Appearing in *IEEE International Conference on Field-Programmable Technology* (FPT 2022), December 5–9, 2022

Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration

Dongjoon Park, Yuanlong Xiao and André DeHon
Dept. of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA
Email: dopark@seas.upenn.edu, ylxiao@seas.upenn.edu, andre@ieee.org

***Abstract*—To address slow FPGA compilation, researchers have proposed to run separate compilations for smaller design components in parallel. This approach provides small *pages* on the FPGA, allowing users to separately generate partial designs on the pages and load them together. However, this method either forces users to manually decompose a design into small components that fit in small, fixed-sized pages or to use large, fixed-sized pages, reducing the potential compilation speedup benefits. This restriction often results in suboptimal decomposition of a design or diminishes productivity. To overcome these limitations, we utilize the recently supported Hierarchical Partial Reconfiguration technology from Xilinx to generate a more flexible framework. Depending on the size of user designs, our framework provides larger pages that are hierarchically recombined from multiple smaller pages. This flexibility relieves users of the burden to decompose the original design and offers more opportunities for design-space exploration. When tested on the ZCU102 embedded platform with the Rosetta HLS benchmarks, our system achieves 1.4–4.9× mapped application performance improvement compared to the system with fixed-sized pages while still compiling in 2–5 minutes (2.2–5.3× faster than the vendor tool).**

***Index Terms*—FPGA, Compilation, Streams, Dataflow, Latency Insensitive, Hierarchical Partial Reconfiguration, Incremental Refinement**

I. Introduction

Field-programmable gate arrays (FPGAs) are deployed in a wide range of applications, including machine learning [1], [2] pattern matching [3], [4], and genomics [5]. While FPGAs excel in many applications, the long edit-compile-debug cycle has remained a known bottleneck. Hours or days of compilation time often discourages software engineers who expect seconds or minutes of compilation time on CPUs and GPUs. Vendor tools run a monolithic placement and routing on a design with limited parallelism, meaning that the FPGAs compilation does not fully exploit multi-core CPUs or high computing power in the cloud.

Recently, several works have tried to solve the time-consuming monolithic compilation with separate compilations in parallel. RapidStream provides a mesh of disjoint *islands*, and user designs are placed and routed on each island [6]. RapidStream achieves about 2 hours of compilation time for the designs that take about 10–14 hours with the vendor tool.

Xiao et al. [7]–[9] use Partial Reconfiguration (PR) to support separate compilation to reduce compile time. PR is a powerful technique that partially reconfigures the portion of the reconfigurable device while other parts of the design continue to operate [10], [11]. Conventional uses of PR include dynamic system adaptation [12], [13], area reduction [14], [15], and CGRA/FPGA virtualization [16]–[19]. [7]–[9] employ PR to decouple each *operator*, a logical streaming computational block, from each other, mapping each on its own PR *page*, a physical partition of the reconfigurable fabric [20], [21]. PLD provides different compile options like `-O0`, `-O1`, and `-O3` for FPGA compilation [9]. In `-O1` option where the pre-implemented Network-on-Chip (NoC) along with PR is used, PLD achieves 9–19 minutes of compile time for the designs that take 65–110 minutes with the commercial tool.

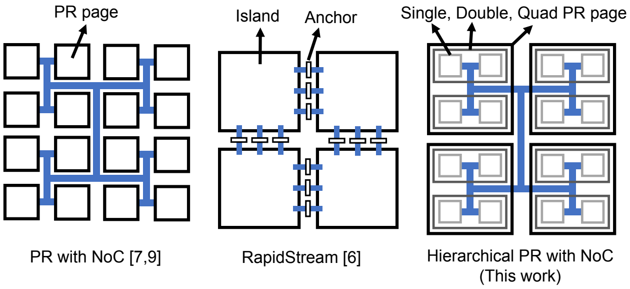


Fig. 1. Comparison between related work

A common challenge from the previous literatures is that the operators are mapped on the fixed-sized islands or pages. If the size of such region is too large, it reduces the speedup benefit of separate compilation as we see in RapidStream where independent page compilation still takes on the order of an hour. On the other hand, if the size of the region is too small, users need to manually divide an application into small operators, and some applications suffer in suboptimal performance because of *unnatural* decomposition as we see in [8].

In this work, we create an overlay with hierarchical PR pages for separately compiled regions (Fig. 1), utilizing Xilinx's Nested Dynamic Function eXchange (DFX) [11]. Our framework consists of PR pages and a pre-built NoC similar to [9]. The difference is that PR pages in this work can be recombined to create double-sized pages or quad-sized pages depending on the size of operators. With a new framework, users can enjoy the benefit of fine-grained separate compilation with single-sized pages so that they can achieve high speedup in compilation. Yet, the users are not forced to decompose a design into small operators because large operators can be mapped to double-sized pages or quad-sized pages. The compilation time is determined by the slowest among all separate compilations, and the users get to control the degree of acceleration in compilation. Our key contributions include:

- We provide a practical open-source framework[1] for the separate compilation that is the first to exploit variable-sized PR pages using Xilinx's Nested DFX technology. Unlike prior works, the users do not need to decompose a design to fit in fixed-sized pages. The users can incrementally increase or decrease the size of an operator, and the system automatically assigns the appropriate size of PR pages, mapping as fast as the operators allow.
- We demonstrate that the use of these variable-sized PR pages improves application performance by 1.4–4.9× compared to a fixed-sized pages system on some Rosetta HLS benchmarks [22] while still compiling 2.2–5.3× faster than Xilinx Vitis. Our framework achieves 2–5 minutes of compilations while Vitis takes 7–22 minutes.
- We illustrate typical use cases for variable-sized PR pages by presenting an incremental development scenario that exploits natural decomposition, inter-operator optimization, and intra-operator optimization.

The rest of the paper is organized as follows: in Sec. II, we introduce hierarchical PR technology and related work on separate compilation on FPGA. Sec. III outlines our idea and engineering considerations. Sec. IV presents a sample incremental refinement scenario and experiment results. Sec. V provides more discussion on results and future directions.

