Accelerating FPGA Developments from C to Bitstreams by Partial Reconfiguration

Yuanlong Xiao

A DISSERTATION

in

Electrical and Systems Engineering

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2023

Supervisor of Dissertation	Graduate Group Chairperson
André DeHon, Professor	Troy Olsson, Associate Professor
Electrical and Systems Engineering	Electrical and Systems Engineering

Committee:

Boon Thau Loo, Professor of Computer and Information Science
University of Pennsylvania

Jing (Jane) Li, Associate Professor of Electrical and Systems Engineering
University of Pennsylvania

Alireza Kaviani, Senior Engineering Fellow, Adaptive and Embedded Computing Group Engineering, Advanced Micro Devices, Inc

André DeHon, Professor of Electrical and Systems Engineering
University of Pennsylvania

Accelerating FPGA Developments from C to Bitstreams by Partial Reconfiguration

ACKNOWLEDGMENT

First and foremost, I would like to thank my advisor, Prof. André DeHon. I have been very fortunate to work with him in IC group for the last few years. His sharp insight, in-depth technical expertise, and tireless support made this work possible. As an advisor, he guides and teaches me through his behavior and personality, such as great self-discipline, insistence on the highest standard, and the quest for scientific perfection. These will be invaluable experiences in my life.

I would also like to thank all the members of my proposal and defense committee: Prof. Jing (Jane) Li, Prof. Boon Thau Loo, and Dr. Alireza Kaviani. Their insightful feedback put this dissertation on the right track. They also provided valuable comments that greatly improved the quality of the final dissertation.

Furthermore, I would like to thank all the members of IC group: Hans Giesen, Dongjoon(DJ) Park, Syed Tousif Ahmed, Rui Ding, Zhaoyang Han, Vipula Sateesh, Andrew Butt, Eric Micallef, Matthew Hofmann, Ezra Thomas, and Raphael Rubin. Your kind help and support are more than words can express.

I would like to express my gratitude to my family. Thanks to my parents for supporting me in gaining versatile experience both in study and life in the US. You are always there to encourage and comfort me despite how hard it is. I would also like to thank my partner Yue Hu who accompanied and encouraged me through the last and hardest time before the final goal. Without you, this dissertation cannot be completed.

ABSTRACT

Accelerating FPGA Developments from C to Bitstreams by Partial Reconfiguration

Yuanlong Xiao

André DeHon

Divide-and-conquer and incremental compilation strategies are widely used in software compilations. The divide-and-conquer means that separate source files are compiled independently by multi-threads to objectives, which are linked together to an executable-format file, while incremental compilation means that software tools only need to re-compile modified source files and quickly re-link the objectives. To enable these strategies for FPGAs, this dissertation presents an open-source framework called PRflow which can speed up the compilation times by an order of magnitude. PRflow supports different optimization levels to make better trade-offs among compile-time, area, and performance. -O0 (PRflow_RISCV) maps applications to a cluster of on-chip RISC-V cores within seconds for quick verification and debugging. -O1 (PRflow) compiles the separate parts of an application to partial FPGA bitstreams for different partial reconfigurable regions on the chip. Separate parts can be compiled in parallel within 24 minutes. The interconnections between separate parts can be set up by sending configuration packets to configure a network-on-a-chip (NoC) without re-routing physical wires. -O2 (PRflow_DW) supports inter-connection customization with a fixed page-size overlay on top of a commercial FPGA to meet high inter-page bandwidth requirements, improving the performance by up to 10× compared with -O1. -O3 (PRflow_HiPR) supports overlay customization for arbitrary inter-page throughput and various page size requirements with similar incremental compile time to -O1 and -O2. HiPR extracts the interconnect information among separate sub-functions and generates a customized overlay with PR regions defined. Users can perform quick incremental compilation for dedicated sub-functions at the cost of an acceptable one-time overlay compilation overhead. -O3 compiles applications with the most aggressive optimization strategies similar to commercial tools.

We demonstrate the PRflow framework on the Xilinx Alveo-U50 data-center card with an xcu50-fsvh2104-2-e FPGA chip (16nm FinFET) by mapping Rosetta HLS complete benchmark set. PRflow can accelerate the compilation times from 2–3 hours (state-of-the-art Vitis) to 10-24 minutes.

We expect PRflow based on PR technique to become an important compilation strategy as the increasing scales of FPGAs greatly slow down the compile times.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Chapter 1

Introduction

1.1 Thesis

By applying the divide-and-conquer strategy to FPGA compilation, the compilation time is reduced from hours to minutes. With modern partial reconfiguration techniques, applications can be separated into several small blocks connected by latency-insensitive links according to directives/pragmas in high-level languages. The separated blocks are compiled in a divide-and-conquer manner: separate parts of the design are compiled in parallel; modified parts can be recompiled independently in the following incremental refinements.

1.2 Motivation

Over the past decades, Field-Programmable Gate Arrays (FPGAs) have been widely used to accelerate diverse applications for image processing [24, 60], machine learning [40, 25], data analysis [18, 25], and others. The hardware programmable features allow the developers to customize the application instances with more flexibility [3, 20]. However, the *coding difficulty* and *long compilation time* hinder the wide deployment of FPGAs. Versatile tools have been released by vendors, such as SDSoC [129], Vitis [131], and OpenCL [54], to relieve users from arcane hardware languages (Verilog or VHDL) by supporting high-level languages (C/C++). While these solutions can improve coding efficiency, the most time-consuming steps (e.g., placement, routing, and bitstream generation) are still unavoidable to compile the High-level source code to FPGA executable bitstreams.

Figure 1.1 shows the compilation time breakdown to map the Rosetta HLS benchmarks [146] to Alveo U50 data center FPGA card [124]. It takes hours for one edit-compile-debug loop, and the HLS step only occupies 2–12% of the total compile time. For larger designs mapped to FPGAs on the cloud, the compilation time can be 10+ hours [13, 110, 70, 21]. For novices, the long compile time may frustrate the developers and make FPGAs unattractive; for experts, the long cycles limit the design space and could finally lead to sub-optimal solutions within a short time-to-market window.

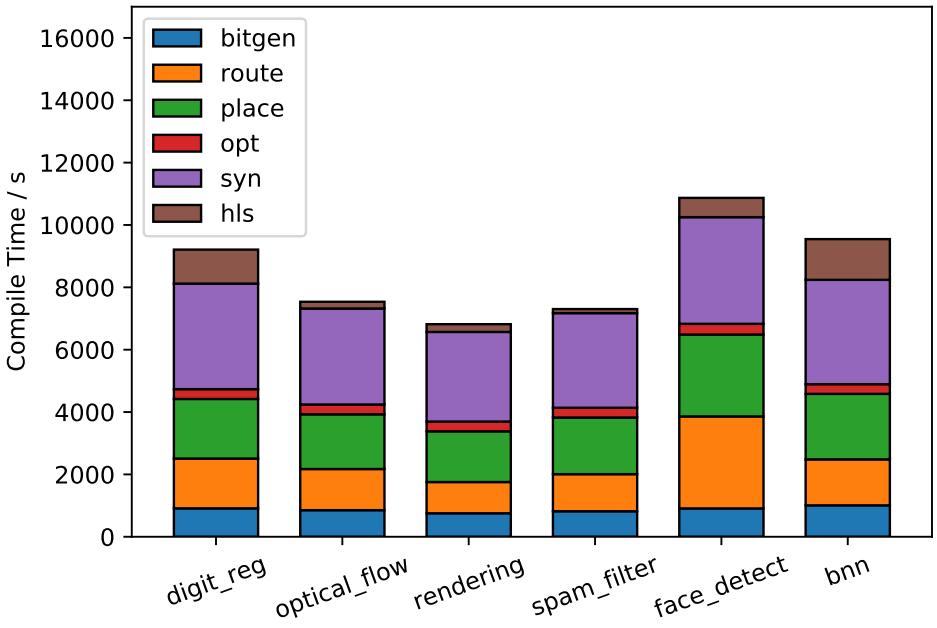
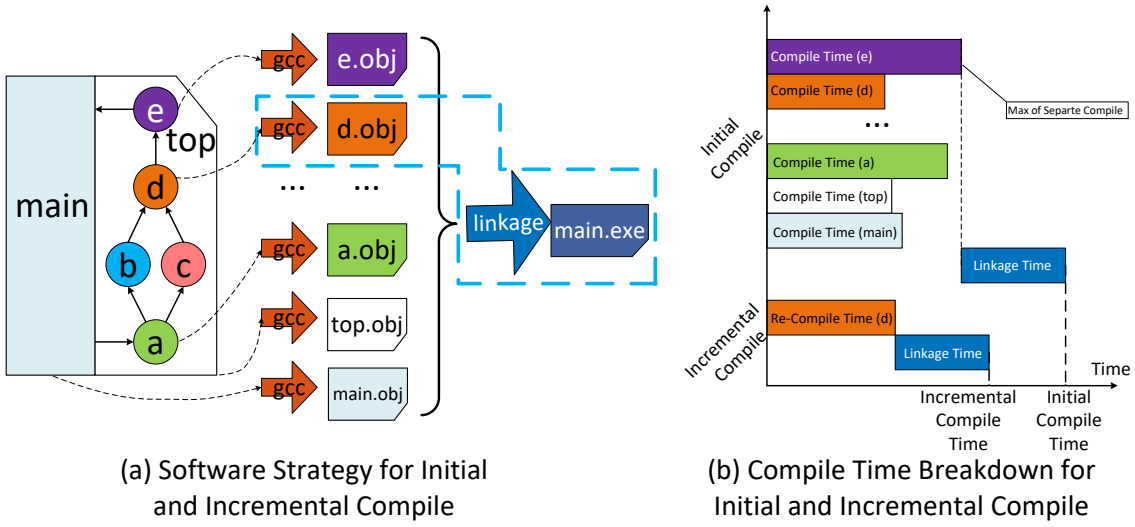


