Appearing in *International Conference on Field-Programmable Logic and Applications* (FPL 2022), August 29–September 2, 2022

HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation

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

***Abstract*—Partial Reconfiguration (PR) is a key technique in the design of modern FPGAs. However, current PR tools heavily rely on the developers to manually conduct PR module definition, floorplanning, and flow control at a low level. The existing PR tools do not consider High-Level-Synthesis languages either, which is of great interest to software developers. We propose HiPR, an open-source framework, to bridge the gap between HLS and PR. HiPR allows the developer to define partially reconfigurable C/C++ functions instead of Verilog modules, which benefits the FPGA incremental compilation and automates the flow from C/C++ to bitstreams. By mapping Rosetta HLS benchmarks, the incremental compilation can be accelerated by 3-10× compared with Xilinx Vitis normal flow without performance loss.**

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

I. Introduction

Over the past decades, Field-Programmable Gate Arrays (FPGAs) have been widely used to accelerate diverse applications on machine learning [1], [2], data analysis [2], [3], image processing [4], [5], and others. The hardware programmable features allow the developers to customize the application instances with more flexibility. However, the coding effort and long compilation time hinder the wide deployment of FPGAs. Vendors have been developing versatile tools, such as Vitis [6], SDSoC [7], and OpenCL [8], to decrease the coding difficulties by supporting high-level languages (C/C++). While these solutions can improve coding productivity, the source code will finally go through placement-and-routing, which is the most time-consuming part. In fact, the incremental compile strategy is poorly supported for this most time-consuming place-and-route step. Fig. 1 profiles the compilation time breakdown to implement Rosetta Benchmarks [9] on a Data Center Card (Alveo U50) [10]. Synthesis usually takes more time for the initial compile (green blocks) as some peripheral modules are compiled once and can be re-used in later incremental compile. However, by changing only one source file, we only see 21–36% reduction in the incremental compile times; it takes almost the same time for placement, routing and bitstream generation. In contrast, software applications can be compiled in a different way, where only the modified source files need to be re-compiled. This can save significant time during incremental development where it is common to change only a few functions at a time. We raise the key question here: *Can we compile the HLS source code incrementally, like software, such that we only need to perform placement and routing on the portions of the design that changes?*

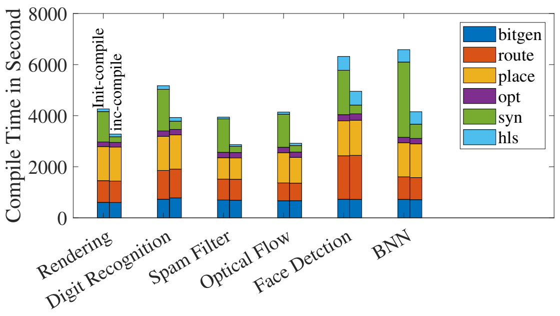


Fig. 1. Initial-Compile vs. Incremental-Compile with Vitis

Several novel proposals [11]–[15] for FPGA parallel compilation flow have been brought forward in recent years. Guo et al. [13] proposed to partition the HLS-code and perform split compile by RapidWright [16], which can accelerate the compilation by 5–7× while increasing the frequency by 1.3×. However, a global stitching step is still needed which restricts the maximum compilation speedup. Xiao et al. [11] proposed a framework that uses Partial Reconfiguration (PR) technique to compile separate C-functions in parallel, so that the incremental-compilation time can be decreased, as only the modified functions need to be recompiled. However, the incremental compiles are based on a pre-compiled fixed overlay, and the applications cannot be mapped until C/C++ functions have been manually decomposed to match the fixed PR block sizes. We propose to customize PR block sizes by defining the partial reconfigurable function in high-level language (C/C++) and automating the complete design flow to generate and exploit PR regions with no manual intervention.

In this paper, we propose a framework called HiPR (**Hi**gh-level **P**artial **R**econfiguration), that is fully compatible with the newest Xilinx Vitis tool flow. Taking as the input the applications based on the Kahn Processing Networks (KPN) model [17], where operators are connected through stream links, HiPR allows the users to define the partial reconfigurable function (operators in KPN) at C-level by using a *pragma* to identify a function as under development and signal that it should be given its own PR region for fast recompilation. When compiling the application for the first time, HiPR compiles each operator function in parallel from C to a post-RTL-synthesis netlist. Using resource requirements from

RTL synthesis, HiPR automatically generates a design-specific overlay with a static region and custom target PR-regions defined in Fig. 2(a). Next, when the user only modifies the target function(s), HiPR only re-compiles the modified function(s). If the user needs to change the interconnection between different operators or add more PR-target functions, HiPR will automatically redefine the floorplan for the static and PR-regions. Based on the context above, we may summarize our contributions as follows:

- We bridge the gap between HLS and Partial Reconfiguration technique by adding a C-level *PR* pragma that signifies when a function should be allocated its own PR region. Our open-source framework HiPR[1] automates the flow from C/C++ to bitstreams, enabling the software developers to use PR techniques without low-level expertise.
- We demonstrate that automatically floorplanned, partial-reconfiguration decomposed designs can support incremental compilation to reduce compile times by evaluating HiPR on the full set of Rosetta benchmarks on the Alveo U50 card to reduce compilation time by 3–10×.

The remaining paper is structured as follows: the recent FPGA compilation techniques are discussed in Sec. II. The proposed model and HiPR toolflow are presented in Sec. III, followed by the light-weight floorplanner in Sec. IV. Sec. V discusses the experiment results and Sec. VI concludes the paper.