II. Background

A. Hierarchical Partial Reconfiguration

PR design consists of two elements: static design and reconfigurable modules. The static design is the part of the design that does not change during partial reconfiguration. A reconfigurable module is a part of the design that is reconfigured during operation. Hierarchical PR is a technology that allows PR's reconfigurable modules to contain one or more reconfigurable modules inside. GoAhead is an academic PR framework that first supports hierarchical PR [23]. Reconfigurable computing researchers have reduced the memory for bitstream storage and accelerated bitstream loading time using hierarchical PR in GoAhead [24].

Xilinx has recently supported hierarchical PR with the name of Nested DFX [11]. In the Xilinx Vivado PR technology, the user first defines PR regions with *pblocks* and assigns a reconfigurable module to each pblock. After running the placement and routing, the user carves out the reconfigurable modules and locks the left-over logic and routing to create the static design. *Partition pins*, the interfaces between the static logic and the reconfigurable logic, are defined, too. Vivado allows the static design to use resources of the reconfigurable pblocks to enhance routability.

To generate hierarchical PR regions, given the initial routed design, the user *subdivides* the pblock, assigns new reconfigurable modules, and goes through the conventional PR process to generate partition pins to the lower level. Then, the user *recombines* subdivided regions to generate the upper level context bitstream. In order to load the subdivided smaller bitstream, the user first needs to load the upper level bitstreams to set up the context and load the lower level bitstreams.

One use case of the hierarchical PR is the use of PR in the Xilinx DFX platform [25]. DFX platforms consist of the dynamic region where the user designs are mapped and the static *shell* region where the basic infrastructure logic resides. The users get to use PR by default with the DFX platforms. If the users want more granular reconfiguration, they need to subdivide the dynamic region that is already reconfigurable, utilizing hierarchical PR [9], [26].

B. Separate Compilation on FPGA

Guo et al. have suggested RapidStream [6], a separate compilation framework using RapidWright [27]. They provide islands where the processing elements (PEs) of the user design are compiled in parallel. However, in the experiments, the granularity of the islands is 2×2 clock regions on Xilinx U250 FPGA. The size of an island is more than 50K LUTs, and the large area reduces the benefit of separate compilation.

Moreover, after PEs are placed and routed in separate islands, the tool needs to go through a top level stitching operation. This requirement is detrimental, especially in an incremental development scenario where users want to quickly iterate each design point. Even with the fast open-source router [28], the inter-island routing takes about half of an hour. If the size of island decreases, the global stitching time is expected to increase.

Thomas et al. [29] accelerate FPGA compilation by replicating processing units (PUs) alongside pre-implemented connectivity logic. For designs that take 80–120 minutes to compile, they achieve about 10 minutes of total compilation time for 960 LUT pages including a minute of replication processing by RapidWright [27]. But the applications of this work are inherently limited to those with the identical PUs.

Xiao et al. [7]–[9] have utilized PR to separately compile user design components. Because the size of PR page can be smaller or larger than the clock region, this approach can offer more speedup or flexibility in compilation. Authors use a pre-routed NoC [7], [9] or switchboxes, [8], so that, even if the interconnections between the operators change, the users do not need a sequential inter-page stitching process.

Nevertheless, in this approach, the size of PR page is fixed, so it is the user's responsibility to decompose a design into small operators. It is not only an additional burden but also sometimes limits the performance of the design. If an operator is unnaturally decomposed into smaller operators, the bandwidth of the NoC that connects PR pages becomes a bottleneck [8]. Furthermore, the fixed-sized page limits the design-space exploration. For instance, if the users want to accelerate a data-independent loop by unrolling it, the degree of unrolling is determined by the page size because the operator should not exceed the size of the page.

[1] https://github.com/icgrp/prflow_nested_dfx

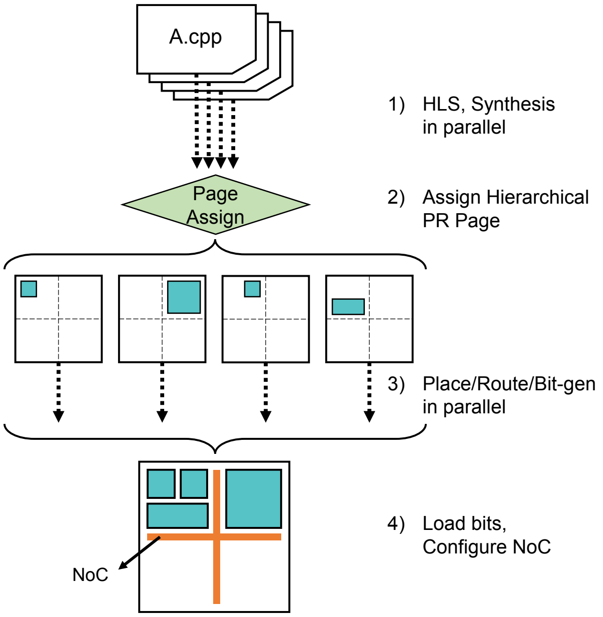

A.cpp
1) HLS, Synthesis in parallel
Page Assign
2) Assign Hierarchical PR Page
3) Place/Route/Bit-gen in parallel
4) Load bits, Configure NoC
NoC

Fig. 2. Separate Compilation with Hierarchical PR Pages Overview

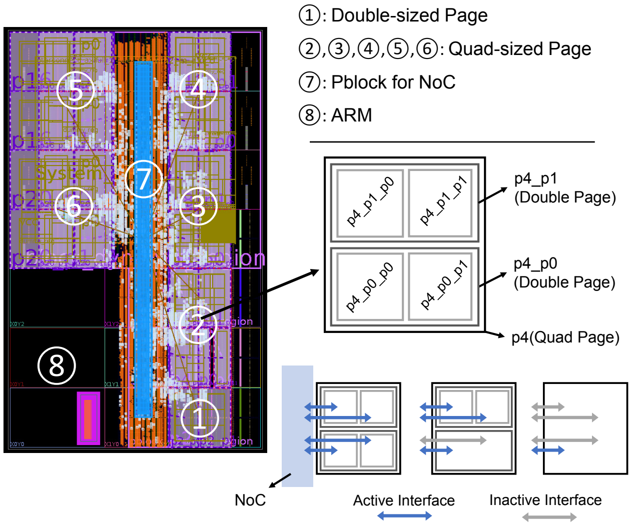


Fig. 3. Example Overlay on the Xilinx ZU9EG FPGA

One option to avoid unnatural decompositions is to create a custom PR design that pre-defines a large enough PR page for each operator [30]. The developer can then incrementally improve operators. However, if too large an area for an operator is reserved, a design with large number of operators cannot be mapped in the available device area. If too small an area for an operator is reserved, it limits the design space exploration. Additionally, unless a pre-built NoC like [7] and [9] is used, when the user wants to change the interconnections between the operators, the user needs a slow compile to regenerate the static design, giving up the benefit of fast compilation.

