaluate a concrete instance of this vision. *PRflow* is an architecture and tool for the divide-and-conquer approach that works for Virtex UltraScale Plus FPGAs on top of the Vivado HLS and Vivado tool suites (Sec. V). The stock commercial tools have a number of limitations that make this usage challenging and that reduce the magnitude of the benefit (Sec. IV). Nonetheless, we show how to make use of the functionality available. We also explore the use of Project X-Ray/Yosys/VPR for synthesis and physical mapping and show preliminary evidence this may allow even faster mapping times (Sec. VI). We make the following contributions:

1) Characterization of Vivado compilation times to optimize a fast, separate compilation design point (Sec. IV)
2) Tool flow that operates on top of Vivado to automate the separate compilation strategy (Sec. V)
3) Tool flow that operates on top of Yosys and VPR to automate synthesis, place, and route (Sec. VI)
4) Characterization of the mapping time benefits and performance impact on the Rosetta Benchmark suite (Sec. VII)

## II. Background

### A. Out of Context

Vivado already supports separate HLS and Verilog synthesis using Out-of-Context (OoC) design flow [2]. This can be used by design groups to separately develop and synthesize IP and compute blocks that can later be assembled for monolithic place and route. Vivado will use multiple processes on a single workstation or server to exploit some parallelism in module synthesis. We incorporate this capability into our design flow and use it explicitly to allow separate compilation of leaf blocks, including spawning the tasks to separate computers in the cloud (Sec. V-A).

### B. Partial Reconfiguration

Partial Reconfiguration (PR) is an architectural feature and FPGA usage strategy that allows a subset of the FPGA resources to be reconfigured separately from the rest of the FPGA. This allows portions of the functionality deployed on an FPGA to change while leaving other pieces in place. Done properly, the non-changing portion of the FPGA can continue to operate during the reconfiguration [3], [4]. The partial bitstream can be smaller than the full bitstream, meaning the time to load the partial bitstream can be smaller than the full bitstream, roughly in proportion to the size of the reconfigured region to the full FPGA.

The standard Xilinx design discipline for PR divides the FPGA resources into a single static region and one or more PR regions [5], [6]. The static region is the part that will never change, while each PR region can be loaded separately. To define each PR region, the developer identifies a specific set of physical resources as a *p-block*. These p-blocks identify the physical regions that logic can target and that can be reconfigured. When mapping the logical design, the application designer identifies a portion of the design as a *reconfigurable partition*. Each reconfigurable partition can then be assigned to a reconfigurable p-block. Significantly for our purposes, by constraining portions of the design (reconfigurable partitions) to specific parts of the FPGA chip (reconfigurable p-blocks) and using standard interfaces between the partitions, we can isolate the design components (partitions) from each other so that they can be compiled separately.

Conventional usage of PR have targeted area reduction [7], [8], [9], load-time reduction, and design specialization [10], [11]. Beyond [1], previous work has not directly targeted the reduction of compilation time as we do in this work.

### C. Pre-Compiled Macros

Previous work has demonstrated that pre-implemented large macroblock components can be an effective approach to reducing FPGA compilation time. HMflow exploited pre-compilation of hard (internally, pre-placed and routed) macros, macroblock floorplanning, and custom routing to quickly assemble DSP designs from Xilinx's System Generator [12], [13]. Once placed and routed, HMflow uses vendor tools for final bitstream generation.

Just-in-Time Assembly of Accelerators (JITA) similarly pre-computes the compilation for components so that their compilation time is removed from the critical path to produce a user design [14]. JITA uses 9600 LUT p-blocks with a configurable overlay network to avoid vendor placement, routing, and bitstream generation. JITA's overlay network is a static, nearest-neighbor design similar to FPGA switchboxes except that it routes word-wide data and is configured on top of the FPGA.

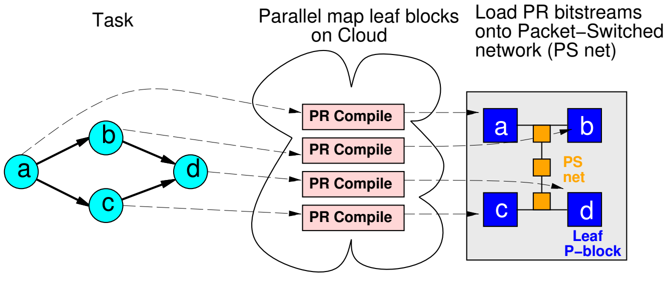

Task
Parallel map leaf blocks on Cloud
Load PR bitstreams onto Packet–Switched network (PS net)
a
b
c
d
PR Compile
PR Compile
PR Compile
PR Compile
a
b
PS net
c
d
Leaf P–block

Fig. 1. Separate Compilation Strategy

We build upon both of these ideas, taking the next step to allow and compile any leaf logic component rather than limiting leaf blocks to a pre-defined set of pre-compiled blocks. Using a packet-switched overlay network, we completely eliminate the routing time to wire between leaf functions as well as the possibility of routing failure.

### D. Pre-defined and Pre-Sized Regions

Cutting the FPGA up into separate regions with pre-defined sizes is a known technique for separating concerns and simplifying design composition. This allows designs to be separately and independently compiled to defined regions then composed at the coarser-grain region level. This technique has been used with PR regions in the past. Brebner introduces Swappable Logic Units [15]. SCORE introduces logic pages that could be separately implemented, reconfigured, and managed at runtime [16]. While not fully exported to the application developer, the Stratix-10 architecture internally has a notion of configuration regions for clocking and partial reconfiguration [17].