II. Background

A. FPGA Compilation and PR Technique

Different from the incremental compilation strategy in software, the FPGA compilation can take hours to days, as the EDA tools need to place and route fine-grained (bit-wise) netlists. This cross-module optimization can generate the best area-performance solutions, but the heuristic algorithms that are usually adopted to solve these NP hard problems [18]–[20] result in long compile time. Moreover, even tiny modifications can trigger complete recompilation, which lengthens the edit-compile-debug loop and reduces the development efficiency at the initial tuning and verification stage.

Partial Reconfiguration (PR) techniques are widely-supported by modern FPGAs [21], where only a portion of the FPGA chip is reconfigured while allowing other modules to run. Xilinx recently released the Dynamic Function eXchange technique (DFX) [22] technique, with which the user can redefine PR regions into sub-PR regions without recompiling the static logic. Additionally, the abstract shell [22] by Xilinx can isolate different PR regions better by only including the related wires, which provides short compilation-times potentials, as CAD tools do not need to load the whole chip database. However, Xilinx leaves all these detailed PR definitions to the designers, which makes DFX inaccessible for the vast majority of HLS users.

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

B. Compilation Acceleration

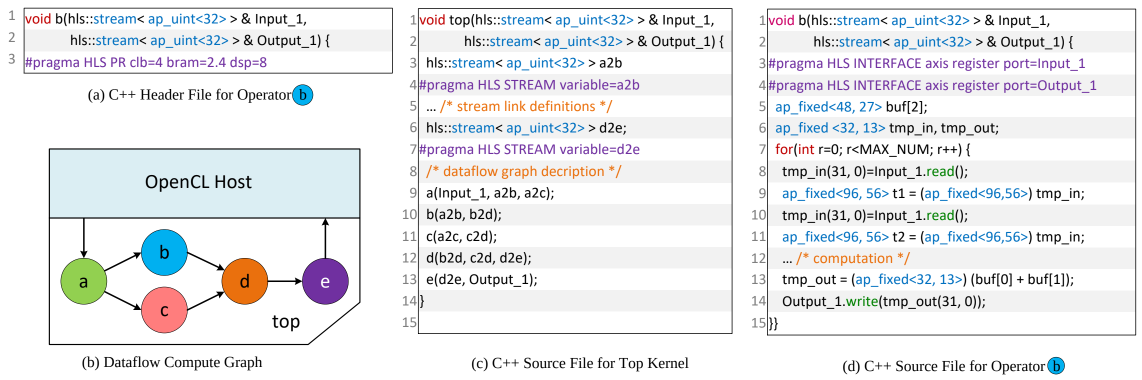
Various approaches in literature propose to divide the FPGAs into separate physical blocks and conduct independent logic mapping [23]–[33]. However, these approaches do not support high-level compilation from C or address the compile time reduction. Grigore et al. [34] proposed a toolflow to automate the generation of partially reconfigurable modules from MaxJ language to bitstreams. However, the toolflow heavily relies on GoAhead [35] and Xilinx ISE, which are not compatible with modern FPGA vendor tools, and the compilation time is not considered. RapidStream [13] can accelerate the compile time by leveraging RapidWight [16] to perform parallel compilation from HLS code to bitstreams. Unfortunately, global routing is still needed to stitch the separate blocks together. Xiao et al. [11], [12], [36] propose to use PR technique to accelerate the compile time. Separate PR regions connected by pre-compile Network-on-a-Chip can accelerate the compile time.

The approach we propose differs from the methods above: HiPR can automatically generate the PR overlay according to application requirements while [11], [12], [36] rely on fixed, pre-compiled overlays; the compilation isolation enables parallel compilation on the cloud, while the global stitching for final bitstream generation cannot be separated in RapidStream [13]; our flow is fully compatible with modern FPGA vendor tools unlike [34].

C. Floorplan for Partial Reconfiguration

The Floorplan is the key to filling the gap between RTL synthesis (generated by HLS) and placement-and-route implementation, and there is a significant body of literature on PR floorplanning [37]–[41]. Taking into account both the heterogeneous resource distributions and PR constraints for modern FPGAs, many floorplanners use heuristic methods [42]–[44]. Bolchini et al. [42] adopt Simulated Annealing (SA) algorithm to explore a reduced search space represented by sequence pair [45]. A greedy floorplan method (Columnar Kernel Tessellation) is proposed in [44] to reduce the resource wastage. A Genetic Algorithm (GA) is adopted in [46] to explore wider feasible solutions. Analytic methods, such Mixed-Integer Linear Programming (MILP) and Nonlinear Integer Programming (NLP) have recently been brought forward to generate global optimal solutions [46]–[49]. The MILP-based floorplanner [46], [47] can find the global optimum, and the users can also change the objective functions with different weights to total wire length, aspect ratio, and resource wastage. FLORA [48] is another MILP-based floorplan tool, which takes into account the more realistic PR constraints and adopts a fine-grained model for modern FPGAs.