III. Idea: Overlay using Hierarchical PR

Our approach uses PR, offers smaller pages and avoids a global stitching phase. Our approach uses hierarchical PR so that the users do not need to decompose a design into small operators. Instead, our system recombines multiple pages, hierarchically to hold large operators. In this section, we explain our separate compilation flow and highlight important details in engineering a practical implementation.

A. Separate Compilation Framework using Hierarchical PR

We build upon PLD [9], an open-source framework[2] using PR for separate compilation. PLD offers Xilinx Vitis compatible tool flow for the data streaming compute model where operators communicate through the NoC with streaming links. PLD uses a single-flit, deflection-routed Butterfly-Fat-Tree (BFT) network [31] for the NoC. The inputs to the framework are the C source files for each operator and the top-level source file for the operator composition and software simulation.

PLD parses the top-level source file and configures the NoC to link operators together. The NoC is configured for linking by configuration packets sent to the network interface for each page. Using Vitis_HLS and Vivado, PLD separately compiles each operator in parallel and generates individual partial bitstream. When running an application, the NoC configuration packets and input data for the application are sent from the host to a DMA operator first. The DMA page operator then sends the configuration packets and input data to appropriate page through the NoC.

In PLD, the overlay contains the fixed-sized PR pages connected by the NoC. The size of each operator must be small enough to fit in the fixed-sized pages. We propose to subdivide PR pages to create hierarchical PR pages and recombine smaller pages to create larger pages (Fig. 2). The inputs to our framework are the same as the inputs to PLD. HLS and synthesis for each operator runs in parallel. Unlike PLD, however, we synchronize compile runs after synthesis to assign a PR page to each operator. The tool finds the PR pages of appropriate sizes based on the utilization reports after the synthesis, and then, separate jobs for implementation are launched. After all compile jobs are done, partial bitstreams are loaded together. Fig. 3 shows the screenshot of the example overlay with the hierarchical PR pages on the Xilinx ZU9EG FPGA.

We build the NoC with an input and output port for each of the smallest, single pages. If an operator is assigned to a larger recombined page, the leaf interface for the recombined page uses interface pins to a single page only, for simplicity. The bottom right figures in Fig. 3 show three different cases to use four single pages. A recombined page (a double page or a quad page) uses the bandwidth of single page's leaf interface, and the NoC is oblivious of the page size.

[2] https://github.com/icgrp/pld2022

B. Overlay generation using Hierarchical PR

The Xilinx DFX platform initially consists of a single dynamic region. First, we subdivide the dynamic region into the highest level PR pages, p2, p4, p8, p12, p16, p20, and p_NoC in the Fig. 3 example. Xilinx's Nested DFX technology does not allow more than one pblock to be subdivided until the first pblock has a placed and routed design [11]. This means that we subdivide p2, and after the top-level implementation is finished for p2, p4 can be subdivided. Creating the final routed design for Fig. 3 requires a total of 18 implementations. *Abstract Shells* for all the pblocks are generated from this final routed design.

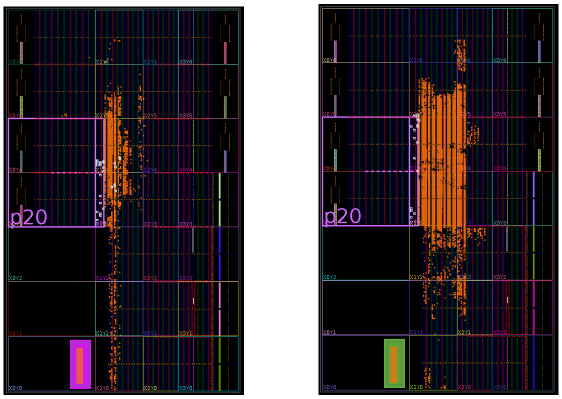
C. Abstract Shell and p_NoC

In Xilinx Vivado, when compiling a reconfigurable module on a PR region, we first load a static design to set up the context and compile the module to the designated pblock. One known issue is that even if the module is compiled on the small pblock, Vivado still loads the entire static design to compile a module to the small pblock. This in-context flow slows down the compilation when the static design is large [7]. To resolve this issue, Vivado supports Abstract Shell [11], which is the minimal logical and physical database for a reconfigurable pblock.

Our framework produces abstract shells for all the PR pages available. We notice that sometimes, the sizes of the static design in the abstract shells are largely unbalanced. This is because the abstract shell contains some placement and all the routing information in the *expanded region* [11]. Vivado relaxes routing requirement for the reconfigurable modules by allowing reconfigurable modules to utilize routing resources outside the pblock in the expanded region (Fig. 4). Because the size of expanded region of each PR page differs, the size of abstract shell differs as well.

We indirectly control the size of the expanded region in the abstract shell by instantiating a pblock for the NoC (p_NoC, as shown by the blue highlighted part in Fig. 3), a similar technique that was used in [7], [32] to accelerate mapping before abstract shells were supported by Xilinx. Because other pblocks cannot route over p_NoC, the expanded regions are limited, resulting in more balanced sizes of abstract shells throughout all pblocks. p_NoC also relieves routing congestion by restricting the NoC logic and distributing the design's routing. Before subdivision, Vivado does not know whether a large pblock will be subdivided or not, so the routing could be unnecessarily congested near the PR pages. In our example overlay, without p_NoC, we encountered a routing failure while subdividing PR pages.