Figure 1.1: Rosetta HLS Benchmark Compilation Time

The reason behind this long compile time is that the state-of-art EDA tools try to co-optimize and compile the whole design in a monolithic way to get high-quality and area-efficient solutions. Even small changes force the EDA tools to recompile the entire design. As the scale of the application becomes large, EDA tools must solve large mapping problems. In contrast, modern software compiles in a sharply different way. When a C++ application is compiled for the first time, separate files can be compiled in parallel, shown in Figure 1.2(a). The compile time is determined by the worst compile time from all the separate blocks plus the linkage time shown in Figure 1.2(b). For every refinement



(a) Software Strategy for Initial and Incremental Compile

(b) Compile Time Breakdown for Initial and Incremental Compile

Figure 1.2: Software Compile Strategies and Compile Time Breakdown

iteration later, the software compiler only re-compiles the modified function(s) and re-links the objective(s) again. The dashed blue shape in Figure 1.2(a) shows the case when only function d is refined.

This dissertation describes a divide-and-conquer strategy to compile separate FPGA blocks to speed up the process from C++ sources to FPGA executable bitstreams. Separate compile is partially supported by vendor tools by out-of-context RTL synthesis [138] but is not supported directly for placement, routing, and bitstream generation. The Partial Reconfiguration (PR) is a potential technique to achieve our divide-and-conquer goal. Specifically, we can predefine a cluster of non-overlapping layout blocks on an FPGA chip, and map separate functions at C-level to these physical locations at the layout level. This process is non-trivial as hardware expertise is required to use PR well, which is not easily accessible for software programmers. Moreover, we note software linkage for CPUs cannot be trivially applied to FPGAs due to the fundamental difference between FPGA architectures and software models. C++ objectives are linked together in a temporal domain, where the machine code can be fetched by CPUs according to the instruction address at a different time for execution, while FPGA separate blocks are connected in a spatial

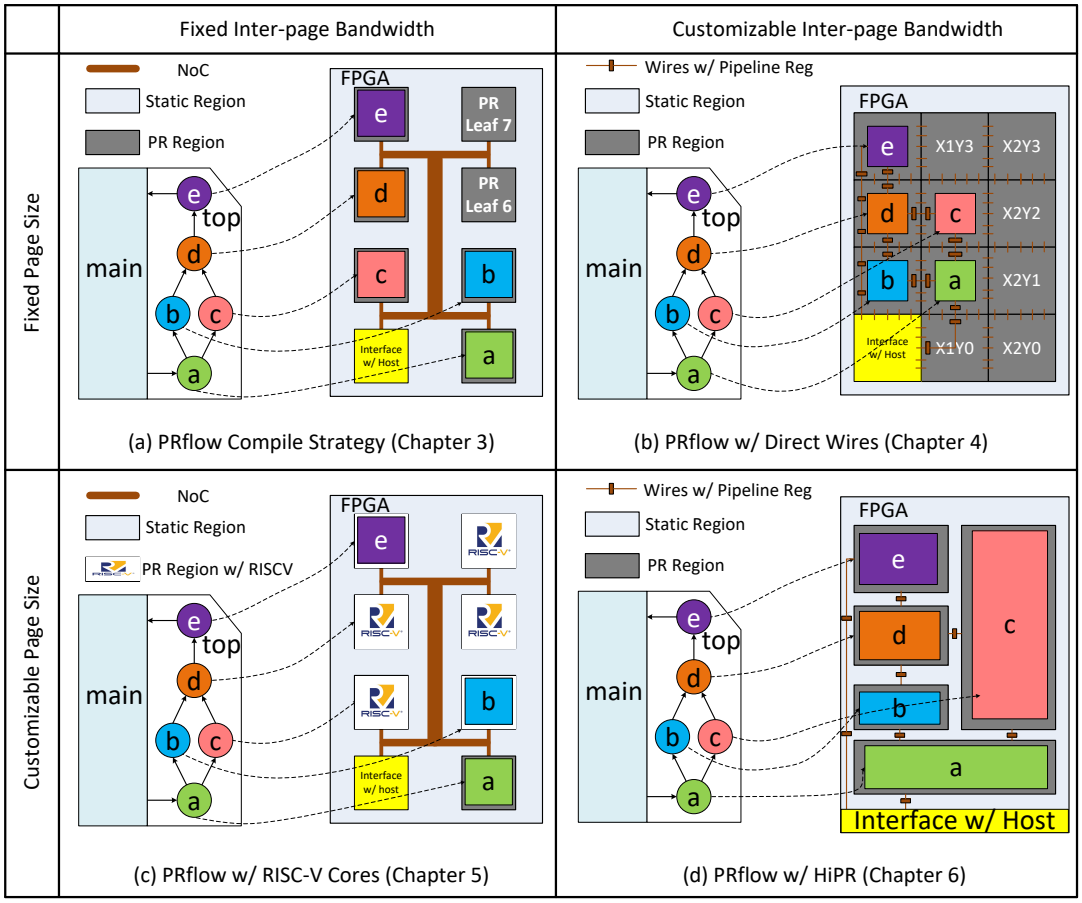


Figure 1.3: Hardware Compile Strategies

domain, where different blocks communicate with each other concurrently for execution. Therefore, this dissertation proposes a framework that not only automates the process from C++ to separate PR bitstreams but also provides several spatial linkage solutions, such as packet-switched network-on-a-chip (Chapter 3), direct wires (Chapter 4), and dedicated links (Chapter 6). These frameworks make the best use of existing commercial tools to reduce engineering efforts and guarantee the quality of results.

1.3 Divide-and-Conquer FPGA Compilation

1.3.1 PRflow

To compile FPGA in a divide-and-conquer manner, we first developed a framework called PRflow, which uses partial reconfiguration and a packet-switched network-on-a-chip (PSNoC) to isolate design components for separate compilation, dividing the FPGA capacity into a number of independent, partial reconfiguration regions. Components (IP blocks, computational operators) can be compiled into partial reconfigurable (PR) regions independently. The packet-switched network provides the communication linking that allows the independently-compiled blocks to interact. Before mapping applications, an overlay with a cluster of PR regions is compiled with a PSNoC to connect these PR regions together, as shown in Figure 1.3(a). With this fixed overlay, separate blocks of certain applications can be compiled in parallel. Since there are several steps to compile a C file to a partial bitstream (HLS, RTL synthesis, physical implementation, and bitstream generation), we abstract these steps and dependencies as jobs with Sun Grid Engine [94] on our own web of servers or with Slurm [43] on Google Cloud. For the compile time, PRflow can decrease the compilation time from hours to 24 minutes. However, two issues come along with PRflow: 1) the performance (execution time) of an application can be degraded by the limited inter-page communication bandwidth (the IO bandwidth between the NoC and a page); 2) separate blocks at C-level cannot be mapped to the fixed PR regions unless they are small enough to fit the PR region size.