## III. Idea and Basic Design

The basic idea of a separate compilation methodology is to divide the FPGA capacity into a set of regions of pre-defined size and separately map blocks in the user's application to the regions (See Fig. 1) [1]. We call the physical partial reconfiguration regions for user logic *leaf p-blocks* and the logic we map separately into these leaf p-blocks *leaf blocks*. Once mapped, the partial bitstream for each leaf block can be loaded onto the FPGA. A packet-switched overlay network provides connectivity between the leaf p-blocks.

### A. Compute Model

We assume a dataflow streaming compute model [18], [16], [19] to assemble and coordinate the application. The design is decomposed as a network of computation and memory operators interconnected by dataflow stream links. The dataflow stream links abstract away implementation and timing, so that operators do not need any knowledge about the internal implementation of the operators they communicate with nor the communication channels among them. One or more operators are grouped together into a leaf block for mapping. The dataflow stream links support the separate compilation of these leaf blocks and the variable delay in interconnect among them that arises both from leaf p-block placement choices and from congestion on the packet-switched network.

In the current work, we demand that the operators in the user design be smaller than the user logic capacity in the leaf p-blocks in the target platform. Automated decomposition of user logic into leaf blocks is complementary future work.

### B. *Packet-Switched Network*

We use a packet-switch network so that there is no need to synthesize, place, and route any unique connectivity to interconnect the separately-compiled leaf blocks. The leaf blocks are simply configured with the destination addresses on the network for the downstream modules, and leaf interface logic attaches the address to data as it enters the network. We use a deflection-routed, packet-switch network since modern designs show how they can be lightweight [20]. We specifically use a Butterfly Fat Tree (BFT) topology [21], [22] that can be parameterized and tuned to provide different levels of interconnect according to Rent's Rule [23].

### C. *Network Interface*

The user operator logic communicates through streaming input and output ports, either HLS streams or AXI Lite streams. To connect user module operators to the deflection-routed, packet-switched BFT, the leaf blocks include leaf interface logic as shown in Fig. 2. This leaf interface includes logical stream FIFOs that receive the packets from BFT and send the packets out to the BFT. These FIFOs decouple the user module from BFT so that user logic and BFT can run at different clock frequencies. The minimal complexity deflection-routed BFT neither deals with flow-control for streams nor guarantees in-order transfer of the packets to the destination leaf block [20], [22]. We add sequence numbers to packets for ordering, and we design the leaf-interface logic to store data into FIFOs in order and deal with flow control using a windowed acknowledgment scheme [24]. Because the leaf interface resides in the reconfigurable leaf block, the composition of the leaf interface can be tuned to the user module. For instance, the depth and throughput of the FIFOs and the number of input and output FIFO streams can vary based on the requirements of the user module. Our packets include a destination leaf and port identifier, an address for sequencing, and the payload. In this work, we use a 48-bit packet with 5b for leaf identifier, 4b for port, 7b for address, and 32b for payload. On the network, a 49th bit is used for flow control. The 32b network reduces the mapping time required by the static region (Sec. IV-C).

### D. *Management, Processor, and Memory Interface*

In addition to the leaf p-blocks, we also place embedded hard processors and memory interfaces on the BFT. On Zynq devices, one or more leaves hold the embedded ARM processor(s). These processors can serve as operators in the dataflow network, using the same dataflow streaming interfaces. We also use one ARM as a configuration controller to manage leaf p-block loading and configuration of the leaf blocks. Leaves with DMA interfaces to the DRAM controller(s) provide the interface to DRAM.

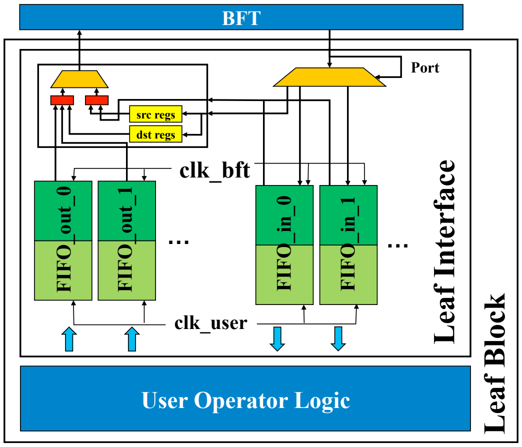


Fig. 2. Leaf Block Composition and Interface

## IV. Vivado Characteristics

Intuitively, we expect that the time to perform physical implementation mapping (placement and routing) will be driven by the number of logical design elements (LUTs, BRAMs, DSPs, nets) in the netlist and perhaps the number of physical elements to which they can be mapped (LUTs, BRAMs, DSPs on the FPGA or in a p-block). All the mapping problems are NP-hard and typical algorithms are heuristics. So, we might expect, at least, linear scaling with the number of design or physical elements. By making leaves small compared to the size of the FPGA (e.g., maybe 1% or 2% of the size), we would hope to reduce mapping time commensurately.

We will see that Vivado is not designed to completely deliver the full benefits. In this section, we characterize the rough behavior of Vivado and use that to guide the design of our PRflow mapping strategy. For the experiments in this section we use Vivado 2018.2 running on a compute server with two 2.7 GHz Intel E5-2680 CPUs and 128 GB of RAM.