Tab. I compares the sizes of the static design in quad pages, both the case with p_NoC and the case without p_NoC. In the case without p_NoC, we could not process all the subdivisions because of a routing failure, so we create abstract shells from the initial routed design with the highest level user PR pages (p2, p4, p8, p12, p16 and p20) only. In the case with p_NoC, the abstract shells are generated from the routed design after the final subdivision. We can see that p_NoC balances the sizes of static designs in abstract shells. Fig. 4 shows the difference in the sizes of static design in one of the quad pages. When compiling a sample design (an operator of *Digit Recognition* benchmark in Sec.IV) on p20, the time for place, route, and bitstream generation is 93 seconds for Fig. 4 (a) with p_NoC while it is 157 seconds for Fig. 4 (b) without p_NoC. The sizes of generated bitstreams are 2.0 MB and 3.2 MB respectively.



(a) p20 with p_NoC (b) p20 without p_NoC

Fig. 4. Abstract shells for p20 comparison. Orange elements are the static resources showing the extent of the expanded regions.

TABLE I
SIZE OF STATIC DESIGN OF ABSTRACT SHELLS (NUM OF LUTS)

Quad-sized page	with p_NoC	without p_NoC
p4	1745	8459
p8	2817	129
p12	1946	2138
p16	3465	5613
p20	3007	15508

Notes: The numbers on the table are the number of LUTs remaining in the static design (orange elements in Fig. 4) of the abstract shell, not the size of quad pages.

D. Blocked Resources in PR Pages

Previous works regarding separate compilation using PR [7]–[9] did not fully address the blocked resources in PR regions. Xilinx PR technology blocks some resources in reconfigurable pblocks to allow better routing. blockedBelsOutputs.tcl and blockedSiteInputs.tcl are created while generating the overlay and contain the information on blocked resources [11]. We parse these files to exclude blocked resources from the available resources in each pblock. Tab. II shows the available resources in the highest level user PR pages in our example overlay. Tab. II compares the available resources after the first subdivision and the available resources after the last subdivision for the example overlay. As the pblocks are subdivided deeper, the number of blocked resources generally increases because of the additional routing from the static design and upper level hierarchy.

TABLE II
Available Resources in PR regions after Subdivision (%)

	First Subdivision			Last Subdivision		
	LUTs	BRAM18s	DSPs	LUTs	BRAM18s	DSPs
p2	99.9	100	95.0	99.6	96.7	100
p4	99.7	100	100	99.1	91.7	97.1
p8	99.9	100	100	99.6	95.0	99.2
p12	99.9	100	98.8	99.5	89.2	97.5
p16	99.9	89.2	100	99.7	90.0	97.9
p20	99.9	89.2	99.7	99.8	93.3	96.9

When generating an overlay, it is possible to reduce the number of blocked resources by instantiating modules that tightly fit in the pblocks. Because the tight modules push out static logic and routing, the reconfigurable part and the static part become more separated, thereby reducing the number of blocked resources. A blocker module that blocks static routing in a reconfigurable pblock is also an alternative [23], [33]. But such modules make the routing in the overlay generation more challenging. In this work, we give Vivado freedom in routing and take the blocked resources into account when calculating PR pages capacity. We leave it as the future work to reduce the number of blocked resources while reserving more area for the PR pages from the total device area.

E. Margin in PR Pages

Resource utilization in PR pages close to 100% could lead to routing congestion or routing failure, so we added a conservative `MARGIN` value for each page. If $LUT_{op} * (1 + LUT\_MARGIN) < LUT_{page}$, we consider the page has enough LUTs, when LUT_{page} refers to the number of LUTs in a designated page after excluding blocked resources. We have the same rules for BRAMs and DSPs. If the page has enough LUTs, BRAMs, and DSPS, we consider the page is large enough to accommodate the operator. We set most margins at 10–15% but increase the margins up to 20% for specific pages where Vivado has borrowed a large number of routing resources.

F. Greedy Page Assignment Algorithm

We developed a page assignment algorithm to automatically assigns operators to the pages of proper sizes. This automatic page assignment makes the framework accessible to a wider set of developers include HLS developers, because they do not need to be aware of the details of the PR regions and page mapping. Furthermore, if a user must intervene and manually assign the pages after the synthesis runs, it requires minutes of the human time, which could undermine the benefits of fast compilation.

Our greedy algorithm aims (a) to reduce both internal and external fragmentation of the pages and (b) to reduce the number of unnecessary incremental compile jobs. The algorithm starts when the framework generates post-synthesis utilization reports for all the operators (Fig. 2, Step 2).

If it is the first compilation, the algorithm sorts the operators in the descending order of their *sizes*. We define the size of an operator as $LUT_{op}/LUT_{total} + BRAM_{op}/BRAM_{total} + DSP_{op}/DSP_{total}$, when LUT_{total}, $BRAM_{total}$, and DSP_{total} refer to the total each resource available in the device. In this way, the algorithm takes all three resources into account. We start from the largest operator because if small operators are assigned first to single pages that are scattered, it causes external fragmentation, leaving no quad pages available for a large operator.

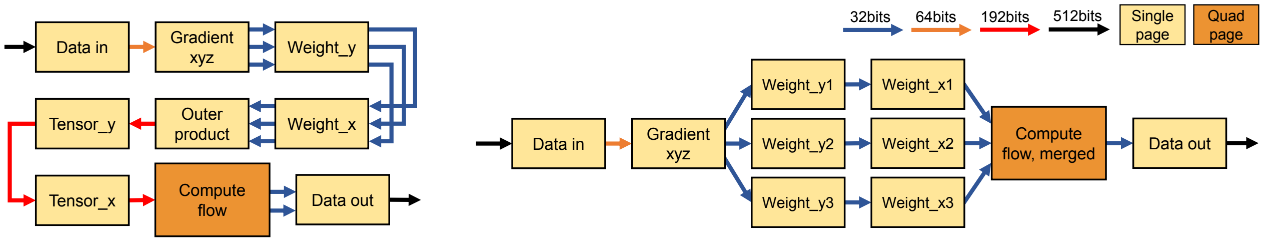
First, we look for single pages to check whether the specific operator can fit in a single page. We create `possible_pblock_list` that contains all the single pages that are large enough to accommodate the operator. To determine whether the operator fits in a specific page, we use the `MARGIN` value discussed in Sec. III-E. Among the possible pblocks, we choose the smallest pblock to reduce internal fragmentation, leaving as much area as possible to other operators. If there is no possible single page, we move to double pages, and perform the same process. If there is no possible double page, we move to quad pages.