1.3.2 PRflow_DW

For operators with high bandwidth requirements, the input/output data need to be timely demultiplexed/multiplexed before communicating with the PSNoC. For example, operator **a** has two output ports, but the page has only one input/output with the PSNoC in Figure 1.3(a). This IO-bottleneck can potentially degrade the performance. To address the performance degradation issue, we need to increase the inter-page bandwidth while preserving the quick separate compile. From Figure 1.4(a), we note that only a small portion of

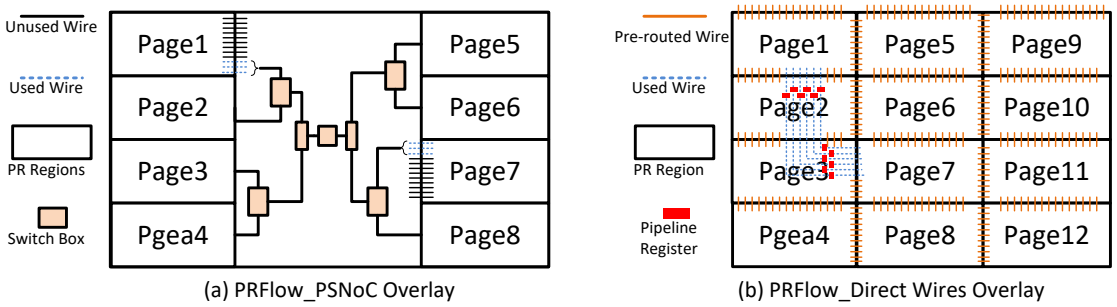


Figure 1.4: PSNoC vs. Direct Wires

the interconnect resource is used by the PSNoC. However, increasing the datawidth of the PSNoC to increase the bandwidth is not area efficient, since the bandwidth requirements between pages can be unbalanced. For example, it is possible that only page1 and page7 need high bandwidth. Therefore, we develop an overlay with a grid of PR regions connected by pre-routed wires between adjacent PR regions, shown in Figure 1.4(b). We pre-routed as many wires as possible under a frequency constraint. A coarse-grain placer and router are deployed to find a feasible placement for all the operators at C-level with two constraints: 1) PR regions should have enough resources to map the operators; 2) the stitched wires between adjacent PR regions are sufficient to construct pipeline routes to connect operators together. As the scale of the operators is small enough compared with bit-wise level netlist placement, it takes a negligible amount of time to find the placements. Since all the operators are still compiled in parallel, the application will continue to benefit from a short compile time. Operators are connected according to their requirements. Therefore, the IO bottleneck can be addressed and the performance can be improved. We call this framework PRflow_DW (**PRflow** with **D**irect **W**ires) shown in Figure 1.3(b). For each page, PRflow_DW compiles the mapped operator along with the route-through pipelined wires. As all the pages can be compiled independently on the cloud, we can still benefit from the divide-and-conquer strategy while improving performance.

1.3.3 PRflow_RSICV

It is common that developers may want to quickly get a functionally-correct application to run immediately as a baseline for later refinement and optimizations. However, the separate blocks at C-level cannot be mapped to the fixed PR regions unless they are small enough to fit the PR region size. This may leave some burden on the users to set up an immediately-executable application. Also, FPGA maps computation spatially, and resource consumption can blow up when the parallelism factor is large (e.g., unroll factor). However, users need to get quick returns during the initial verification stage, and 20 minutes of compilation time is still too long. To address this size-fit issue, we extend the PRflow overlay with RISC-V [118, 92] soft cores support (PRflow_RISCV), shown in Figure 1.3(c). Specifically, we pre-implement RISC-V cores on all the PR regions connected by the NoC. For the initial run, developers can map separate C blocks to different RISC-V cores within seconds. PRflow and PRflow_RISCV are not exclusive: developers can map all the operators first to RISC-V cores and incrementally map some operators to RISC-V cores and the other operators to hardware logic in PR regions. This hybrid implementation is meaningful since RISC-V cores can also act as probe modules similar to Integrated Logic Analyzer (ILA) [130]. We can map one operator we are interested in to the RISC-V core and use our support feature (hardware `$printf`) to send the debugging information back to the host for analysis on the run. Also, we can flexibly redirect data from the producer operator to one RISC-V core which can analyze the redirected information and send them continually to the consumer operator.

1.3.4 PRflow_HiPR

PRflow with Direct wires (Figure 1.3(b)) and PRflow with RISC-V soft cores (Figure 1.3(c)) can partially solve the bandwidth and size issues of PRflow (Figure 1.3(a)) respectively. However, PRflow_DW still suffers from fixed-page issues and PRflow_RISCV cannot fully exploit the hardware parallelism. Therefore, we present our final solution PRflow with HiPR (**Hi**gh-level **P**artial **R**econfiguration). PRflow_HiPR can solve both issues with ac-

ceptable initial compile time overhead shown in Figure 1.3(d). Instead of pre-compiling a fixed overlay, HiPR can floorplan individual locations for all the operators that are labeled as Partial Reconfiguration Function by *pragma*. Developers can define elastic ratio to intentionally create larger shapes for certain operators, which may be tuned up in the following developing steps. 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. 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. Therefore, custom overlays can be generated without human intervention for different applications. Both the size of PR regions and the wires between different PR regions can be customized according to the requirements. With HiPR, more developers can benefit from quick PR compilations without bottom-level hardware knowledge and expertise.

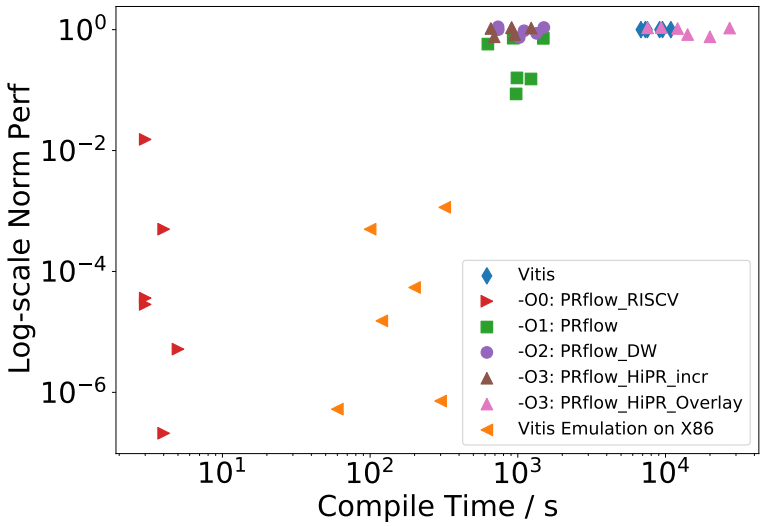


Figure 1.5: Key Results

1.3.5 Key Results and Summary

This dissertation applies the divide-and-conquer strategy to FPGA compilations. With different frameworks, we provide various compilation options similar to software. As shown in Figure 1.5: PRflow_RISCV (-O0) can run the application within seconds of compilation time and get a performance similar to X86 Emulation; PRflow can deliver good performance with 10–20 minutes of compilation time; PRflow_DW (-O2) can improve the performance by providing more inter-page bandwidth; PRflow_HiPR (-O3) can customize an application-specific overlay for incremental compilation with an acceptable overlay compilation overhead without performance loss. For the later incremental compilation, it takes the same order of compilation time as -O1 and -O2.

1.4 Dissertation Overview

The rest of this dissertation is organized as follows.

Chapter 2 presents the necessary background of modern reconfigurable device architectures and normal compilation flow. Next, the partial reconfiguration technique, the main method this dissertation leverages to speed up the compilation time, is introduced. Following are the prior works of different compilation acceleration methods. Finally, floorplan algorithms for partial reconfiguration are summarized, which motivates the lightweight simulated annealing algorithms for our HiPR framework.

Chapter 3 details the initial implementation and evaluation of the divide and conquer compilation strategy. It begins by elaborating on the key idea of using separate partial reconfiguration blocks and a Network-on-a-chip to accelerate the compile time for FPGAs, followed by the hardware library to construct the FPGA overlay. Then, it characterizes how vendor tools behave under different contexts, which determines some parameters for the overlay, such as the bus width of the NoC, PR page numbers, PR page size, etc. Next, it explains how the tool flow is constructed by leveraging existing tools, such as Makefile [42], Sun Grid Engines [94], Slurms on Google cloud [43], etc. The preliminary results in this chapter show that PRflow can accelerate the FPGA compilation by 6.4–10.9×. Finally, the

IO bottleneck and fixed-page-size issue are discussed.

Chapter 4 explains how the IO bottleneck is addressed by using direct wires overlays. The IO workload and compute workload are profiled over different benchmarks. New hardware libraries will be introduced to show how direct wires overlay can reduce the area overhead compared with PRflow overlay. A simple coarse-grain placer and router are explained to automate the operator assignments. The experiments in this chapter will show how the performance is improved by addressing the IO bottleneck.

RISC-V soft cores are presented in Chapter 5. Both the hardware and software libraries are proposed to show how RISC-V supports software libraries commonly used by FPGA HLS code with low memory consumption. A case study shows how RISC-V can perform quick functionality verification and on-chip debugging as a replacement for Integrated Logic Analyzer (ILA).

Chapter 6 introduces the final solutions to the IO bottleneck and fixed-size issues. A lightweight Simulated Annealing (SA) method is explained. The evaluation shows SA method can generate similar-quality of results to the Mixed-Integer Linear Programming (MILP) method with order-of-magnitude less execution time. The experiments of mapping Rosetta benchmarks show HiPR can accelerate the compile time without losing performance with an acceptable one-time compile time overhead.

Chapter 7 presents some meaningful future works. Chapter 8 highlights the main points of the dissertation and concludes.

1.5 Dissertation Contributions

Particularly, we made the contributions in this dissertation as follows:

- We introduce the idea of separate compilations and a linking network to apply the divide-and-conquer compilation strategy to FPGAs to accelerate the FPGA compilations. An FPGA compile framework is developed to abstract the FPGA chip as separate blocks connected by a network-on-a-chip. Compared to state-of-the-art vendor tools, this separate compile solution can accelerate the FPGA compilation by

6.4–10.9×. This part of the work is published in [123, 95] and released as open-source[1].