### A. *Leaf Size*

For a fixed-size FPGA, when the resource requirements are not too close to 100% utilization, we see (Tab. I) that the mapping time is driven by the number of design elements in the netlist and not by the number of physical resources available (e.g., number of LUTs in a p-block). Specifically, as long as the design size is sufficiently smaller than the p-block size, increasing the size of the design increases the implementation time. However, keeping the design size fixed and increasing the size of the p-block does not have a big effect on mapping time. For this experiment, we use two designs; one is a large shift-register, where we vary the length of the shift register, and the second has a variable number of connected MicroBlaze [25] cores. Nonetheless, a large portion of the implementation time is the fixed cost of starting Vivado

TABLE I
IMPLEMENTATION TIME VS. DESIGN AND P-BLOCK SIZE ON XCZU9EG

| | | **P-Block Size** (LUTs) | | | | |
|---|---|---|---|---|---|---|
| **Design** | **Size** | 3960 | 6160 | 7920 | 10120 | 15840 |
| Shift Register | 623 | 203 | 206 | 206 | 205 | 204 |
| | 1633 | 210 | 210 | 210 | 208 | 210 |
| | 2661 | 220 | 218 | 218 | 217 | 217 |
| | 3614 | 229 | 233 | 224 | 227 | 225 |
| | 4616 | | 239 | 239 | 234 | 237 |
| | 5623 | | 239 | 244 | 241 | 242 |
| MicroBlaze Cores | 1435 | 182 | 181 | 180 | 181 | 185 |
| | 2860 | 196 | 192 | 195 | 192 | 198 |
| | 4285 | | 210 | 211 | 210 | 207 |
| | 5710 | | 605 | 231 | 223 | 226 |

(cells show compilation time in seconds)

TABLE II
STATIC REGION IMPACT ON IMPLEMENTATION TIME (32 LEAF BFT WITH 32B PAYLOAD WIDTH DATAPATH)

| | PR implementation | | OoC impl. |
|---|---|---|---|
| | BFT in Static | BFT as P-block | leaf only |
| LUTs in mapping | 30611 | 8590 | 1435 |
| optimize time (s) | 29 | 10 | 79 |
| place time (s) | 238 | 161 | 27 |
| route time (s) | 170 | 113 | 74 |
| total time (s) | 437 | 284 | 180 |

and loading the device database. This is one limitation on our ability to achieve ideal speedups with Vivado.

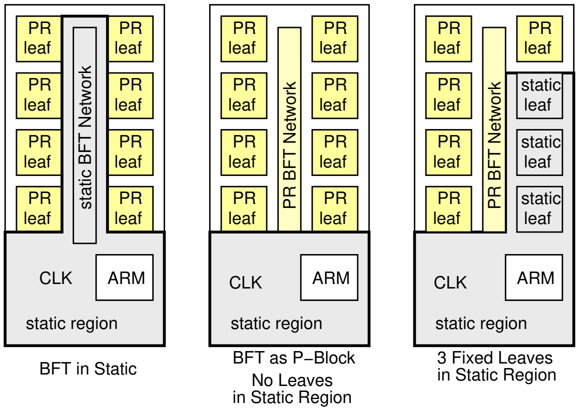
### B. Partial Reconfiguration Compile

Vivado PR implementation time includes the static region and all the PR blocks. Implementation takes time for elements in the static region even though the static region is constrained and there are no decisions to make on where to place its elements and how to route its nets; Vivado still spends time loading and validating the logic and connections. If we are only mapping a single reconfigurable p-block, we can assign minimal dummy logic to the p-blocks we are not currently mapping, but Vivado demands inclusion of the full static logic. To illustrate this phenomenon, Fig. 3 shows an experiment where we vary the size of the static region by varying the number of leaves on the BFT that are included in the static region. For this experiment, we place a 1522 LUT MicroBlaze processor [25] in each of the leaves added to the static region. In all cases, there is only one non-dummy reconfigurable p-block being mapped. As we see, the implementation time increases with the size of the static region. As another example, in Tab. II, we look at the time to perform an OoC implementation of a reconfigurable p-block that will only synthesize, place, and route the leaf p-block compared to the time of a PR mapping along with the static region. We see that the PR mapping ("BFT in Static" column) takes longer than the leaf p-block compiled in isolation.

As a result, the time for a leaf implementation is:

$$T_{prmap} \approx T_{leaf} + T_{static} \qquad (1)$$

| Fixed Leaves in Static Region | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
|---|---|---|---|---|---|---|---|---|
| Implementation Time for Non-Dummy Leaf (seconds) | 288 | 323 | 347 | 385 | 411 | 458 | 494 | 526 |



BFT in Static
BFT as P–Block No Leaves in Static Region
3 Fixed Leaves in Static Region

Fig. 3. Impact of Static Region Composossition on PR Leaf Mapping Time

TABLE III
PARTIAL RECONFIGURATION MAPPING TIMES VS. LEAVES AND WIDTH

| | | Time (seconds) | | |
|---|---|---|---|---|
| $N_{leaf}$ | Width | $T_{prmap}$ | OoC | $T_{static}$ |
| 8 | 48 | 308 | 220 | 88 |
| | 72 | 297 | 224 | 73 |
| | 96 | 306 | 226 | 80 |
| 16 | 48 | 333 | 220 | 113 |
| | 72 | 336 | 224 | 112 |
| | 96 | 346 | 226 | 120 |
| 32 | 48 | 399 | 220 | 179 |
| | 60 | 412 | 226 | 186 |

### C. Optimizing Partial Reconfiguration Compile Time

We can reduce the static time by moving all the logic (even the logic we intended not to change among applications) into reconfigurable p-blocks. Specifically, we can move the BFT into one or more reconfigurable p-blocks [1]. Tab. II also shows how this reduces the PR implementation time.

Once the BFT is placed in reconfigurable p-blocks, what is left is a small amount of global wiring (e.g., clocks) and the connections among the partially reconfigurable blocks. This is mostly the connections between the BFT and the leaves. The BFT leaf connection nets are proportional to the number of leaves and the width of the network. There will be some time to map the minimal logic in each of the leaves and some fixed time to load the chip database and deal with global resources, so static time increases with the number of leaves and, to a lesser extent, with the width of the network (Tab. III). We select a configuration ($Width$=48, $N_{leaf}$≈32) where $T_{static}$ is roughly 180 seconds so that it does not dominate $T_{leaf}$.