While the analytic (MILP) method can outperform heuristic method with the ability to find global optimal solution, it suffers from long execution time and poor scaling with problem size (detailed in Sec.V-A). Hence, HiPR adopts the SA-based floorplanning algorithm to accelerate the compile time, extending the SA by considering modern hierarchical DFX constraints (detailed in Sec.IV-A).



```
1  void b(hls::stream< ap_uint<32> > & Input_1,
2          hls::stream< ap_uint<32> > & Output_1) {
3  #pragma HLS PR clb=4 bram=2.4 dsp=8
```

(a) C++ Header File for Operator b

(b) Dataflow Compute Graph

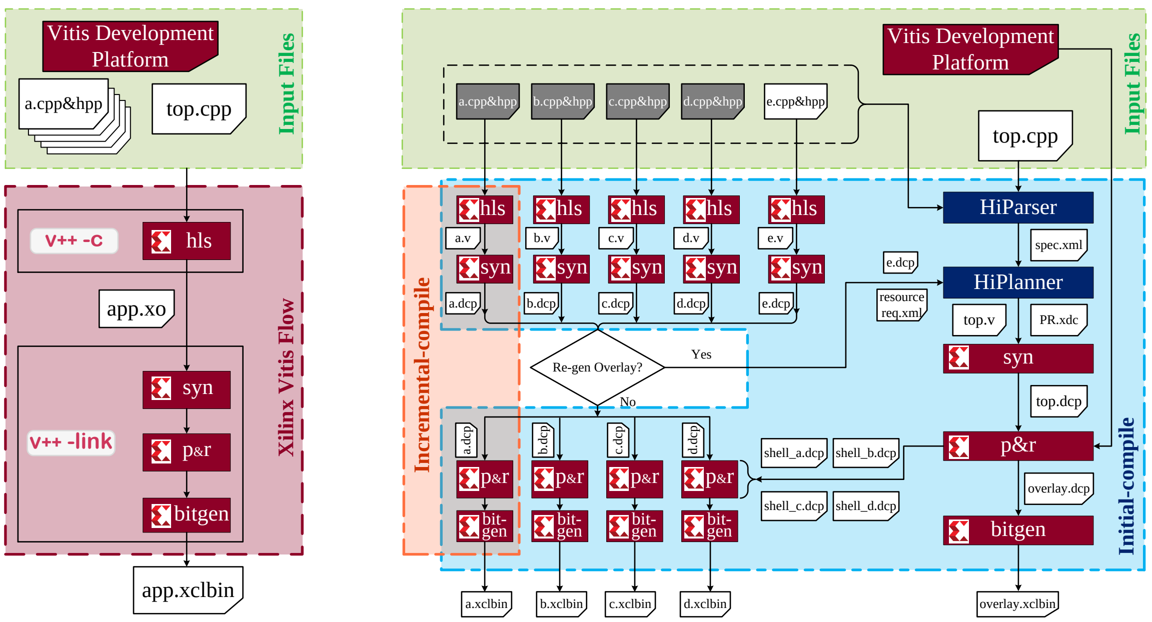
```
void top(hls::stream< ap_uint<32> > & Input_1,
         hls::stream< ap_uint<32> > & Output_1) {
 hls::stream< ap_uint<32> > a2b
#pragma HLS STREAM variable=a2b
 ... /* stream link definitions */
 hls::stream< ap_uint<32> > d2e;
#pragma HLS STREAM variable=d2e
 /* dataflow graph decription */
 a(Input_1, a2b, a2c);
 b(a2b, b2d);
 c(a2c, c2d);
 d(b2d, c2d, d2e);
 e(d2e, Output_1);
}
```

(c) C++ Source File for Top Kernel

```
1  void b(hls::stream< ap_uint<32> > & Input_1,
2          hls::stream< ap_uint<32> > & Output_1) {
3  #pragma HLS INTERFACE axis register port=Input_1
4  #pragma HLS INTERFACE axis register port=Output_1
5   ap_fixed<48, 27> buf[2];
6   ap_fixed <32, 13> tmp_in, tmp_out;
7   for(int r=0; r<MAX_NUM; r++) {
8    tmp_in(31, 0)=Input_1.read();
9    ap_fixed<96, 56> t1 = (ap_fixed<96,56>) tmp_in;
10   tmp_in(31, 0)=Input_1.read();
11   ap_fixed<96, 56> t2 = (ap_fixed<96,56>) tmp_in;
12   ... /* computation */
13   tmp_out = (ap_fixed<32, 13>) (buf[0] + buf[1]);
14   Output_1.write(tmp_out(31, 0));
15 }}
```

(d) C++ Source File for Operator b

Fig. 2. Dataflow Graph and Code Prototype



(a) Xilinx Vitis Flow

(b) HiPR Flow

Fig. 3. Toolflow

III. Problem Model and Proposed Framework

A. Compute Model

The dataflow computational graph model [12], [17], [24], [50] has proven effective in isolating kernels for separate compilation. For Kahn Processing Networks (KPN) [17], each kernel, called an *operator*, is described by a C function in HiPR: the operator receives inputs and sends outputs through latency-insensitive streams [51]; reads to empty streams stall until data become available.

The dataflow graph in our model is illustrated in Fig. 2(b): 1) the design consists of a cluster of operators; 2) different operators are connected by stream links. Fig. 2(c) presents how to describe the dataflow graph in a C program. The operators should obey standard HLS prohibitions such as no allocation or recursion. The interfaces are defined as streaming type (Fig. 2(d) Line 1-4). By calling the `read()` function (Fig. 2(d) Line 8, 10), the operator waits for the valid input data. After all the computations are completed, the operator sends the data out by calling the `write()` function (Fig. 2(d) Line 14).

B. HiPR Framework

We first briefly summarize the Xilinx Vitis flow in Fig. 3(a). Taking in all the C/C++ files as the inputs, `vitis_hls` is called to generate `app.xo` file. Compiling `app.xo` to FPGA-loadable file `app.xclbin` by executing the linkage command(`v++ -link`) is the most time-consuming step. As this linkage step is not open for normal commercial users, it is hard to perform incremental compile with the PR technique.