- We show the linking can be mapped directly to the FPGA and compiled in parallel with the computation. The routing capability in regions of the FPGA is profiled and is related to packet-switched NoC bandwidth. It is demonstrated that the direct-wire switch box routing can reduce the compilation time compared to monolithic design mapping without sacrificing performance. We also show the potential to unify logic and switch partial reconfiguration regions. This work is published in [120].

- We show how the same source code can be compiled to the FPGA regions or processor. RISC-V soft cores are utilized and applied for quick on-chip functionality verification and debugging. The software library to support streaming class by vendors are proposed. This work is published in [122, 86].

- We bridge the gap between HLS and the 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[2] 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 with the full set of Rosetta benchmarks on the Alveo U50 card to reduce compilation time by 3.5–7.6×. This work is published in [121].

[1] https://github.com/icgrp/pld2022.git
[2] https://github.com/icgrp/hipr.git

Chapter 2

Background

Benefiting from the hardware-programmable features, FPGA developers can customize the application instances more flexibly. However, FPGA compilation is intrinsically slow as it stems from ASIC design flow, such as placement and routing. In a sense, the FPGA implementation problem is even more complex than ASIC since only fixed placement locations and limited tracks of wires can be selected for placement and routing.

This chapter will present the background on which the rest of our work is based. If the readers are familiar with the basic architecture of FPGAs, they can start from Chapter 2.2

2.1 FPGA Architecture

The traditional FPGA architecture is shown in Figure 2.1. Look-Up-Tables (LUTs) are the most basic elements. A 4-LUT can be configured to map arbitrary 4-input Boolean equations. The output of a LUT can be configured as combinational or sequential by a multiplexer within the Configurable Logic Block (CLB). Through reconfigurable interconnects, the CLBs plus interconnects architecture theoretically can map any logic. However, as memory and multiplier are common enough in general computing, and it is not efficient to use CLBs to map them, modern FPGAs also integrate Block RAMs (BRAMs) and Digital Signal Processors (DSPs) inside. Leveraging these AISC-like IPs to map corresponding compute elements (memories and multipliers), both area efficiency and timing can be improved.

In addition to these essential elements, modern commercial FPGAs have much more

complex architectures. We will use the UltraSacle+ FPGAs from Xilinx (AMD) as an example.

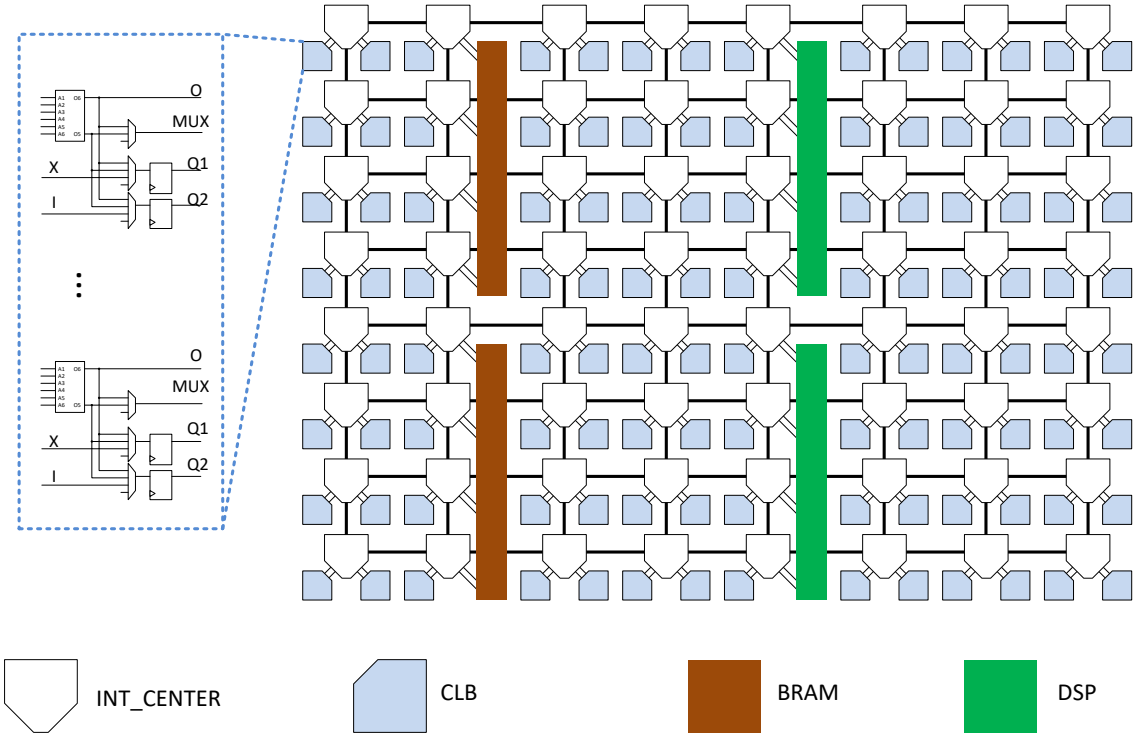


Figure 2.1: Basic FPGA Architecture

Stacked-Silicon Device

The latest Xilinx FPGA devices provide Stacked Silicon Interconnect (SSI) technology, enabling vendors to create reconfigurable devices with more capacity by stacking several dies, called Super Logic Regions (SLR), on one package substrate. For example, the Alveo U250 Data-center card contains 4 SLRs in a stacked style with SLR0 connected to SLR1, SLR1 connected to SLR2, and SLR2 connected to SLR3. A silicon interposer (LAG_LAG tile in layout) is responsible for high bandwidth transfer between SLRs, but it can also add more latency. Even though SSI can provide users with high-capacity devices, users still need to carefully floorplan their design to avoid large data transfer between SLRs. This

might affect the PR region definition in the rest of the work, which is the key technique we use to accelerate the compilation.

URAM and HBM

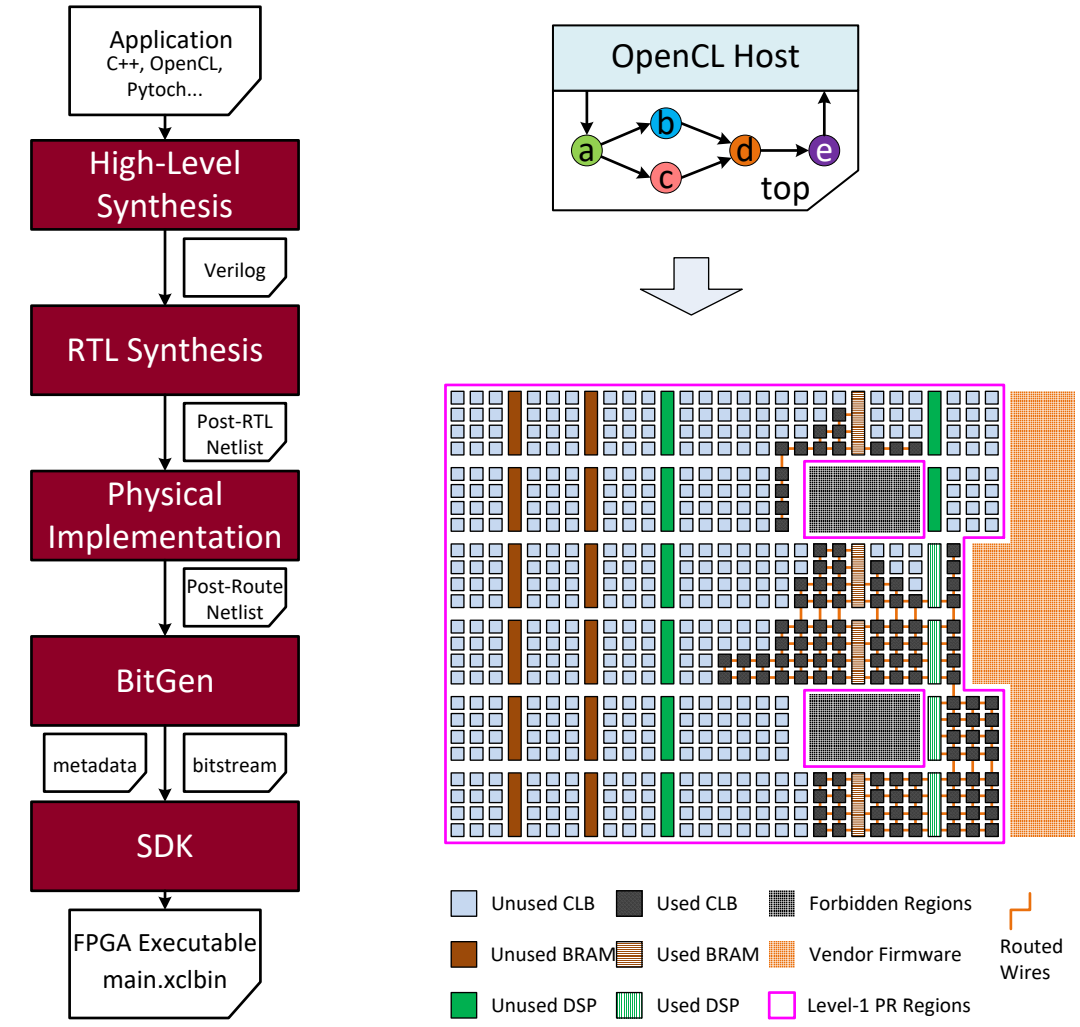
Traditional FPGAs include BRAMs and Distributed RAMs for on-chip buffering. However, reconfigurable devices are processing more data at a greater rate. When hundreds of megabytes of data need to be handled, external DDR memories are adopted, which increase the complexity of the design and slow down the data transfer speed by the limited off-chip bandwidth. UltraRAM is a new flexible memory, as every URAM block is a dual synchronous port memory of 4,096 deep and 72 bits wide [126]. Cascading URAMs can provide users with more than 500Mb lightweight and power-efficient on-chop buffers.