## V. TOOL SUPPORT

As shown in Fig. 4, we add three tools for design preparation and compilation coordination around the Vivado HLS and Vivado synthesis and implementation routines.

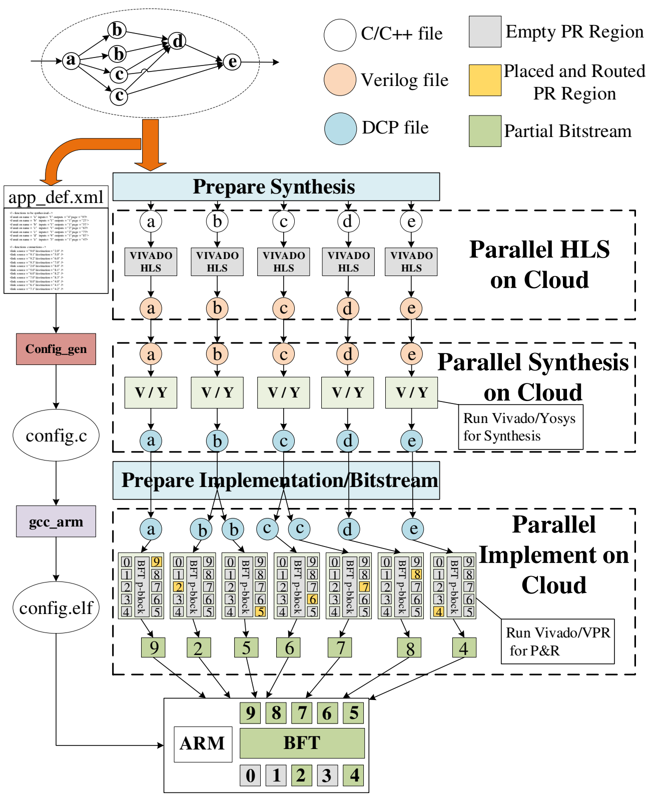


Fig. 4. PRflow Separate Compilation Flow

### A. Prepare Synthesis

Designs start with the user logic for operators expressed in C or Verilog with stream inputs and outputs. In the first phase, we use separate OoC synthesis runs to compile each of these into a netlist design checkpoint (.dcp) file. We require a top level application definition file in XML (`app_def.xml`) that identifies all the source files for the operators that need compilation and the dataflow connectivity among the operators. Our tool sets parameters and generates the appropriate leaf interface (Sec. III-C) for the operator based on its needs (e.g., number of input and output streams). It creates a custom leaf block wrapper for each operator including both the user operator logic and the selected leaf interface. It generates TCL scripts to control synthesis for each operator leaf block. Finally, it spawns each of the synthesis runs to the cloud server. Overall operation is coordinated with a Python script.

### B. Prepare Implementation

After the leaf blocks have been synthesized to netlist design checkpoints, our second phase directs the separate physical implementation of each leaf block to a particular leaf p-block. Based on a leaf assignment from the application definition file, a leaf implementation tool packages up the design for implementation, including setting the p-blocks that are not being mapped in this separate compilation to dummy designs. It creates the TCL scripts to control the implementation runs and spawns the implementation runs to cloud servers. The result of the synthesis is a partial reconfiguration bitstream for the target leaf p-block.

### C. Generate Configuration (config_gen)

Concurrent with leaf p-blocks implementation runs, we also produce and compile C code for the embedded ARM processor to link up the leaf blocks. A configuration generation tool takes in the application definition XML file that expresses the operator dataflow. For each stream link, it has the ARM processor send control messages over the BFT to configure the source and destination ports in the respective leaf interfaces so that they know each other's location and port identifiers to construct packets.

## VI. Project X-Ray/Yosys/VPR

We also explore using open-source tools for leaf RTL to bitstream mapping. We continue to use Vivado HLS for C to RTL (Verilog) mapping. We use Yosys [26], [27] for Verilog Synthesis and VPR for packing, placement, and routing [28] enabled by the Project X-Ray architecture files [29]. This has three advantages: (1) we can avoid the portion of time Vivado spends on static logic, (2) inspired by Maverick [30], we can build an architecture file for a single leaf p-block to reduce device loading time,[1] and (3) we can tune the open-source tool parameters aggressively for low mapping times. Since Project X-Ray currently only supports Artix-7 devices, we map to comparably sized leaf p-blocks of (11,916 LUTs, 20 36Kb BRAMs, 40 DSPs) or (16,720, 30, 60) on an XC7A50T for our mapping time estimates. Since DSP support is still under development, we implemented a minimal mapping that simply used each DSP block as an 18×25 multiplier. Since Project X-Ray currently supports only a single clock, we use a single, unified BFT and user clock for this mapping. As a consequence of these limitations, we have not produced bitstreams to run on actual devices with this flow. We see these results as illustrating how it is possible to reduce mapping time with appropriate tool design.

## VII. Demonstration

### A. Methodology

To characterize the impact of the PRflow approach, we use the Rosetta Benchmark suite [31] as a set of illustrative designs. We map to the UltraScale+ Zynq XCZU9EG FPGA (274,080 LUTs, 548,160 FFs, 912 36Kb BRAMs, quad ARM Cortex-A53 processors), using Vivado, Vivado HLS, and SDSoC 2018.2. We perform mappings on a cluster of 8 compute servers, each with two 2.7 GHz Intel E5-2680 CPUs and 128 GB of RAM (total of 8 × 2 × 8=128 cores).