For HiPR, it takes the same input source as Xilinx Vitis: each operator is described by a C++ function; the function can be defined as partial reconfigurable (Fig. 2(a) Line 3). In this example, we define operators a, b, c and d as Partially Reconfigurable functions (PR-functions) and operator e as a Non-Partially Reconfigurable function (NPR-function). We classify

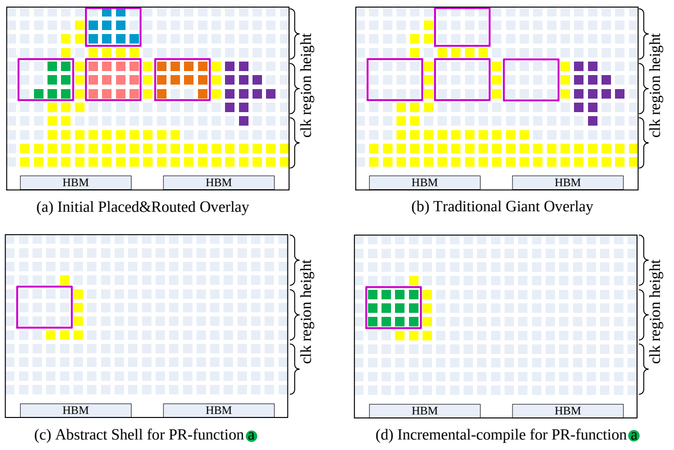

HBM
HBM
clk region height
(a) Initial Placed&Routed Overlay
(b) Traditional Giant Overlay
(c) Abstract Shell for PR-function a
(d) Incremental-compile for PR-function a

Fig. 4. Initial-compile vs. Incremental-compile

the development compilation into 2 types: initial-compile and incremental-compile. For the initial-compile, shown in the blue dashed block in Fig. 3(b), the `HiParser` parses the `top.cpp` file, extracts the interconnection between different operators. The `HiParser` also needs to parse the header files of all the operators, shown in Fig. 2(a), to detect whether a function/operator should be partially reconfigurable. All the above parsed information is included in `spec.xml` file. At the same time, HiPR calls `vitis_hls` and `vivado` to perform compilation for the separate operators in parallel. As the overlay generation is needed for initial-compile, the post-synthesis information is delivered to `HiPlanner`. A simulated annealing floorplanning (Sec. IV) is conducted to generate the `PR.xdc`, which will be fed into `vivado` to generate a partial reconfigurable overlay. When the initial-compile is completed, an `overlay.xclbin` is generated, which corresponds to post-routed device layout in Fig. 4(a). For traditional PR flow without abstract shell [22], a giant overlay (Fig. 4(b)), which contains the definition for all PR regions, is generated. It will be entirely loaded in whenever any PR region needs re-implementation, which can last 10-20 minutes for Alveo data-center FPGAs. With the abstract shell technique, independent `DCP` files are generated for PR functions to perform in-context implementation. In this example, 4 abstract shell `DCP` files are generated for the 4 PR functions (a, b, c, d). Fig. 4(c) shows the abstract shell for PR-function a. Only the partition pins and wires (yellow blocks) related to that PR region are reserved. The post-synthesis netlists for the PR-functions can be placed and routed within the PR regions defined by their abstract shells in parallel. As we use the same Vitis development platform (`hw_bb_locked.dcp`) released by Xilinx [52], the final `xclbin` files can be executed by `Xilinx Runtime` by loading the `overlay.xclbin` first and then `xclbin` files for 4 PR-functions.

The header file is used to signify whether the function/operator is partial reconfigurable. The user can also specify the resource ratio parameters. For example, in Fig. 2(a) Line 3, we can see operator b is a partial reconfigurable function, and the ratio means the final reconfigurable region contains 4 times the CLBs, 2.4 times the BRAM and 8 times the DSPs than the initial resource requirement. This is important to reserve enough space in the floorplanned PR block to accommodate design growth, as the developer can change functionality, add code to fix bugs, and increase parallelism. An application-specific overlay will be finally generated.

For incremental compilation, the developer can modify the PR-functions and perform quick compile as shown in the orange dashed block in Fig. 3(b). For instance, when function a is modified, only this function is recompiled by `vitis_hls` and `vivado`. The post-synthesis design netlist (`a.dcp`) is placed and routed within the PR region individually without touching other parts of the chips shown in Fig. 4(d). Based on these dependencies, we use Google Cloud Platform with the parallel task manager Slurm [53] installed to schedule the compilation tasks. HiPR can generate proper scripts with correct dependencies, and submit the compilation jobs to Slurm. HiPR also supports local machine compilation by using a makefile [54]. The parallelism depends on the local cores and memory size. If the existing PR regions cannot fit the increasing operator size, the users can change the *pragma* in the header file and HiPR will re-generate the overlay by re-launching initial-compile. Changing the streaming links between the operators also lead to re-launching initial-compile as it affects the interconnect wires in static regions.

In summary, HiPR launches the initial-compile to generate an overlay with several partial reconfigurable regions according to the *pragmas* in the C++ header files. Thereafter, the users can tune the PR-functions by launching quick incremental-compile within individual PR regions.

IV. HiPlanner

Our floorplanner, `HiPlanner`, is the key step to bridge HLS and physical PR implementations. Various approaches have been proposed for floorplanning. We adopt the traditional Simulated Annealing (SA) as our main floorplanner engine, since it is faster than analytical methods [46], [48]. We also implemented the MILP-based floorplanner according to [48] for detailed comparisons in Sec. V-A.