If the user has previously compiled the application with our system, edited some operators and incrementally compiled the design, the algorithm first checks whether the previous pblock assignment works for the new operators. If all the operators fit in the previously assigned pages, we keep the assignment and launch compile jobs for place and route immediately. In this way, only operators that are modified by the user are recompiled.

If there exist operators that do not fit in the previously assigned page, we try to assign these operators only on the remaining unused pages. If we successfully find the possible pages, we launch new compile jobs for the modified operators. If we cannot find the possible pages, we start a new page assignment for all the operators. We sort the operators in descending order and check from the single pages again. But this time, instead of choosing the tightest pblock from `possible_pblock_list`, if the previously assigned page is in the `possible_pblock_list`, we choose the previous assignment. With this approach, we can avoid unnecessary recompilations for the unchanged operators. The page assignment algorithm finishes in less than a second for designs with up to 22 operators and 22 pages.

IV. Experimental Evaluations

We create an overlay for the Xilinx ZCU102 evaluation board which uses an UltraScale+ ZU9EG FPGA. We use the officially released Xilinx ZCU102 DFX platform [34] with the dynamic region modified to include more area. The dynamic region available to the users consists of 264,464 LUTs, 1,752 18Kb BRAMs, and 2,448 DSPs. Our framework uses Xilinx Vitis 2021.1 including Vitis_HLS and Vivado. We evaluate our framework on a workstation equipped with the 3.7GHz AMD Ryzen 9 5900X 12 Core CPU with 24 processing threads and 128 GB of RAM.



(a) Natural Decomposition of Optical Flow

(b) Decomposition of Optical Flow after Incremental Development

Fig. 5. Optical Flow Dataflow Graph

TABLE III
OPTICAL FLOW INCREMENTAL DEVELOPMENT WITH HIERARCHICAL PR PAGES

Steps	Largest Operator	Page Size	Compile Jobs	Largest Operator Resource Usage			Incremental Compile Time	App Latency
				LUTs	B18s†	DSPs		
1) Natural	Compute flow	Quad	9	16829	7	24	**261s**	18.7ms
2) Merge Tensor_x	Compute flow	Quad	1	17560	7	54	**245s**	18.1ms
3) Merge Tensor_y	Compute flow	Quad	1	18473	33	68	**274s**	16.0ms
4) Merge Outer prod.	Compute flow	Quad	1	20357	39	74	**288s**	14.7ms
5) Split Weight_x	Weight_x1	Single	3	1690	6	10	**94s**	14.7ms
6) Split Weight_y	Weight_y1	Single	3	1791	18	10	**107s**	14.7ms
7) Optimize Compute flow	Compute flow	Quad	1	20307	45	78	**291s**	8.7ms
8) Change data type	Compute flow	Quad	1	12471	49	192	**274s**	8.7ms

† B18s: BRAM18s

"App Latency" – application execution time per input

A. New Overlay with Hierarchical PR Pages

Our hierarchical overlay consists of 22 single pages, (6933–8409 LUTs, 38–72 18Kb BRAMs, 47–72 DSPs). As shown in Fig. 3, 22 single pages can be recombined to create 11 double pages, (14632–16001 LUTs, 102–116 18Kb BRAMs, 116–143 DSPs). They can also be recombined to create 5 quad pages, (29899–31750 LUTs, 214–228 18Kb BRAMs, 233–282 DSPs).[3] The total available resources on PR pages are (64% LUTs, 70% BRAMs, 57% DSPs) of the device's dynamic region. We use a deflection-routed BFT network [31] with Rent exponent of p=0.5 as our NoC. The size of `p_NoC` is 25,920 LUTs, and the BFT consumes 11,799 LUTs. The clock frequency for the NoC and the user operators is 200MHz.

B. Incremental Development Scenario for the Optical Flow

The Rosetta Benchmark Suite [22] offers a variety of realistic HLS benchmarks. We use *Optical Flow*, *3-D Rendering*, *Digit Recognition* and *Spam Filter* in the Rosetta Benchmark for the demonstration as these are the applications whose decompositions by [9] can be mapped on the ZU9EG FPGA. We compare the performance against the fixed-sized PR pages system. We first present an incremental development scenario for the *Optical Flow* benchmark to illustrate different use cases.

1) Natural Decomposition: The *Optical Flow* Rosetta Benchmark is already decomposed into a set of stream-connected operators and offers both a fixed-point version and a floating-point version. However, in the Rosetta Benchmark *Optical Flow*, there is *Compute flow* operator that contains about 60% of the total LUTs of the design. In [7]–[9], the users cannot map *Compute flow* in the small, fixed-sized PR page, so the authors refine the code with a mix of fixed-point computations and floating-point computations to reduce LUT usage. *Compute flow* is also divided into two operators because of the limited DSPs available in the page.

We, however, can start from the natural decomposition of *Optical Flow* using only fixed-point computations. This is possible because our framework can map the large operator to a double or quad page. The flexibility helps the users to quickly run an application on the device and incrementally refine it. The natural decomposition shown in Fig. 5 (a) is almost identical to the original diagram from [22]. The post-synthesis resource estimate for *Compute flow* is 16K LUTs, and it is mapped to a quad page. All 8 other operators are mapped to single pages. The compilation time and the application runtime latency per input are shown in the last two columns of Tab. III. Tab. III shows the page size and resource usage of the largest operator only. Even with a large operator, compile time never gets much larger than 4 minutes.