For comparison, we generate a number of baselines. We include two monolithic compile cases: one that uses the same decomposed blocks as the PRflow designs and includes all the logic in the final PRflow mapped designs, including the infrastructure BFT network and a second that directly connects

[1] We further reduce device load time by providing a binary alternative to the default XML architecture description.

blocks and does not decompose the design. The version with no BFT might represents a design without any of our infrastructure overhead. The version with the BFT illustrates the overhead it contributes. Since our compute model requires some decomposition that is different from the original SDSoC-targeted Rosetta Benchmark reference designs, we also include an SDSoC mapping to the same part where we only tuned the parameters to exploit our XCZU9EG target.

### *B. Design Point*

For the XCZU9EG, we use a 32-leaf BFT with p=0.5 and a datapath width of 48 to support 32b payloads. We split the BFT into 5 p-blocks (one root and four 8-leaf subtrees). At this size, the BFT costs 15,956 LUTs, or about 500 LUTs per endpoint. The network runs at 300 MHz. With one physical input and output from the network to a leaf, the peak leaf bandwidth is 1.2 GB/s each direction. One of the leaves is connected to an embedded ARM core for configuration. One is connected to the DRAM controller for memory access, which we run at 250 MHz with a 32b interface for a peak DRAM bandwidth of 1 GB/s. The leaf interface resource requirements scale with input ports, $I$, and output ports, $O$:

$$\text{Leaf Interface LUTs} \approx 206 + 66I + 227O \quad (2)$$

$$\text{Leaf Int. 36Kb BRAMs} = 1 + 2I + O/2 \quad (3)$$

We use a distribution of leaf p-block sizes (12×(5760 6-LUTs, 24 36Kb BRAMs, 48 DSPs), 4×(4800,12,72), 4×(4800,24,48), 2×(5760,12,72), 6×(6720,24,48), 1×(4320,12,48)), 1×(9120,36,48)). The total leaf-block logic capacity is 63% of the FPGA logic capacity. Assuming an average leaf I/O of 2 inputs and 2 output ports, this means 54% of the FPGA logic will be available for user operator logic. The smallest leaf p-block is 1.6% of the FPGA logic capacity, and the largest leaf p-block is 2.4%.

### *C. Application Refinement*

We use the Rosetta Benchmark source from https://github.com/cornell-zhang/rosetta, commit ID 6bc38c0.

*a) 3D Rendering:* The stages in the rendering pipeline map directly to operators that can be placed in leaf blocks and connected into a streaming pipeline. Given the capacity on the ZU9EG, we build 2 parallel rendering pipelines and split the image into 2 regions, that can run concurrently.

*b) Digit Recognition:* The basic operator is a hamming distance calculator followed by a systolic sorter that keeps the $K$-minimum matches computed locally by that operator. Each operator gets 6 BRAMs to hold 512 training samples for comparison. We place two such operators in each leaf block, then use 18 parallel leaf blocks to perform the matching. A final leaf block combines the $18\times K$ minimum candidates from the final $K$ minimum values from each of the 18 hamming-sort operators and performs voting.

*c) SPAM Filtering:* This basically performs a dot-product against the identified features. Features weights are kept in BRAMs in leaf pages and data for classification is streamed from DRAM. We use 8 pages to perform dot products and a total of 20 pages.

*d) Optical Flow:* The original optical flow Rosetta Benchmark was already decomposed into nine operators that fit into our streaming flow. The “Weight” and “Tensor” modules required more DSPs than available in any of our leaf pages and the “Compute Flow” required more LUTs. Consequently, we decomposed these further by splitting along independent x, y, and z components. This gives us a design that fits on our leaf p-blocks, but the input bandwidth needed by the “Outer Product” exceeds the bandwidth provided by the single leaf connection into the BFT. Consequently, we decompose the “Outer Product” along with “Tensor” and “Compute Flow” operators so we could utilize bandwidth from multiple leaves. The resulting design uses 16 leaf blocks.

*e) Binarized Neural Network (BNN):* The original Rosetta Benchmark implementation of the BNN kept network weights in DRAM and read them into the FPGA one level at a time, performing the levels in series. We store all network weights in on-chip BRAM, but our preliminary version processes BNN stages in series as well, achieving no net speedup.

*f) Face Detection:* Face Detection was the most challenging to decompose into small leaf blocks with limited input and output bandwidth. It was important to keep the classifiers local to the windows image (e.g. Integral Image) in the same page. This required, splitting the windowed image in the Y-dimension across five leaf blocks, splitting the classifiers along with the windowed portions, and replicating the windowed image for different classifiers. Our decomposed design must serialize the communication of classifier partial sums between leaf blocks, resulting in a large communication bottleneck that does not exist in the monolithic design. More significant reorganization will be needed to reduce these bottlenecks.

We validated the PRflow revised designs using the original Rosetta Benchmark test suites. We mapped designs and test them on the ZCU102 development board containing an XCZCU9EG-ffvb1156-2-i.

### *D. Parallel Mapping Speed*

Fig. 5 shows the mapping time for 3D Rendering for the monolithically compiled design that matches our final design and the PRflow leaf mappings. We set jobs to 32 for the monolithic mappings to exploit the parallelism available on the 16 core, 2 hyperthread machines. The PRflow case exploits parallelism across multiple servers. We set each run to use a single job per run. For the Vivado Quick option, we use `-directive Quick` with `place_design` and `route_design`. Reported time for each phase in Fig. 5 for the PRflow is the maximum time of the parallel Vivado runs in the respective phase. Tab. IV includes the compile time data for all the benchmarks and includes mapping times for the original Rosetta Benchmark SDSoC compiles.