A. Problem Formulation

Modern data-center FPGA devices can be described by Cartesian integer coordinates as shown in Fig. 5. In addition to the heterogeneous resource (i.e., CLB, DSPs, BRAMs, ...) with a non-uniform distribution, the vendors also pre-implement some firmware circuits and define a Level-1 DFX region for the users. The basic element of the floorplan is one column wide and one clock region hight (hereafter referred to as a tile). Vertically-stacked PR regions within one clock region are not supported.

The `HiPlanner` takes in the resource requirements from RTL synthesis and a device description file and produces a set of PR constraints that are fed to `vivado` along with the logic netlists to generate an overlay. We model the FPGA device as a 2-dimension matrix, which contains columns of

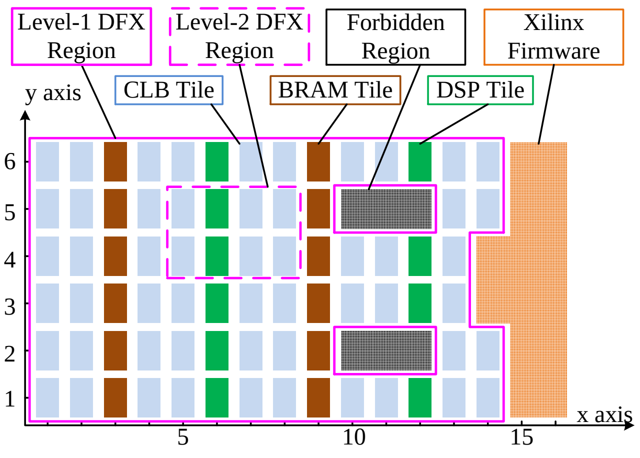


Fig. 5. Data-center FPGA Device Architecture

resources (CLBs, Block RAMs, DSPs, and IOBs). We define the variables for our model as below:

W :=	width of the device in units of tiles;
H :=	height of the device in units of tiles;
T :=	set of tile types considered (CLB, BRAM, DSP);
F :=	set of forbidden areas;
PR :=	set of PR functions;
L :=	set of all the links between 2 PR functions;
x :=	rightmost column coordinate for a tile;
y :=	lowest row coordinate for a tile;
w :=	width of a PR region in units of tiles;
h :=	height of a PR region in units of tiles;
a :=	an area represented by a 4-element vector $< x, y, w, h >$, where x, and y are the lower-left coordinates for the region and w and h are the width and height of the region (e.g., $< 5, 4, 4, 2 >$ for a Level-2 DFX region in Fig. 5);
f :=	an area that could not be used by PR regions ($f \in F$), such as $< 10, 2, 3, 1 >$ and $< 10, 5, 3, 1 >$ in Fig. 5);
r_t :=	number of type t resources ($t \in T$);
l_{pr_i,pr_j} :=	number of interconnect wires between PR regions pr_i and pr_j ($pr \in PR$, $l \in L$);
l_{dma} :=	number of wires connected to DMA (Direct Memory Access).

Based on the columnar-style of modern FPGAs, a W-element vector<CLB, CLB, BRAM, BRAM, ... CLB, CLB> is used to represent the resource distribution over one row. The goal of the `HiPlanner` is to find a set of non-overlapping areas $a_j :< x_j, y_j, w_j, h_j > \mid j \in \{0, .., |PR| - 1\}$ to map all the PR functions $pr_i \in PR| \; i \in \{0, .., |PR| - 1\}$.

With the specified variables as above, we compute the centroid coordinates of an area a_i:

$$xc_{a_i} = x_{a_i} + w_{a_i}/2 \tag{1}$$

$$yc_{a_i} = y_{a_i} + h_{a_i}/2 \tag{2}$$

We use Manhattan Distance to represent the wire length between 2 areas:

$$Mdist_{a_i,a_j} = |xc_{a_i} - xc_{a_j}| + |yc_{a_i} - yc_{a_j}| \tag{3}$$

B. Objective Function

The main factors we consider in optimization objective functions are total wires length, wastage areas, and PR function overlaps as below.

$$min: \; \alpha * WL_{norm} + \beta * RW_{norm} + \gamma \tag{4}$$

where α and β are weights for total wire length and resource wastage respectively; the sum of α and β is 0.5; γ is the overlapping areas in units of tiles.

The absolute total wire length, WL_{abs}, is computed as:

$$\begin{aligned} WL_{abs} = & \sum_{pr_i,pr_j \in PR|i<j} Mdist_{a_i(pr_i),a_j(pr_j)} \cdot l_{pr_i,pr_j} \\ & + \sum_{pr_i \in PR} Mdist_{dma,a_i(pr_i)} \cdot l_{dma} \end{aligned} \tag{5}$$

where pr_i and pr_j are 2 different PR functions; $a_i(pr_i)$ means area a_i is assigned to PR function pr_i. The first term represents the total number of wires for all the links between PR regions, and the second term represents the number of wires between PR regions and the static DMA regions.

The normalized total wire length is calculated as:

$$WL_{norm} = \frac{WL_{abs}}{|L| \cdot \max\{l_{pr_i,pr_j} | l_{pr_i,pr_j} \in L\} \cdot (W + H)} \tag{6}$$

where $|L|$ represents the total link number; $max\{l_{pr_i,pr_j} | l_{pr_i,pr_j} \in L\}$ represents the maximum width of all the links; $W + H$ represents the maximum Manhattan distance between two PR regions or between one PR region and the DMA location. The normalized total wire length is less than 1.