2) Inter-operator Optimization: One issue in the natural decomposition (Fig. 5 (a)) of *Optical Flow* is the 192 bits of datawidth between *Outer product*, *Tensor_y*, *Tensor_x*, and *Compute flow*. Our NoC supports only 32 bits of data inputs and outputs in a cycle, so these operators would suffer from an IO bottleneck. [8] resolves this issue by replacing the NoC with switchboxes that directly link neighboring PR pages. This solution, however, forgoes the simple linking of the NoC and requires more parallel compile jobs for the switchboxes.

With our solution, users can merge different operators that suffer from the limited NoC bandwidth since a large operator

[3] Available resources after excluding blocked resources in Sec. III-D

can be mapped in a large PR page. For instance, the users can merge *Outer product*, *Tensor_y* and *Tensor_x* to *Compute flow* so that these operators do not need to communicate through the NoC. These merge steps are shown in the steps 2–4 in Tab. III. The resource consumption of *Compute flow* increases, but the quad page is large enough to accommodate the merged operator. Because we only need to recompile *Compute flow*, the number of compile jobs is 1. The application latency decreases from 18.7ms to 14.7ms.

We can also split *Weight_x* and *Weight_y*, exploiting data parallelism. These changes are shown in the step 5 and 6 in Tab. III. Because we recompile only small operators this time, incremental compile times are far less (1–2 minutes) than the compile time for *Compute flow* (4–5 minutes).

3) Intra-operator Optimization: The users can also further optimize within an operator. The users can unroll some loops to enhance data parallelism, or pipeline some loops to improve pipeline parallelism. With the fixed-sized PR pages, the users need to consider the optimization before they decompose a design because if the size of an operator increases, it may not fit in a fixed-sized page. But the hierarchical PR page lets the users quickly map a design with the straight-forward dataflow graph and perform such optimizations later. As one of these optimizations, in the step 7, we unroll a loop in *Tensor_y* with a factor of 2 and perform the same optimization for *Tensor_x*. Because these optimizations all occur inside the merged *Compute flow*, we categorize them as intra-operator optimizations.

In the step 8, we change the datawidths of variables used in the merged *Compute flow*'s pixel computations. We change the datawidth of `calc_pixel_t` from 64 to 96 to reduce the error rate of the application. In doing so, we partially use a floating-point computation because using solely the fixed-point results in a 62K LUTs of *Compute flow* operator that even a quad page cannot accommodate. We refer to the 64 bits of `calc_pixel_t` with only fixed-point computations as *Optical, (64,fixed pt)*. We refer to the 96 bits of `calc_pixel_t` with a mix of fixed point computations and floating point computations as *Optical, (96,mix)*. The final dataflow graph after the step 8 is shown in Fig. 5 (b).

C. Experiment Results for Rosetta Benchmarks

Experiment results on Rosetta Benchmarks with Vitis monolithic compilation are illustrated in Tab. IV. The clock frequency of the designs for the Vitis's monolithic compilation is 200MHz, and the resource usage represents the application only, excluding the interconnect resources between the host and the kernel. The compile time in Tab. IV does not include time in the packaging step in the Vitis flow. Experiment results on Rosetta Benchmarks with both the fixed PR pages and the hierarchical PR pages are shown in Tab. V. Compared to the resource usage in Tab. IV, our approach uses more resources as it adds an interface for each operator to communicate with the NoC. In Fig. 6, we show the benchmark page mappings with the hierarchical pages. In the following sections, we demonstrate how hierarchical PR pages allow simple incremental changes to improve the application latency per input compared to the fixed-sized pages and compile faster than Vitis monolithic compilation.

TABLE IV
ROSETTA BENCHMARK WITH MONOLITHIC VITIS FLOW (200MHZ)

Benchmarks	Resource Usage			Compile Time	App Latency
	LUTs	B18s	DSPs		
Optical, (64,fixed pt)	26807	164	174	711s	19.1ms
Optical, (96,mix)	19213	187	300	695s	19.4ms
Rendering, Par=1†	4113	65	13	427s	2.5ms
Digit Rec, Par=40†	30650	411	1	919s	12.2ms
Digit Rec, Par=80†	54194	731	1	1340s	11.5ms
Spam, Par=32†	10296	38	224	742s	35.7ms
Spam, Par=64†	16284	38	448	848s	30.4ms

† Par: Parallelization Factor

1) Optical Flow with Fixed-Sized PR Pages: The first two rows in Tab. V are the experiment results of the decompositions for the fixed-sized PR pages. As stated in Sec. IV-B, the users need to use a mix of fixed-point computations and floating-point computations to shrink the size of operators. We refer these decompositions as *Optical, (64,mix)* and *Optical, (96,mix)* when 64 and 96 are the datawidths of `calc_pixel_t`. All 17 operators can be mapped on single pages, but the application latency is worse than the monolithic Vitis flow (Tab. IV) because of the limited NoC bandwidth.

2) Optical Flow with Hierarchical Pages: The third and fourth rows in Tab. V represent *Optical Flow* versions (*Optical, (64,fixed pt)* and *Optical, (96,mix)*) for the hierarchical pages. They are the incrementally developed versions from Sec. IV-B (Tab. III's step 7 and step 8). The application latencies of 8.8ms and 8.8ms are slightly different from 8.7ms and 8.7ms in Tab. III's step 7 and step 8. This is because the page assignment is different for the incremental development case and compilation from the scratch. As the new *Compute flow* mitigates the limited NoC bandwidth issue, we achieve 3.6× and 4.9× reduction in the application latency compared to the fixed-sized pages system. The compilations are still 2.2× and 2.3× faster than the monolithic Vitis flow.

3) 3-D Rendering: We increase the data parallelism by splitting computationally heavy operators into data parallel operators, and one operator in *Rendering, PAR=2* is mapped to a double page. Our framework achieves 1.4× reduction in the application latency compared to the framework with only single pages and compiles 2.5× faster than the monolithic Vitis flow.