From Fig. 5 and Tab. V, we can see the compile time reduction and the major source of benefits. PRflow is able to exploit some additional parallelism in HLS and logic synthesis and significantly reduce optimization time. PRflow must spend time reading a DCP checkpoint for the static region and a netlist DCP for the synthesized leaf block, which adds about

TABLE V
ROSETTA BENCHMARK DETAIL COMPILATION TIME

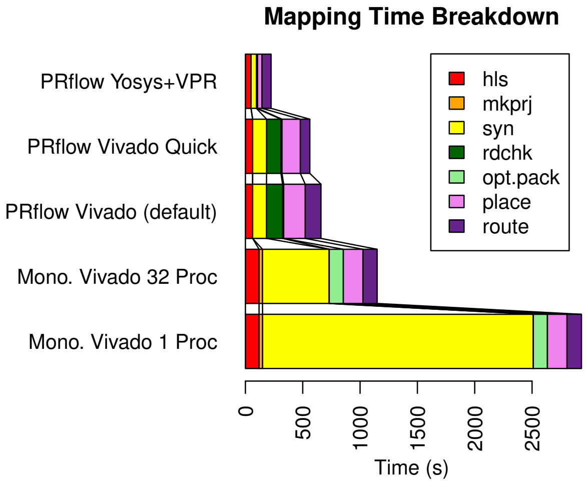
| | Monolithic (undecomposed), 32 Proc. | | | | | | | Vivado Quick PRflow | | | | | | | Yosys+VPR PRflow | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | hls | mkprj | syn | opt | place | route | total | hls | syn | rdchk | opt | place | route | total | hls | syn | pack | place | route | total |
| Digit Recognition | 145 | 32 | 752 | 144 | 297 | 377 | **1747** | 64 | 132 | 132 | 10 | 181 | 99 | **603** | 64 | 46 | 44 | 26 | 184 | **337** |
| SPAM Filtering | 90 | 33 | 592 | 144 | 178 | 161 | **1198** | 65 | 128 | 135 | 11 | 178 | 98 | **612** | 65 | 33 | 37 | 29 | 140 | **295** |
| 3-D Rendering | 119 | 30 | 582 | 123 | 171 | 124 | **1149** | 63 | 122 | 128 | 10 | 168 | 87 | **566** | 63 | 59 | 31 | 23 | 40 | **185** |
| Optical Flow | 179 | 52 | 1243 | 238 | 446 | 402 | **2560** | 126 | 176 | 132 | 15 | 201 | 131 | **724** | 126 | 39 | 71 | 36 | 115 | **311** |
| Binarized NN | 7487 | 33 | 1391 | 172 | 356 | 350 | **9789** | 199 | 336 | 142 | 12 | 205 | 112 | **1004** | †70 | †43 | †41 | †46 | †118 | †**309** |
| Face Detection | 526 | 32 | 1061 | 172 | 319 | 365 | **2475** | 174 | 307 | 132 | 13 | 179 | 113 | **918** | not map | | | | | |

Max component is computed per component, so max total shown is often lower than sum of component maximums. † omit bin_conv leaf.

TABLE VI
AREA AND PERFORMANCE COMPARISON

| | SDSoC | | | | Monolithic | | | | | | Vivado PRflow | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | no BFT (undecomposed) | | | with BFT | | | leaf | | |
| | par | Area | Fmax | per input | Area | Fmax | per input | Area | Fmax | per input | blocks | Fmax | per input |
| 3D Rendering | 2 | 8901,71,0 | 100MHz | 6.17ms | 11472,38,0 | 250MHz | 2.42ms | 43501,146,0 | 200MHz | 1.18ms | 12 | 250MHz | 1.18ms |
| Digit Recognition | 40 | 388685,306,1 | 100MHz | 13.00ms | 32563,177,0 | 215MHz | 6.70ms | 66636,318,0 | 250MHz | 16.58ms | 22 | 250MHz | 16.58ms |
| SPAM Filtering | 32 | 12754,73,224 | 100MHz | 82.13ms | 11827,37,224 | 250MHz | 126.52ms | 48904,216,256 | 250MHz | 49.10ms | 20 | 250MHz | 48.90ms |
| Optical Flow | | 38802,93,124 | 100MHz | 6.35ms | 78381,81,252 | 250MHz | 3.82ms | 59789,161,282 | 167MHz | 25.80ms | 16 | 250MHz | 25.80ms |
| Binarized NN | | 46165,599,3 | 100MHz | 5.30ms | 48753,591,3 | 150MHz | 3.56ms | 49985,582,6 | 250MHz | 14.63ms | 29 | 250MHz | 17.42ms |
| Face Detection | | 67791,134,79 | 100MHz | 28.19ms | 52885,85,79 | 188MHz | 17.16ms | 96763,338,108 | 150MHz | 351.93ms | 20 | 150MHz | 351.93ms |

Area given as (LUTs, BRAMs, DSPs). par = PAR_FACTOR; parallelism parameter in Rosetta Benchmark Source (or inserted for 3D Rendering)
per input – run time required for each image frame or dataset to characterize throughput achieved



hls = Vivado HLS; mkprj = create Vivado project; syn = Vivado Synthesis; rdchk = Read Checkpoint (DCP file); opt = logic optimization (for Vivado); pack = block packing for VPR; place = Placement; route = Routing