The normalized resource wastage RW_{norm} is computed as:

$$RW_{norm} = \frac{1}{|PR|} \sum_{i \in \{0,..|RP|-1\}} \sum_{t \in T} \frac{r_{a_i,t} - r_{pr_i,t}}{r_{chip,t}} \tag{7}$$

where $r_{a_i,t}$ represents resource type t in an area a_i that is assigned to PR function pr_i; $r_{pr_i,t}$ represents the number of resource type t for PR-function pr_i. The numerator means the extra resource the PR region provides beyond what the PR functions really need. We divide it by the total resources of the chip $r_{chip,t}$ and $|PR|$ to guarantee that the normalized resource wastage is also less than 1.

As the sum of α and β is 0.5. The floorplan is only legal when the cost function is less than 1, as any overlapping areas will increase the γ to more than 1.

C. Greedy PR Shape Generation and Simulated Annealing

Since the FPGA fabric is non-homogeneous, when we move a region from one x location to another, the existing width, w, and h height may not provide the needed resources. Consequently, we use a greedy method to reshape the region to cover

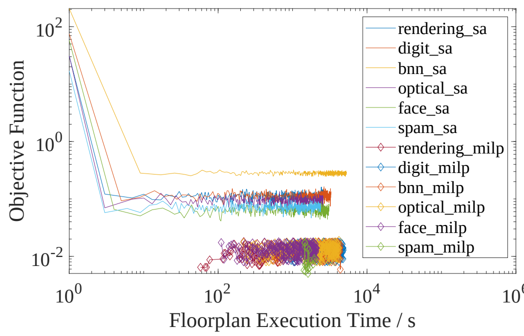


Fig. 6. Floorplan Execution Time in Seconds

TABLE I
FLOORPLAN RUNTIME (IN SECONDS)

Name	PR #	LUT	BRAM18	DSP	Runtime
3d-rendering	6	5718	64	15	0.050 s
Digit Rec	20	40758	320	0	0.006 s
Spam Filter	15	11382	12	256	0.004 s
Optical Flow	16	18489	84	330	0.086 s
Face Detect	20	66654	169	100	0.050 s
Binary NN	22	36950	1,042	5	0.022 s

the required resources. For each PR region $a :< x, y, w, h >$, when the x and y are determined, we will greedily include more columns in the right direction by increasing the w to meet the resource requirements, assuming $h = 1$ initially. When $x + w$ reaches W or the w/h is more than 80, we increase h by 1 and start over from the previous greedy step again. If $y + h$ reaches H, we set x and y all to 0 and start the previous greedy step again. This can provide access to the whole chip resources.

For the initial point, we randomly generate the x and y coordinates for all the PR regions and perform the greedy reshaping method to generate the PR regions. For the following simulated annealing step, we randomly select one PR region and randomly generate the new x and y coordinates and refine the PR regions by using the greedy reshaping method above. In fact, the PR regions can be represented as $a_j :< x_j, y_j, f_w(x_j, y_j, pr_i), f_h(x_j, y_j, pr_i) >$, as w_j and h_j are determined by the x_j, y_j and pr_i.

V. Experimental Evaluations

We evaluate the compile time acceleration of our framework by implementing the realistic Rosetta HLS Benchmarks [9] on the Alveo U50 Data Center card [10] with a Virtex UltraScale+ XCU50 FPGA and 8 GB HBM. Subtracting the pre-implemented firmware from Xilinx, a large PR region is available for the users (751,793 LUTs, 2,300 18Kb BRAMs and 5,936 DSPs). HiPR uses Xilinx Vitis 2021.1 including associated Vivado and Vitis_HLS and XRT as the backend. The Google Cloud computing instance cluster includes a controller node (30-cores, 3.1 GHz Intel Xeon Intel(R) Cascade Lake processors and 128GB RAM) and 32 computing nodes (8-cores, 2.8 GHz Intel Xeon Intel(R) Cascade Lake processors and 32GB RAM for each).

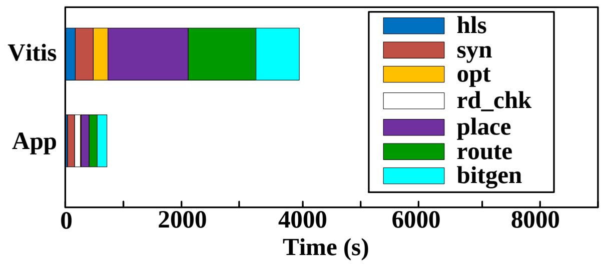


Fig. 7. Incremental-compile Times Breakdown (Digit Recognition)

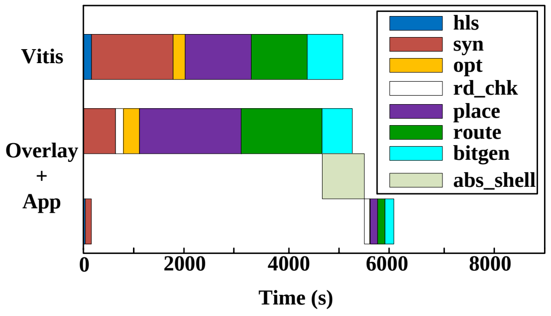


Fig. 8. Initial-compile Times Breakdown (Digit Recognition)

A. Floorplanner

First, we show that `HiPlanner` performs comparably to state-of-the-art floorplanners. The proposed SA-based floorplanner is implemented in C++ prototype and is compared against our implementation of the MILP floorplanner [48] that already showed better results than [46] and [43]. However, since [48] is only based on Virtex-7 series and did not consider the hierarchical DFX features, we enhanced it to support these features mentioned in Sec. IV-A.