High Bandwidth Memory (HBM) is another type of large onboard memory that has 20× (460 GB/s) more bandwidth than DDR DIMMs. The HBM memory is basically 3D-stacked synchronous dynamic random-access memory (SDRAM). For Alveo U50 [124], it has 8 GB HBM with 32 Pseudo Channels (PC), also called banks, which can independently access 256 MB range at 14.375 GB/s max theoretical bandwidth per bank. Configured properly, the bandwidth between HBM and FPGA fabric can be up to 460 GB/s (14.375 GB/s*32). This technique can enable users to integrate low bandwidth/watt compute modules into one device.

Therefore, modern FPGAs are featured for high integration with more components (HBM, PCIe, etc.), high logic capacity, and high clock frequency (900MHz IPs), which can map versatile applications, including data processing, image processing, and machine learning.

2.2 FPGA Compilation vs. Software Compilation

FPGA compilation flow is shown in Figure 2.2. Here we take Xilinx Vitis [131] flow as an example. We usually develop our applications in high-level languages, such as C/C++. The High-Level-Synthesis (HLS) tool parses the C/C++ code and synthesizes it into Hardware Deception Language (HDL), such as Verilog or VHDL. If device-related IPs are generated



(a) Modern FPGA Compile Flow

(b) Modern FPGA Mapping

Figure 2.2: Xilinx FPGA CAD Flow

(e.g., floating division/multiplication modules), some TCL files can be generated, guiding the synthesis tool on initiating the IP instances later. Next, the Verilog/VHDL code is compiled by the RTL synthesis tools (Vivado) to intermediate representatives, such as a netlist of primitives. Closely coupled to the device and technology, these primitives will be mapped to the different on-chip resources (LUTs, BRAMs, and DSPs). Different vendors have different devices so that the same primitives can be mapped to different on-chip

resources. Next, these on-chip resources will be physically optimized, placed, and routed within layout resource constraints. Vendors now accept some constraints to guide their tool to perform placement and routing better. However, most of the work is still conducted by the CAD tool, usually by solving NP-hard problems with high complexity [119], which can take hours to days.

Typically, FPGA compilation maps the entire design once, which is good for quality, as compilers can perform cross-module optimizations. However, this also means compilers are solving giant problems — millions of individual logic elements on modern data-center FPGAs — for any changes. Due to the ASIC-like feature of FPGAs, the bitwise placement and routing can be performed repeatedly with tiny local changes.

However, compilation for processors is typically fast. Historically, compiler developers have intentionally eschewed non-linear algorithms to keep the compilation fast. Furthermore, software compilers do not need to deal with bit-wise spatial problems (placement and routing), making the problems easy to solve. In addition, software compilers naturally support incremental compilation well [14, 7]. For example, the C program is often written in several separate files. Each file represents one sub-function. Therefore, these individual files can be compiled separately and linked to make an executable file. If one sub-function is changed, only the modified files are re-compiled and linked quickly to re-generate a new executable file. Finally, software compilers also provide different optimization levels (`-O0`–`-O3`) to support better trade-offs between compilation time and quality.

2.3 Partial Reconfiguration

Partial Reconfigure (PR) technique is widely supported in modern FPGAs, by which only portions of the FPGAs need to be reconfigured while the rest of the FPGAs continues to run [51, 134]. The bitstream loading time can be shortened as the size of the bitstream is roughly proportional to the amount of the logic being reconfigured, which can be several orders smaller (usually several kilobytes) than the complete bitstreams (usually hundreds of megabytes). Conventional usage of PR includes bitstream loading time reduction, area

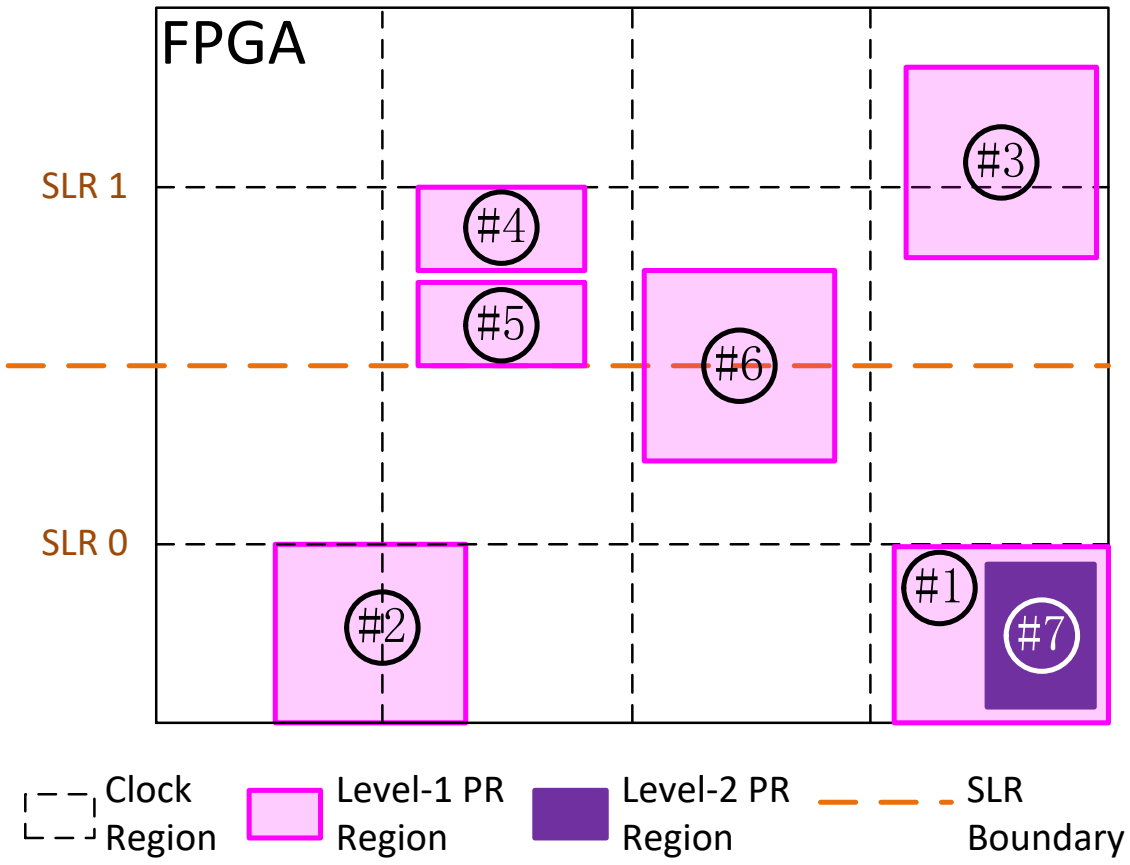


Figure 2.3: Partial Reconfiguration Constraints – #1 is legal and recommended; #2 and # 3 are legal but not recommended as timing can be degraded across clock regions; #4 and #5 are illegal since two PR regions cannot share a column of resources within the same clock region; #6 is strongly not recommended as timing can be degraded and routing can fail due to the limited number of connecting wires by LAGUNA cells

reduction [117, 85, 83], and design specialization [35, 113]. For Xilinx flow, there are mainly two constraints for PR region: 1) no overlapping area is allowed between two PR regions; 2) one column of resources in one clock region cannot be shared by two PR regions. Figure 2.3 show different cases for PR region definitions: #1 is legal and recommended; #2 and #3 are legal but not recommended as timing can be degraded across clock regions; #4 and #5 are illegal since two PR regions cannot share a column of resources within the same clock region; #6 is strongly not recommended as timing can be degraded, and routing can fail

due to the limited number of connecting wires by LAGUNA cells.

Xilinx recently released Dynamic Function eXchange (DFX) [134, 139] with two essential features: 1) Hierarchical PR definition; 2) abstract shell design checkpoint. Hierarchical PR means users can define sub-PR regions in a previously defined PR region without re-compiling the logic outside of the previously defined PR region. In Figure 2.3, we can define PR region #7 inside PR region #1 without re-compiling the other PR regions. Abstract shell means an individual context design checkpoint for each PR region will be generated for later implementation, which only contains logic and wires related to one PR region. In contrast, one giant context design checkpoint will be generated for all the PR regions, which contains logic and wires related to all the PR regions for previous PR without the abstract shell technique [127]. Abstract shell technique is valuable to reduce the implementation time since loading the context design checkpoint for specific PR regions can be timing-consuming [123].