Fig. 5. Compilation Time Comparison for 3D Rendering

TABLE IV
ROSETTA BENCHMARK COMPILATION TIME (SECONDS)

| Design | SD SoC | Monolithic (undecomposed) | | PRflow | | |
|---|---|---|---|---|---|---|
| | | | | Vivado | | Yosys |
| | | 1 proc. | 32 proc. | | Quick | +VPR |
| Digit Recognition | 2472 | 3604 | 1747 | 638 | 603 | 337 |
| SPAM Filtering | 1770 | 2971 | 1198 | 658 | 612 | 295 |
| 3-D Rendering | 1769 | 2931 | 1149 | 659 | 566 | 185 |
| Optical Flow | 2660 | 4082 | 2560 | 744 | 724 | 311 |
| Binarized NN | 10726 | 11669 | 9789 | 1000 | 1004 | †309 |
| Face Detection | 4347 | 4352 | 2475 | 972 | 918 | — |

† omit bin_conv leaf.

two minutes to the flow; this cost is offset by about 30 seconds since we do not need to perform a `mkproj`. Physical optimization, place, and route takes about 300 seconds as we saw in Sec. IV; this is largely fixed time including device load and time checking the static region; the result is that we do not get a net speedup on physical mapping time on small designs like 3-D Rendering when using Vivado. Vivado Quick mapping saves 20–70 seconds. This leaves PRflow leaf mapping time fairly fixed around 600–700 seconds across all benchmarks (Tab. IV) except the largest leaf in BNN and face detect which take significantly more time in HLS and synthesis. Nonetheless, BNN and face detect see large parallelism gains for separated HLS compilation and synthesis. The monolithic mapping time increases for the larger benchmark, resulting in a speedup for PRflow that increases with design size.

The Yosys+VPR path allows us to reduce the fixed overheads and further accelerate physical mapping. It avoids the read checkpoint time and reduces device reading time. Once tuned for fast mapping, VPR packing, placement, and routing are typically 50–200 seconds, faster than the 300 seconds taken by Vivado. Together, this suggests mapping times of 2–5 minutes are possible to achieve. Currently, Yosys+VPR flow cannot map leaves with floating-point IP macros (two in optical flow, one in face detection), cannot route the largest leaf in BNN (bin_conv) and face detect (strong_classifier), and routes many of the face detect pages slower than Vivado.

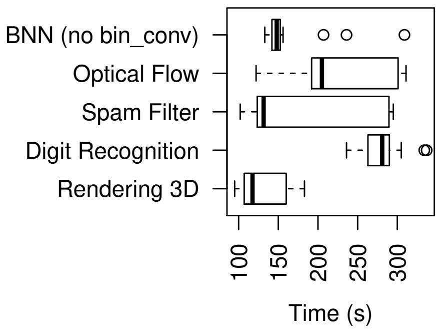


Fig. 6. Distribution of Yosys+VPR PRflow Mapping Times for Leaf Pages in Rosetta Benchmark Applications

This suggests these challenging leaf blocks should be further decomposed for Symbiflow fast mapping.

*Incremental Mapping:* A common application development strategy is to develop and test each operator one at a time, incrementally adding functionality to the design. Similarly, when debugging the complete design, a good strategy is to make small changes to one or a few operators at a time. Consequently, a common usage pattern is to change a single leaf operator and recompile. The PRflow times in Tab. IV report the worst-case time require to compile any of the leaves. Fig. 6 plots the distribution of mapping times showing most map faster than the worst case.

### *E. Application Performance*

Tab. VI shows the total resources and performance for the reference and Vivado PRflow cases. The PRflow case uses more resources due to its infrastructure overhead. We report the maximum clock frequency at which each design runs; for the PRflow case, the network runs at 300 MHz (200 MHz for face detect), and the table reports the leaf block frequency.

Our monolithic designs run at comparable or higher speeds to the original SDSoC Rosetta benchmarks, showing that our modifications remain competitive implemenations. The SDSoC cases are often frequency (Fmax) limited by their data mover interfaces. Half of our PRflow decomposed designs (3D Rendering, Digit Recognition, SPAM filtering) achieve the same performance as the monolithic versions. Optical flow runs at one-sixth the speed of the monolithic case and face detection one-twentieth due to the limited bandwidth between leaves over the BFT. Overall, this shows that the methodology supports high performance design even when exploiting separate component compilation. There is a higher premium for minimizing communication between components, but communication minimization is generally a good optimization target for all design mappings.

## VIII. Discussion and Future Work

It is possible to compile all the versions (monolithic and decomposed) from the original source. So, one view is that you quickly compile with the Yosys/VPR versions like a `-O1` compiler optimization in 2–5 minutes, use the Vivado option as a `-O2` option to get a result in 12 minutes, then use the monolithic version like a `-O3` option to get an integrated design when you can afford the hour long mapping time.

The Yosys/VPR mapping shows that it is possible to craft tools to accelerate physical mapping of reconfigurable p-blocks, but conventional tools are not optimized to do so. Hopefully, this work provides motivation for supporting such optimizations. Future work includes actual bitstream generation on this path as Project X-Ray support matures.

Once we remove the time taken for the static region, this decomposition solution is scalable. Leaving the size of the leaf block fixed and scaling up the number of leaves, the per leaf mapping time remains constant. We simply use more, parallel cloud resources to compile the design in the same time.

Relying directly on the vendor tools, we have accepted that each leaf p-block would be unique, both in size and in resources borrowed by the static region. This means we must compile each leaf block separately for each leaf p-block in which it may be placed. Existing work shows how to identify compatible regions for relocation [32] and how to keep static resources out of leaf p-blocks [33], [34].

To better handle a diversity of user designs with a potential variety of leaf block sizes, we expect it will be useful to stock a set of pre-compiled overlay architecture templates with different size trees and different size and distributions of p-blocks. We have already found it useful to stock overlay templates with larger pages to use during development before tightening or partitioning leaf blocks.