The floorplan times for real benchmarks are summarized in Tab. I. Here we only list our Simulated Annealing execution time as all the MILP method cannot converge to the optimal solutions within 24 hours. We compare the cost function over the execution time between our SA method and MILP in Fig. 6. Our SA method can always generate a legal floorplan within 1 second, but it takes more than 100 seconds for the MILP to generate feasible solutions. However, the cost function of MILP is better than SA even when it does not reach an optimal solution.

B. Compilation Time and Performance

Incremental-Compile: The main contribution of HiPR is to accelerate the incremental compilation as only the modified functions need to be recompiled. Fig. 9 shows the compilation distribution for different operators over the full benchmark sets. The operators can be incrementally recompiled in 7-20 minutes. For all the benchmarks, the median values are near 11 minutes. This means that in most cases, users can benefit from short incremental-compilation to tune their target functions more efficiently. We can see the incremental-compilation can be improved by a factor of 3–10. Fig. 7 shows the compilation time breakdown for digit recognition benchmark. We can see

TABLE II
ROSETTA BENCHMARKS INCREMENTAL-COMPILE TIMES (SECONDS)

	Vitis Flow with 30 Threads					HiPR with 8 Threads for each Operator					
	hls	syn	p&r	bitgen	total	hls	syn	p&r	bitgen	total	Speedup
3d-rendering	105	220	2353	600	3278	19	95	585	209	908	3.6
Digit Recognition	144	322	2681	780	3927	28	93	425	149	695	5.6
Spam Filter	69	240	1866	690	2885	17	87	441	147	692	4.1
Optical Flow	88	255	1905	688	2936	16	120	385	136	657	4.4
Face Detection	539	344	3349	722	4954	17	97	655	183	952	5.2
Binary NN	488	556	2399	711	4154	223	418	433	158	1232	3.4

TABLE III
ROSETTA BENCHMARKS 1ST-COMPILE TIMES (SECONDS)

	Vitis Flow with 30 Threads					HiPR with 8 Threads for each Operator					
	hls	syn	p&r	bitgen	total	syn	p&r	max(bitgen, shell_gen)	max op †	total	Overhead §
3d-rendering	104	1190	2364	606	4264	674	4756	814	908	7152	67 %
Digit Recognition	144	1627	2673	729	5173	674	3955	801	695	6125	18 %
Spam Filter	69	1308	1867	698	3942	766	2243	840	692	4541	15 %
Optical Flow	84	1293	2094	668	4139	645	4761	817	657	6880	66 %
Face Detection	542	1738	3280	728	6288	679	6292	928	952	8851	40 %
Binary NN	485	2946	2430	723	6584	697	6751	952	1232	9632	46 %

† Maximum compile time for all the operators
§ The overhead is calculated by divide the total time different between HiPR and Vitis over the Vitis time.

TABLE IV
PERFORMANCE COMPARISON: VITIS VS. HIPR

	Vitis Flow		HiPR	
	Freq (MHz)	Runtime (ms)	Freq (MHz)	Runtime (ms)
3d-rendering	200	2.2	200	1.6
Digit Recognition	250	9.2	250	6.3
Spam Filter	300	18.6	300	20.0
Optical Flow	200	13.6	200	7.5
Face Detection	200	21.0	200	22.0
Binary NN	150	5250	150	4700

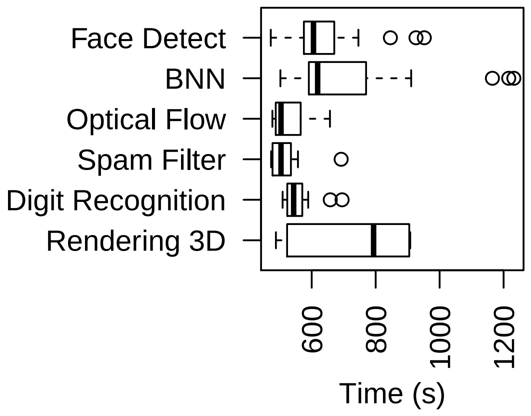


Fig. 9. Operators Mapping Time Distribution

the place-and-route time is accelerated most. Tab. II shows the detailed compilation time. For HiPR, we choose the maximum compile time from all functions for each benchmark as the compilation time. Even with the worst case, HiPR can still outperform Vitis by 3.4–5.6×.

First-Compile: When a benchmark is compiled for the first time, it takes more time for Vitis to compile peripheral modules, such as AXI bus, debugging logic, DMA/HBM driver and others. For HiPR, it needs to implement an overlay with PR modules defined. In Fig. 8, we can see HiPR takes more time to generate the overlay for digit recognition benchmark, as the operators have to be placed and routed along with overlay generation. In Tab. III column 9, we choose the maximum value between overlay bitstream generation and abstract shell generation, as they can be conducted simultaneously. It takes at most 67% overhead in compile time to set the overlay up. However, this process is usually performed once, and users can benefit from incremental-compilations afterward.

Performance Comparison: Tab. IV summarizes the performance between Vitis and HiPR. As we rewrite the original code in the latency-insensitive style (Sec. III-A), the throughput is slightly different from Vitis implementation performance. However, HiPR achieves the same frequency and better performance than the original Vitis Flow. The combination of smaller, localized blocks with pipelined interconnect, enabled by the latency-insensitive discipline, makes it easier to achieve higher clock frequencies [12], [13], which compensates for the frequency loss from the PR technique. HiPR can compile with the same frequency from normal Vitis Flow for each benchmark.