Vendors tools provide interfaces to support logic mapped to manually defined PR regions. However, defining PR regions and partitioning the RTL logic into separate PR logic is a non-trivial step requiring bottom-level hardware expertise. We will build a high-level interface that allows the users to define PR functions at the C level instead of at the hardware Verilog level in Chapter 6. We abstract away the low-level details and relieve the software users of tedious and error-prone work so that more software users can benefit from the PR incremental compile for FPGAs to accelerate the edit-compile-debug cycle.

2.4 FPGA Compilation Acceleration

Various frameworks have been brought forward to accelerate the traditional FPGA compilations. One path to faster compilation has been to pre-define overlay architectures for the FPGA [11, 64, 115, 65, 66, 40, 144, 28, 80, 52]. Pre-mapping macro components have been demonstrated to be effective to reduce FPGA compile time. HMflow [74, 75] exploited pre-implemented hard macros (internally placed and routed), macro-level floorplan, and custom routing to assemble DSP designs from System Generator by Xilinx [132]. As the macros

already include hundreds to thousands of CLB slices, the coarse-grain placement for macros is an order of magnitude simpler than bit-wise LUT-level placements. The custom router can stitch these hard macros back together for fast mapping instead of aggressively optimizing for strict timing constraints. Just-in-Time Assembly of Accelerators (JITA) presents overlays with different numbers of PR tiles (2×2 or 3×3) connected by the nearest neighbor bus [81]. Each PR tile is coupled with a switch box that can route bus signals across different PR tiles. Each tile is sized at 9,600 LUTs, 360KB BRAMs, and 80 DSPs. Different compute primitives (e.g., REDUCE, VMUL, MM, AVG, etc.) are implemented on all the PR tiles, and the corresponding partial bitstreams are stored as executables in the Domain Specific Language (DSL) Library. The interpreter can construct different accelerators by loading different primitives into PR tiles and connecting these primitives with switch boxes. Therefore, hardware compile time is replaced by bitstream loading time. Both HMflow and JITA can accelerate compile time by pre-implement separate layout blocks. However, the performance and area efficiency strongly rely on the matchup between the pre-compiled primitives and target accelerators. These methods, in fact, sacrifice FPGA hardware flexibility.

Dividing FPGA into separately managed physical regions is also explored in [12, 19, 21, 143, 63, 83, 85, 97, 144]. However, these works do not address compile time reduction or support high-level synthesis from C. Cascade [104] and SYNERGY [70] both target accelerating FPGA compile time and improving FPGA developing experience. Cascade [104] is an open-source framework that can compile Verilog code with unsynthesizable primitives (`$printf` or `$finish`) to many small pieces (subprogram). These subprograms can initially be mapped to software and be executed in the form of simulation. The scheduler searches the binary caches to find the FPGA bitstream for each subprogram. If the corresponding binary can be found, the subprogram can be moved to FPGA fabrics for execution. Otherwise, the controller will compile the subprogram into bitstream under the hood without interrupting the existing program execution. The users only see the program execution runs immediately and is accelerated over time, and finally, all the subprograms are offloaded into hardware.

Cascade bridges the gap between software and hardware programming for FPGA developers. However, Cascade's strategy is to hide the compilation time behind software simulation rather than decrease the absolute compilation time. This does not help much for the incremental scenario, where lengthy compilation for each case (usually many trial cases) is still unavoidable. Moreover, the performance deteriorated by 3×, and the input source is still Verilog, which has very different coding styles for software programmers. SYNERGY [70] is a compiler/runtime-based solution based on Cascade [104] and AmorphOS [63], which can transparently transform the Verilog program to a distributed-system-like intermediate representation (IRs). These IRs can easily be executed either by software or native FPGAs. Benefiting from extended Cascade and AmorphOS, SYNERGY can also perform workload migration, hardware accelerator suspend/resume, and spatial/temporal multiplexing between different tenants. The evaluations show SYNERGY can offload computing instances across a cluster of Altera SoCs or Xilinx FPGAs on Amazon F1 with the ability to suspend and resume programs, which speeds up the virtualized performance within 3-4×. However, SYNERGY does not solve the long compilation issue. As long as the scheduling is changed, the whole FPGA chip needs to be recompiled monolithically, which greatly increases the runtime overhead. Even though the recompiled bitstream caches can relieve the compile time overhead, it does not decrease the compile time for the incremental compilation. Grigore et al. [46] proposed a toolflow to automate the generation of partially reconfigurable modules from the MaxJ language to bitstreams. However, the toolflow heavily relies on GoAhead [29] and Xilinx ISE, which are incompatible with modern FPGA vendor tools, and the compilation time is not considered. Seiba supports processor integration with compiled FPGA logic to accelerate the edit-compile-debug loop [116]. Rather than accelerating the initial development and iterations before the design is suitable for hardware mapping, Seiba supports iterative improvement after the design has been initially mapped. Mapping an application to soft CPU cores on FPGAs provides a ramp-up solution from pure software simulation to pure hardware implementation. This dissertation in Chapter 5 will show how to use pre-compiled RISC-V [92] soft cores to achieve this goal.

The recent Xilinx-released open-source work RapidWright [73], an updated version of RapidSmith [76], supports manipulating design checkpoints from Vivado and is able to seamlessly interact with Vivado for low-level modifications. RapidWright enables users to perform more low-level manipulations on FPGA layout and makes many excellent works possible [147, 48, 82, 143, 84, 68]. RapidStream [48] can accelerate the compile time by leveraging RapidWright [73] to perform parallel compilation from HLS code to bitstreams. The application is pre-divided into separate sub-functions linked by streaming interfaces, which makes it possible to separately compile these sub-functions until post-placed&routed netlists. However, to generate an FPGA executable bitstream, inter-block routing is still needed to stitch the separate blocks together, which is hard to be performed in parallel. A complete bitstream will be generated for FPGA execution, which may take 30 minutes to one hour for modern data-center FPGAs. This inter-block routing and complete bitstream generation may limit the compilation time speedup in practice.

2.5 Floorplan for Partial Reconfiguration

The Floorplan is the key to bridging the gap between RTL synthesis (generated by HLS or manually prepared) and placement-and-route implementation, and there is a significant body of literature on PR floorplanning [37, 106, 6, 87, 88, 30, 112, 90, 102, 5, 6, 107]. Taking into account both the heterogeneous resource distributions and PR constraints for modern FPGAs, many floorplanners use heuristic methods [8, 99, 111]. Bolchini et al. [8] propose a floorplanner, based on an accurate model of the devices to find an optimal solution for reconfigurable devices, which takes into account both heterogeneous resource distribution and reconfiguration constraints. Simulated Annealing (SA) algorithm is adopted to explore a reduced search space represented by sequence pair [89]. The wire length can be optimized by 12–29% w.r.t [87]. A greedy floorplan method (Columnar Kernel Tessellation) is proposed in [111] to reduce resource wastage. It takes into account PR resource wastage, heterogeneous resource distributions, and reconfiguration time. The cost function is the weighted summation of waste resources and the total Manhattan distance. The modules

with DSP and BRAMs have the priority to be placed. Columnar Kernel Tessellation can reduce resource wastage. The case study shows it has less resource wastage compared with [88]. A Genetic Algorithm (GA) is adopted in [100] to explore wider feasible solutions. Experiments with the 20 largest MCNC benchmarks show 17% improvements on critical path at the cost of 2% area and 8% runtime, compared with Xilinx's early access partial reconfiguration design flow. Analytic methods, such Mixed-Integer Linear Programming (MILP) and Nonlinear Integer Programming (NLP), have recently been brought forward to generate global optimal solutions [100, 101, 105, 91]. The MILP-based floorplanner [100, 101] 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. Rabozzi et al. aggregate the FPGA device into coarse grain partitions according to the resource types to reduce the design space. Mixed-integer linear constraints are adopted to guarantee no overlapping area between two PR regions. Two compile strategies are proposed: [HO] (Heuristic-Optimal) and [O] (Optimal). [HO] can find a feasible solution quickly, while [O] can find the global optimal solution with a significantly long execution time when the scale of the problem is large. FLORA [105] is another MILP-based floorplan tool that takes into account the more realistic PR constraints and adopts a fine-grained model for modern FPGAs. While the analytic (MILP) method can outperform the heuristic method with the ability to find the global optimal solution, it suffers from a long execution time and poor scaling with problem size. Hence, HiPR adopts the SA-based floorplanning algorithm to accelerate the compile time, extending the SA by considering modern hierarchical DFX constraints (detailed in Chapter 6).