## IX. Conclusions

Hour-long compilation times make FPGA design untennable for many potential users and limits the quality of designs even experts can afford to develop. We show that FPGA compilation does not need to take hours. We can divide the FPGA compilation problem into independent pieces that can be mapped separately. The solution requires: (1) decomposition of the design into separate modules, (2) decomposition of the FPGA into separate physical regions, (3) adaptation and optimization of the CAD tools to focus on a single design component and physical region, and (4) provision of support infrastructure to connect up the separate designs in their separate regions. We provide these with (1) Dataflow-streaming module discipline, (2) Partial Reconfiguration p-blocks, (3) revised tool-chains that work only on the PR regions, (4) a packet-switched overlay network. This allows a form of separate module compilation and linkage that is familar in the software world. We demonstrate the feasibility of compile times in the 2–5 minute range, exploiting either incremental compilation, where only a single module needs to change, or parallel compilation where all modules compile simultaneously on a cloud resources.

## Acknowledgments

This work is funded in part by the Office of Naval Research under grant N000141512006. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

## References

[1] D. Park, Y. Xiao, N. Magnezi, and A. DeHon, "Case for fast FPGA compilation using partial reconfiguration," *FPL*, 2018.

[2] *UG946: Vivado Design Suite Tutorial: Hierarchical Design*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, Aprial 2015. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug946-vivado-hierarchical-design-tutorial.pdf

[3] J. D. Hadley and B. Hutchings, "Design methodologies for partially reconfigured systems," in *FCCM*, April 1995, pp. 78–84.

[4] *Virtex Series Configuration Architecture User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, October 2004, XAPP 151. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp151.pdf

[5] D. Lim and M. Peattie, *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, May 2002, xAPP 290 <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>.

[6] *UG909: Vivado Design Suite User Guide: Partial Reconfiguration*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug909-vivado-partial-reconfiguration.pdf

[7] M. J. Wirthlin and B. L. Hutchings, "Sequencing run-time reconfigured hardware with software," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, February 1996, pp. 122–128.

[8] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. B. W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *INTEGRATION, The VLSI Journal*, vol. 38, no. 1, pp. 107–130, 2004.

[9] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The Erlangen slot machine: A dynamically reconfigurable FPGA-based computer," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 47, no. 1, pp. 15–31, 2007.

[10] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, "Generic software framework for adaptive applications on FPGAs," in *FCCM*, 2009, pp. 55–62.

[11] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *FCCM*, May 2014.

[12] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin, "Using hard macros to reduce FPGA compilation time," in *FPL*, Aug 2010, pp. 438–441.

[13] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *FCCM*, 2011, pp. 117–124.

[14] S. Ma, Z. Aklah, and D. Andrews, "Just in time assembly of accelerators," in *FPGA*, 2016, pp. 173–178. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847341

[15] G. Brebner, "The swappable logic unit: A paradigm for virtual hardware," in *FCCM*, 1997, pp. 77–86.

[16] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek, "Stream Computations Organized for Reconfigurable Execution," *J. Microproc. and Microsys.*, vol. 30, no. 6, pp. 334–354, September 2006.

[17] D. L. How and S. Atsatt, "Sectors: Divide conquer and softwarization in the design and validation of the Stratix 10 FPGA," in *FCCM*, May 2016, pp. 119–126.

[18] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 1974, pp. 471–475.

[19] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM*, April 2007, pp. 55–64.

[20] N. Kapre and J. Gray, "Hoplite: A deflection-routed directional torus NoC for FPGAs," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 10, no. 2, pp. 14:1–14:24, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3027486

[21] C. E. Leiserson, "VLSI theory and parallel supercomputing," MIT, 545 Technology Sq., Cambridge, MA 02139, MIT/LCS/TM 402, May 1989, also appears as an invited presentation at the 1989 Caltech Decennial VLSI Conference. [Online]. Available: http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA214035

[22] N. Kapre, "Deflection-routed butterfly fat trees on FPGAs," in *FPL*, Sept 2017, pp. 1–8.

[23] B. S. Landman and R. L. Russo, "On pin versus block relationship for partitions of logic circuits," *IEEE Trans. Comput.*, vol. 20, pp. 1469–1479, 1971.

[24] D. Clark, "Window and Acknowledgement Strategy in TCP," USC/ISI, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California, 90291, RFC 813, July 1982.

[25] *UG984: MicroBlaze Processor Reference Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 2016. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug984-vivado-microblaze-ref.pdf

[26] "Yosys Open SYnthesis Suite," https://github.com/YosysHQ/yosys, 2019.

[27] D. Shah, E. Hung, C. Wolf, S. Bazanski, and M. M. Dan Gisselquist, "Yosys+nextpnr: an open source framework from verilog to bitstream for commercial FPGAs," in *FCCM*, 2019.

[28] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Tr. Reconfig. Tech. and Sys.*, vol. 7, no. 2, pp. 6:1–6:30, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2617593

[29] "Symbiflow architecture definitions," https://github.com/SymbiFlow/symbiflow-arch-defs, 2019.

[30] D. Glick, J. Grigg, B. Nelson, and M. Wirthlin, "Maverick: A stand-alone cad flow for partially reconfigurable fpga modules," in *FCCM*, 2019.

[31] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *FPGA*, 2018, pp. 269–278.

[32] A. Lalevée, P. H. Horrein, M. Arzel, M. Hübner, and S. Vaton, "Autoreloc: Automated design flow for bitstream relocation on xilinx FPGAs," in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug 2016, pp. 14–21.

[33] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An open-source partial-reconfiguration toolkit for xilinx FPGAs," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 228–235.

[34] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A partial reconfiguration framework," in *FCCM*, April 2012, pp. 37–44.

Web link for this document: <http://ic.ese.upenn.edu/abstracts/prflow_fpt2019.html>