VI. CONCLUSIONS

In this paper, we propose HiPR, a framework that allows the users to define partial reconfigurable C-functions instead of Verilog-modules. This can greatly benefit the incremental FPGA development, as only the modified functions are recompiled (place&route) without waiting the longer time for full recompilation. The experiments from Rosetta Benchmark implementation show that HiPR can decrease the incremental-compilation time by a factor of 3–10× without performance loss or need to target fixed PR region sizes.

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*. IEEE Press, 2018, pp. 1—-14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

[2] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[3] M. Casias, K. Angstadt, T. Tracy II, K. Skadron, and W. Weimer, "Debugging support for pattern-matching languages and accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1073–1086.

[4] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 327–338.

[5] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 9–16.

[6] *UG1145: Xilinx Vitis Unified Software Platform User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1145-sdk-system-performance.pdf

[7] *UG1027: SDSoC Environment User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, May 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf

[8] G. Jo, H. Kim, J. Lee, and J. Lee, "SOFF: An OpenCL high-level synthesis framework for FPGAs," in *Proceedings of the International Symposium on Computer Architecture*. IEEE Press, 2020, pp. 295—-308. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00034

[9] 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.

[10] *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

[11] 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.

[12] 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 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 933–945. [Online]. Available: https://doi.org/10.1145/3503222.3507740

[13] 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 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361

[14] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, 2021, pp. 81—-92. [Online]. Available: https://doi.org/10.1145/3431920.3439289

[15] 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. [Online]. Available: https://doi.org/10.1145/3468044.3468047

[16] 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.

[17] 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.

[18] P. Bressana, N. Zilberman, and R. Soulé, "Finding hard-to-find data plane bugs with a PTA," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 218–231.

[19] R. Venkatakrishnan, A. Misra, and V. Kindratenko, "High-level synthesis-based approach for accelerating scientific codes on FPGAs," *Computing in Science & Engineering*, vol. 22, no. 4, pp. 104–109, 2020.

[20] J. Landgraf, T. Yang, W. Lin, C. J. Rossbach, and E. Schkufza, "Compiler-driven fpga virtualization with synergy," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 818–831.

[21] *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

[22] *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

[23] G. Brebner, "The swappable logic unit: A paradigm for virtual hardware," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 77–86.

[24] 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.

[25] 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.

[26] 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.

[27] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

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

[29] ——, "When application-specific ISA meets FPGAs: a multi-layer virtualization framework for heterogeneous cloud FPGAs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 123–134.

[30] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014, pp. 1–10.

[31] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, pp. 107—127.

[32] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 271–286.

[33] J. Landgraf, T. Yang, W. Lin, C. J. Rossbach, and E. Schkufza, "Compiler-driven FPGA virtualization with SYNERGY," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2021, pp. 818—831. [Online]. Available: https://doi.org/10.1145/3445814.3446755

[34] N. B. Grigore, C. Kritikakis, and D. Koch, "Hls enabled partially reconfigurable module implementation," in *International Conference on Architecture of Computing Systems*. Springer, 2018, pp. 269–282.

[35] CosReCos, *https://www.mn.uio.no/ifi/english/research/projects/cosrecos/*, 2020 (Accessed: 2020-11-16). [Online]. Available: https://www.mn.uio.no/ifi/english/research/projects/cosrecos/

[36] 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.

[37] Y. Feng and D. Mehta, "Heterogeneous floorplanning for fpgas," in *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, 2006, pp. 6 pp.–.

[38] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–8.

[39] P. Banerjee, M. Sangtani, and S. Sur-Kolay, "Floorplanning for partially reconfigurable fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 1, pp. 8–17, 2010.

[40] A. Montone, M. D. Santambrogio, and D. Sciuto, "Wirelength driven floorplacement for fpga-based partial reconfigurable systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[41] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and floorplanning in dynamically reconfigurable fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, pp. 1–34, 2010.

[42] C. Bolchini, A. Miele, and C. Sandionigi, "Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable fpga systems," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 532–538.

[43] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio, "Floorplanning automation for partial-reconfigurable fpgas via feasible placements generation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 151–164, 2016.

[44] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *International symposium on applied reconfigurable computing*. Springer, 2012, pp. 13–25.

[45] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Vlsi module placement based on rectangle-packing by the sequence-pair," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 12, pp. 1518–1524, 1996.

[46] M. Rabozzi, J. Lillis, and M. D. Santambrogio, "Floorplanning for partially-reconfigurable fpga systems via mixed-integer linear programming," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014, pp. 186–193.

[47] M. Rabozzi, A. Miele, and M. D. Santambrogio, "Floorplanning for partially-reconfigurable fpgas via feasible placements detection," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 252–255.

[48] B. B. Seyoum, A. Biondi, and G. C. Buttazzo, "Flora: Floorplan optimizer for reconfigurable areas in fpgas," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[49] T. D. Nguyen and A. Kumar, "Prfloor: An automatic floorplanner for partially reconfigurable fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 149–158.

[50] 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. [Online]. Available: http://ic.ese.upenn.edu/abstracts/score_jmm2006.html

[51] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[52] Xilinx, *xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_all.deb*, 2021 (Accessed: 2022-06-18). [Online]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/alveo/u50.html

[53] Google, *Deploying a Slurm cluster on Compute Engine*, 2021 (Accessed: 2021-08-10). [Online]. Available: https://cloud.google.com/architecture/deploying-slurm-cluster-compute-engine

[54] GNU, *GNU make*, 2021 (Accessed: 2021-09-10). [Online]. Available: https://www.gnu.org/software/make/manual/make.html