2.6 Latency-Insensitive Circuits

Latency Insensitive Circuits (LIC) have been proven to have good timing performance and design scalability in FPGA design [59, 15, 31, 31, 48, 17, 1]. In Kahn Processing Network (KPN) [59], the basic compute kernels are abstracted as autonomous computing stations connected in a network by communication links. Each autonomous station only

processes the incoming data from the input links and produces data to the output links. The autonomous stations are implemented in many works [95, 48, 86, 121, 122] by defining streaming interfaces as the arguments for the functions. Since the interconnect links can be mapped to FIFOs or relay stations [10, 120] in the hardware, it can isolate the critical paths inside the autonomous stations, which avoids cross-station long wires and improves timing [19, 31, 31].

Dynamically Scheduled Circuits (DSC) [56, 22, 55, 57, 23, 58] have been brought forward as a complementary for static scheduled HLS-generated circuits to improve performance (clock latency). Both LIC and DSC use handshake interfaces (valid and ready) to connect modules, but DSC targets finer-grain elastic components, such as elastic buffers, elastic FIFOs, forks, branches, and so on, while LIC targets coarse-grain compute kernels. By generating a particular form of latency-insensitive synchronous circuits, the conservative unnecessary data dependency can be avoided during the execution time, improving the performance at the cost of orders of magnitude area overhead. However, DSC can still be regarded as the first step toward high-performance HLS design.

For this dissertation, we use KPN as our computing model, and we construct the application as a cluster of autonomous C/C++ functions connected by streaming links (detailed in Chapter 3.2). For each autonomous C/C++ function, we still use the traditional static schedule HLS compiler (Vitis_HLS) for better area efficiency.

Chapter 3

Divide-and-Conquer Compilation

Reconfigurable devices allow developers to customize their applications with huge resource flexibility and energy efficiency. However, the long compilation time hinders the broad deployment of FPGAs. Whist vendors tools, such as SDSoC [129], Vitis [131], and OpenCL [54], tend to improve the coding efficiency, the placement, routing, and bitstream generation times are still getting increased since the physical scale of FPGA chips is increasing. FPGA compilation is slow because the EDA tools compile the entire accelerators in a monolithic way, where super-linear algorithms are adopted to find optimal solutions.

We argue that FPGA compilations can be greatly accelerated by adopting the good strategy from software compilation: *divide-and-conquer*. In software, a large application is described by small sub-functions written in different source files, which can be compiled separately and linked back together at the end shown in Figure 3.1. Incremental compilation is possible as only the modified files need to be recompiled later. For FPGAs, the separate compilation is supported only at the RTL synthesis stage. Different Verilog modules can be separately compiled depending on the thread numbers of the workstation. For the implementation stage, the entire netlist is placed and routed in a monolithic way.

We present PRflow, a framework that uses partial reconfiguration and packet-switched overlay network to isolate design components for separate compilations. Instead of mapping applications directly to raw FPGAs, we divide the FPGA layout into numbers of partial reconfigurable regions, which are connected by a packet-switched network through universal streaming interfaces. The components for an application (C/Verilog files) can be mapped

separately to different physical partial reconfigurable regions. As all the components communicate with others only via the NoC, these compilations are completely independent. This is of great value, as the compilation of one application is not limited to one workstation but can be distributed to a cluster of servers on the cloud. Therefore, the speedup is not limited by the number of threads of one workstation. The separate-mapped modules can be linked together by configuring the packet-switched network on the chip in a software manner without physically routing the interconnections again, similar to the linkage stage in software compilation. In fact, the incremental compilation is naturally supported by PRflow, as only the modified modules need to be recompiled within the specific PR regions. The new PR implementation modules can be dynamically linked together by configuring the NoC in seconds.

In this chapter, we begin with the compute model and introduce the basic terms for PRflow. The commercial tools will be characterized first, as it determines how the PRflow is designed. Next, we elaborate on the framework of PRflow, including PR region definition and Packet-Switched Network-on-a-Chip (PSNoC) design. We will evaluate PRflow by mapping Rosetta HLS benchmarks to Alveo U50 data-center card equipped with a Xilinx XCU50 16nm FinFET+ FPGA.

3.1 Divide-and-Conquer with PR technique

3.1.1 Divide-and-Conquer Requirements

Divide-and-Conquer is a common strategy in software compilation. For instance, we usually write different source files for different sub-functions for the C program. All the source files can be compiled independently by `gcc` to objective files that are finally linked to make an executable file (Figure 3.1). For incremental compilation, only the modified files are recompiled and linked again to re-generate the executable file as the dashed blue box shows in the dotted block in Figure 3.1. Consequently, the incremental compiles take much less time than the initial compile.

Two key features for fast software compilation are separate compilations and quick link-

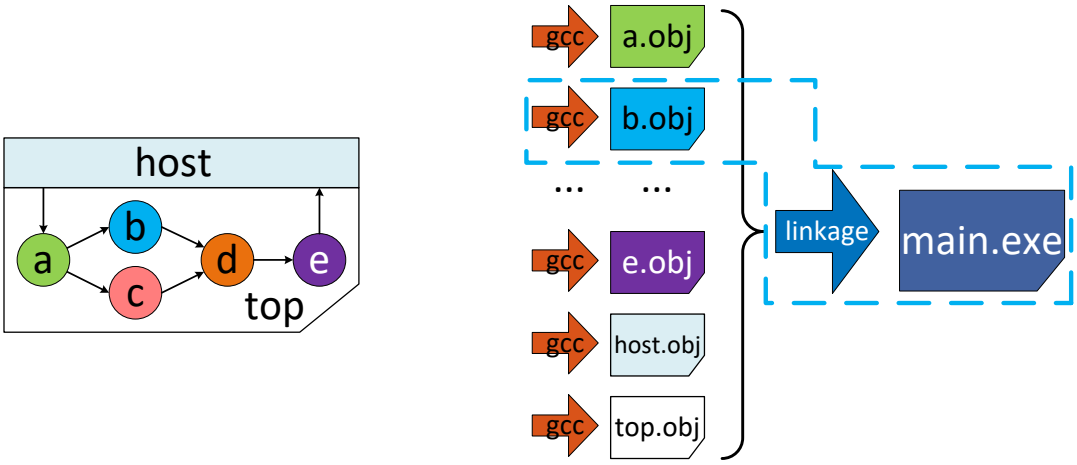


Figure 3.1: Separate Compile and Linkage in Software

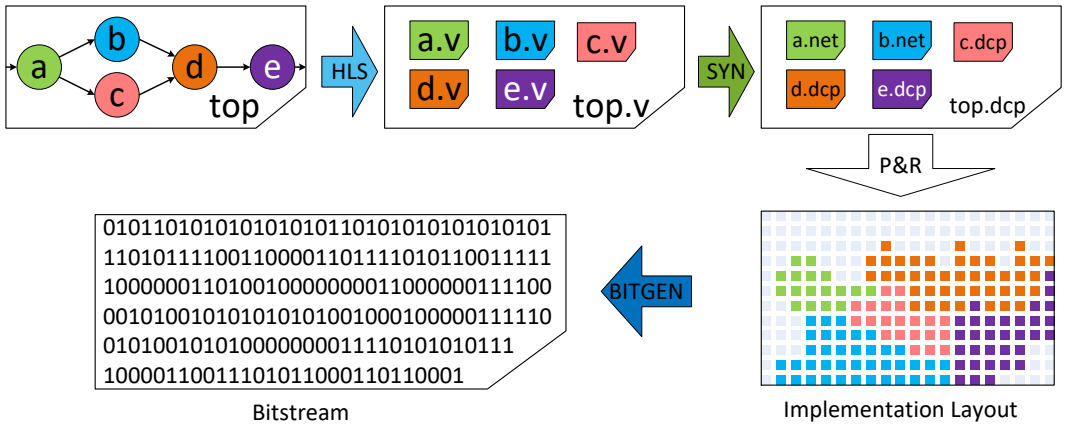


Figure 3.2: Hardware Separate Compile

age. Unfortunately, these two features are not well-supported by commercial FPGA tools. While high-level synthesis and RTL synthesis now support separate compilations, the physical implementation (placement&routing) and bitstream generations, which are the most time-consuming steps, can only be conducted in a monolithic way, shown in Figure 3.2. Consequently, incremental compiles are not well-supported, especially for physical implementation[3].

[3]Vivado supports the incremental implementation strategy, but some criteria must be met between current and previous compilations: 1) 94% cell matching; 90% net matching; 3) WNS > -0.250ns [136].

3.1.2 Separate Compiles with PR

To completely apply the divide-and-conquer strategy to hardware compilation, we need to support separate physical implementation and bitstream generation in addition to HLS and RTL synthesis. Partial Reconfiguration (PR) [50, 134] is a technique we can leverage for two reasons. First, PR is a common feature widely supported by commercial FPGAs, ensuring our methodology can be applied to most of the FPGAs. Second, PR can isolate different hardware modules completely by defining separate partial reconfigurable regions, where independent physical implementation and bitstream generation can be performed in parallel [78].