4) Digit Recognition: We have 10 operators that compute K-Nearest-Neighbors (KNN) in parallel. In *Digit Rec, PAR=80*, we increase the unroll factor in these operators. All the operators that were previously mapped in single pages are then mapped in double pages because of the increased BRAM usage. This optimization leads to 1.5× reduction in the application latency and compiles 5.3× faster than the monolithic Vitis flow.

TABLE V
ROSETTA BENCHMARK WITH PR PAGES

Benchmarks	Resource Usage			Usage by Page Size			Compile Time	Compile Speedup over Mono.	App Latency	App Latency Improvement over Mono.	App Latency Improvement over Fixed-Sized Page
	LUTs	B18s	DSPs	Sngl.	Dbl.	Qd.					
Optical, (64,mix)*	35953	218	168	17	0	0	276s	N/A	32.1ms	0.6	1.0*
Optical, (96,mix)*	40765	222	330	17	0	0	329s	2.2	43.4ms	0.4	1.0*
Optical, (64,fixed pt)	36336	160	148	9	0	**1**	322s	**2.2**	8.8ms	2.2	**3.6**
Optical, (96,mix)	28500	164	262	9	0	**1**	305s	**2.3**	8.8ms	2.2	**4.9**
Rendering, Par=1*	8605	94	9	5	0	0	151s	2.8	2.6ms	1.0	1.0*
Rendering, Par=2	22435	106	18	6	**1**	0	169s	**2.5**	1.8ms	1.4	**1.4**
Digit Rec, Par=40*	44774	381	2	10	0	0	212s	4.3	7.0ms	1.7	1.0*
Digit Rec, Par=80	70638	701	2	0	**10**	0	251s	**5.3**	4.7ms	2.4	**1.5**
Spam, Par=32*	51461	204	256	15	0	0	287s	2.6	72.5ms	0.5	1.0*
Spam, Par=64	57263	198	512	7	**7**	0	307s	2.8	72.4ms	0.4	1.0

* Decompositions that can also be mapped to the Fixed-Sized PR Pages.

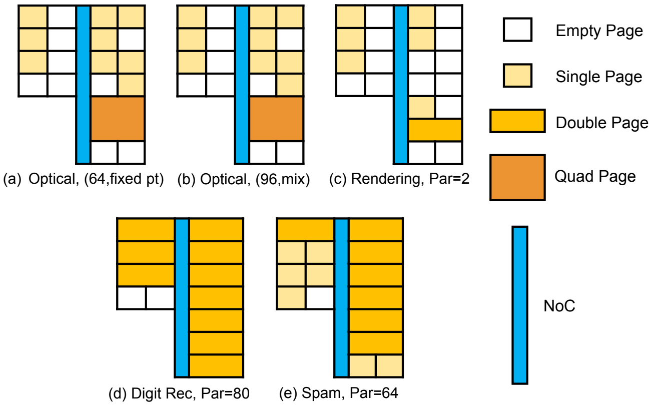


Fig. 6. Rosetta Benchmarks with Hierarchical PR Pages Mappings

While *Digit Rec, PAR=80* achieves 1.5× the performance improvement with the hierarchical PR pages (Tab. V), the performance improvement in the monolithic Vitis flow is only 1.1× (Tab. IV). KNN algorithm consists of the Hamming distance computation and KNN vote computation. While increasing the unroll factor accelerates the Hamming distance computation, the workload for KNN vote computation increases, too, becoming a new bottleneck for the monolithic flow. In the hierarchical PR pages decomposition, KNN vote computation is distributed in 10 operators along with the Hamming distance computation, and KNN vote computation does not become a bottleneck.

5) Spam Filter: We increase the unroll factor from 32 to 64 in *Spam Filter* application, and 7 operators are mapped to double pages because of the increased DSP usage. But we do not achieve speedup in the application latency. This is because we transfer data from host to DMA operator first, and the DMA operator sends data to another operator through the NoC. We can resolve the limited NoC bandwidth between operators by merging those operators as we do in *Optical Flow*. But because the DMA operator is static in the framework, we cannot merge it with other operators. We leave the optimization as the future work.

TABLE VI
BITSTREAM LOADING TIMES FOR FULL APPLICATIONS

Benchmarks	Mono. Vitis	Hierarchical PR pages	
		Dynamic Region	Total
Optical, (64,fixed pt)	147.6ms	165.9ms	496.7ms
Optical, (96,mix)	147.6ms	166.1ms	334.8ms
Rendering, Par=1	153.1ms	167.1ms	319.2ms
Digit Rec, Par=40	176.9ms	166.2ms	463.2ms
Digit Rec, Par=80	211.6ms	166.4ms	522.5ms
Spam, Par=32	300.1ms	166.8ms	612.6ms
Spam, Par=64	466.0ms	165.5ms	619.6ms

D. Bitstream Loading Time Overhead

One advantage of PR in many traditional uses is the short loading time of the partial bitstream compared to that of the full bitstream. But in our framework, the bitstream loading time is often larger than the monolithic bitstream loading time. A downside of the hierarchical PR is that to load a lower level bitstream, we need to load the upper level context bitstreams first. Tab. VI shows the bitstream loading time for both the monolithic Vitis flow and our framework using hierarchical PR. "Dynamic Region" column refers to the first level partial bitstream that contains some peripheral logic for Vitis flow, the NoC, and the context information all the way down to the single pages. "Total" column refers to the loading time for the first level partial bitstream and all the operator bits.

We observe that there exists a significant overhead in bitstream loading time due to the upper level context bitstream, but we can accept the hundreds of milliseconds of overhead to save minutes or hours of compilation time. In an incremental development scenario, it is possible to only load the changed partial bitstreams and related upper level bitstreams, and the overhead is not as large as that of the full application in Tab. VI.

V. DISCUSSION AND FUTURE WORK

In Sec. IV-B, we have seen that a single page takes less than 2 minutes to compile and even a quad page takes only 4–5 minutes to compile. The benefit of separate compilation is that the tool only needs to compile for the portion that is changed, not the entire design.

Although the maximum size of a recombined page is a quad page in our experiments, there is no limit in the maximum size of pages, and larger pages such as octal and beyond are possible. With a deeper hierarchy, smaller base pages are possible as well. But the deeper the PR level gets, the more difficult it is to generate the overlay in the first place since the routing is more difficult. Some resources in the pblock are blocked to accommodate interface pin routing, and blocked resources make the routing of the user module implementation more difficult. Having a pblock for the NoC (`p_NoC`) is one way to provide a hint to Vivado regarding placement and routing. We can also set the range for the partition pin locations to guide Vivado how to route. It is clear there is a rich area for the overlay design and optimization here.

The page assignment algorithm does not consider the effect of PR page's tightness on compilation time. It is challenging to formulate the problem because the compile time is a function of the operator size, PR page size, wiring complexity of the operator, available wiring resources of PR page and others. It will be valuable to develop models that quickly classify fast and slow compile problems based on characteristics such as the ones listed above. Algorithms such as [35], [36] might serve as good starting points.

We saw some cases of performance drops due to NoC congestion from the page assignments during development and will likely need an option to directly target congestion effects as part of an improved page assignment algorithm.

We evaluate our idea of the hierarchical PR pages on a modest embedded platform, but we can extend to the larger data center devices and expect to achieve higher speedup in compilation time.

VI. Conclusions

FPGA compilation is notoriously longer than software compilation. Recent works on separate compilation demonstrate the feasibility of software-like FPGA compilation. We advance the state-of-the-art in separate compilation by supporiting variable-sized pages that provide more flexibility to the users. Our experiment results show that the variable-sized pages give 1.4–4.9× performance improvement compared to the framework with the fixed-sized pages while compiling 2.2–5.3× faster than the commercial tool. Especially in the incremental development scenario, a single page takes less than 2 minutes and a quad page takes 4–5 minutes to compile for the design that the vendor tool takes 11–12 minutes to compile.

References

[1] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *Proceedings of the International Symposium on Computer Architecture*, 2018, pp. 1–14.

[2] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. G. H. Ong, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGA beat GPUs in accelerating next-generation deep neural networks?" in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2017.

[3] V. Sateesh, C. Mckeon, J. Winograd, and A. DeHon, "Pipelined parallel finite automata evaluation," *Proceedings of the International Conference on Field-Programmable Technology*, 2019.

[4] T. Fukač, J. Matoušek, J. Kořenek, and L. Kekely, "Increasing memory efficiency of hash-based pattern matching for high-speed networks," in *Proceedings of the International Conference on Field-Programmable Technology*, 2021.

[5] H.-C. Ng, S. Liu, and W. Luk, "Reconfigurable acceleration of genetic sequence alignment: A survey of two decades of efforts," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2017.

[6] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "Rapidstream: Parallel physical implementation of FPGA HLS designs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 1–12.

[7] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, and A. DeHon, "Reducing FPGA compile time with separate compilation for FPGA building blocks," in *Proceedings of the International Conference on Field-Programmable Technology*, 2019, pp. 153–161.

[8] Y. Xiao, S. Ahmed, and A. DeHon, "Fast linking of separately compiled FPGA blocks without a NoC," in *Proceedings of the International Conference on Field-Programmable Technology*, 2020.

[9] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: Fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 933–945.

[10] *AN 797: Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board*, Intel, 2018. [Online]. Available: https://www.altera.com/documentation/ihj1482170009390.html

[11] *UG909: Vivado Design Suite User Guide: Dynamic Function eXchange*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf

[12] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, "Generic software framework for adaptive applications on FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2009, pp. 55–62.

[13] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, May 2014, pp. 202–205.

[14] M. Farhadi, M. Ghasemi, and Y. Yang, "A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on FPGA," in *IEEE Conference on High Performance Extreme Computing*, 2019.

[15] M. Nguyen, N. Serafin, and J. C. Hoe, "Partial reconfiguration for design optimization," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2020, pp. 328–334.

[16] T. Kong, K. Koul, P. Raina, M. Horowitz, and C. Torng, "Hardware abstractions and hardware mechanisms to support multi-task execution on coarse-grained reconfigurable arrays," in *Workshop on Democratizing Domain-Specific Accelerators*, 2022.

[17] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, "AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Transactions on Embedded Computing Systems*, 2022.

[18] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 4, pp. 1–28, 2020.

[19] Y. Zha and J. Li, "Virtualizing FPGAs in the cloud," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 845–858.

[20] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE): Extended abstract," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, ser. LNCS. Springer-Verlag, August 28–30 2000, pp. 605–614.

[21] 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," *Journal of Microprocessors and Microsystems*, vol. 30, no. 6, pp. 334–354, September 2006.

[22] 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 *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[23] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A partial reconfiguration framework," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 37–44.

[24] D. Koch and C. Beckhoff, "Hierarchical reconfiguration of FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, Sept 2014, pp. 1–8.

[25] *UG1120: Alveo Data Center Accelerator Card Platforms*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, April 2021. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf

[26] *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

[27] C. Lavin and A. Kaviani, "RapidWright: Enabling custom crafted implementations for FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2018, pp. 133–140.

[28] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt, "Rwroute: An open-source timing-driven router for commercial FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 1, pp. 1–27, 2021.

[29] J. Thomas, C. Lavin, and A. Kaviani, "Software-like compilation for data center FPGA accelerators," in *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, June 2021.

[30] Y. Xiao, A. Hota, D. Park, and A. DeHon, "HiPR: High-level partial reconfiguration for fast incremental FPGA compilation," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2022.

[31] N. Kapre, "Deflection-routed butterfly fat trees on FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, Sept 2017, pp. 1–8.

[32] D. Park, Y. Xiao, N. Magnezi, and A. DeHon, "Case for fast FPGA compilation using partial reconfiguration," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2018.

[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] "Vitis embedded platform source repository," https://github.com/Xilinx/Vitis_Embedded_Platform_Source/tree/2021.1, 2021.

[35] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*, 2019, pp. 1130–1135.

[36] D. Maarouff, A. Shamli, T. Martin, G. Grewal, and S. Areibi, "A deep-learning framework for predicting congestion during FPGA placement," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2020, pp. 138–144.