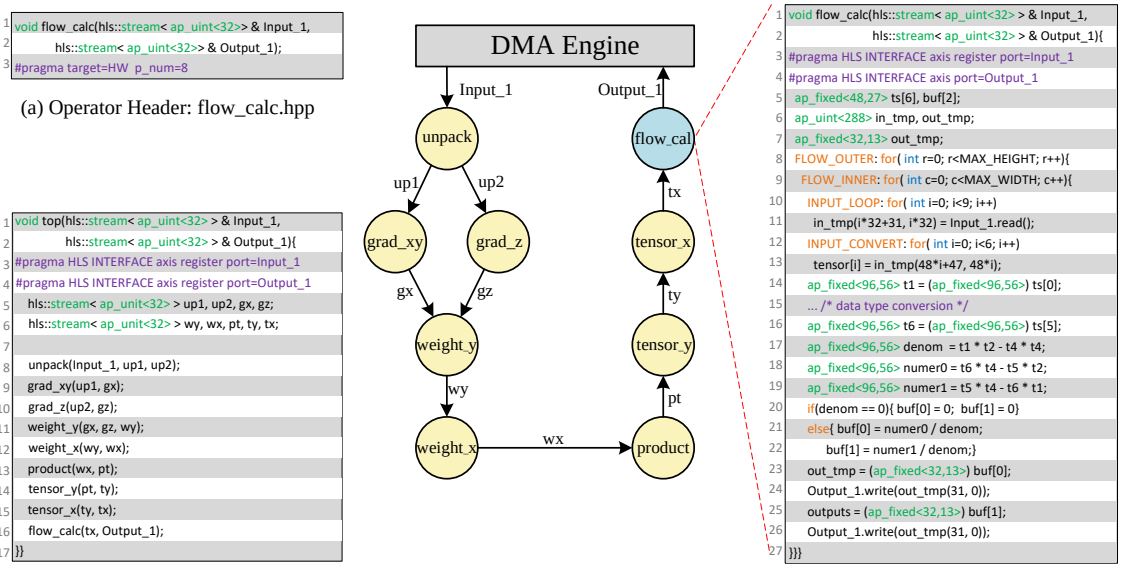
3.1.3 Quick Linkage with a NoC

With pre-divided PR regions, we can isolate the physical implementation and bitstream generation within each local area. However, different PR modules have to communicate with each other via the interconnect wires on the static region, which are fixed for one PR layout. These global interconnect wires have to be re-routed for different applications. To overcome this limitation, we propose to use a Packet-Switched Network-on-a-Chip (PSNoC) to connect the pre-defined PR regions with uniform interfaces. The topology we use is Deflected Butterfly-Fat-tree (BFT) from [61, 62]. Instead of re-routing the global wires for interconnection requirements, we pack output data into packets with a header, which indicates where the packets should go. The PSNoC can dynamically route the packets to the right destination according to the header information. In a sense, this PSNoC acts as a hardware linkage.

3.2 Dataflow Composition Model

We assume a dataflow stream compute model based on Kahn Process Networks [31, 41, 15, 32]. The design is decomposed into computation or memory operators connected by dataflow stream links. The stream links abstract away the timing and implementation details so that each operator only processes the incoming data from the input stream links and produces the processed data to the output stream links without the knowledge of the

other operators. One operator is mapped exclusively to a pblock page. The stream links make separate compilations for the pages possible. Variable latency in the interconnect channels arises from the placement of the operators or the congestion on the PSNoC.



(a) Operator Header: flow_calc.hpp

```
1  void flow_calc(hls::stream< ap_uint<32>> & Input_1,
2          hls::stream< ap_uint<32>> & Output_1);
3  #pragma target=HW  p_num=8
```

(b) Dataflow Graph Description: top.cpp

```
1  void top(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    hls::stream< ap_uint<32> > up1, up2, gx, gz;
6    hls::stream< ap_uint<32> > wy, wx, pt, ty, tx;
7
8    unpack(Input_1, up1, up2);
9    grad_xy(up1, gx);
10   grad_z(up2, gz);
11   weight_y(gx, gz, wy);
12   weight_x(wy, wx);
13   product(wx, pt);
14   tensor_y(pt, ty);
15   tensor_x(ty, tx);
16   flow_calc(tx, Output_1);
17 }}
```

(c) Dataflow Graph

(d) Operator Function: flow_calc.cpp

```
1  void flow_calc(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 port=Output_1
5    ap_fixed<48,27> ts[6], buf[2];
6    ap_uint<288> in_tmp, out_tmp;
7    ap_fixed<32,13> out_tmp;
8    FLOW_OUTER: for( int r=0; r<MAX_HEIGHT; r++){
9      FLOW_INNER: for( int c=0; c<MAX_WIDTH; c++){
10       INPUT_LOOP: for( int i=0; i<9; i++)
11         in_tmp(i*32+31, i*32) = Input_1.read();
12       INPUT_CONVERT: for( int i=0; i<6; i++)
13         tensor[i] = in_tmp(48*i+47, 48*i);
14       ap_fixed<96,56> t1 = (ap_fixed<96,56>) ts[0];
15       ... /* data type conversion */
16       ap_fixed<96,56> t6 = (ap_fixed<96,56>) ts[5];
17       ap_fixed<96,56> denom  = t1 * t2 - t4 * t4;
18       ap_fixed<96,56> numer0 = t6 * t4 - t5 * t2;
19       ap_fixed<96,56> numer1 = t5 * t4 - t6 * t1;
20       if(denom == 0){ buf[0] = 0;  buf[1] = 0}
21       else{ buf[0] = numer0 / denom;
22           buf[1] = numer1 / denom;}
23       out_tmp = (ap_fixed<32,13>) buf[0];
24       Output_1.write(out_tmp(31, 0));
25       outputs = (ap_fixed<32,13>) buf[1];
26       Output_1.write(out_tmp(31, 0));
27 }}}
```

Figure 3.3: PRflow Code Discipline Example

There is some discipline required to design operators. Our discipline includes:

- `hls::streams` and associated API operations (Chapter 3.2.1) are used for all communication.
- Operators are limited to 7 input streams and 7 output streams; this is not fundamental, but a particular system will need to pick some MAX_STREAMS to support the hardware design. This limit impacts the packet headers for the BFT.
- Operators should obey standard HLS prohibitions such as no allocation or recursion;

3.2.1 Streams

Code compiled to the FPGA using native Vitis_HLS `hls::stream` implementation. Figure 3.3(d) Lines 11, 24, and 26 show how to use this class type to read and write data.

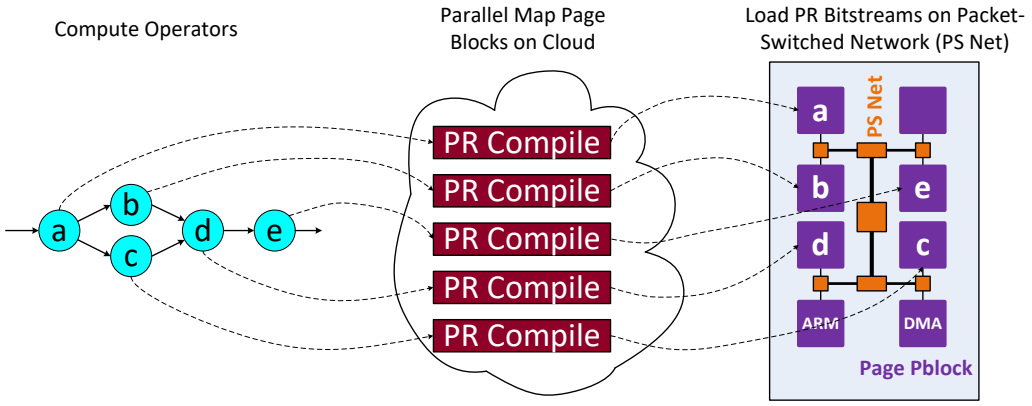


Figure 3.4: Separate Compilation Strategy

3.2.2 Application Composition

We support a stylized discipline for the top-level composition of the operator graph along with pragmas that specify where each operator is mapped. Top-level operator composition instantiates operators as function calls (See Figure 3.3(b)) and can be compiled with Vitis_HLS for a monolithic compilation; alternately, it can be compiled with our tools to generate the linking graph needed to configure the PSNoC. An operator with mapping control directives is shown in Figure 3.3(a) Line 3. Each operator has a line with a target specification. Changing the p_num will change which page an operator is mapped to.

3.3 Framework

The basic idea of separate compilation is to divide the FPGA chip into a set of PR regions with fixed size and map separate blocks in the user's application to these PR regions independently (Figure 3.4). We call the physical partial configurable regions *pages* and the logic we separately map to these PR regions *operators*. Once the mapping is completed, separate bitstreams for corresponding pages are loaded onto the FPGA. A packet-switched network is responsible for the communications between pages.

3.3.1 Packet-Switched Network

We adopt a packet-switched network to connect different PR pages so that there is no need to place and route any dedicated links between pages. The pblock page is easy to be configured with the addresses for the destination downstream modules, and the interface logic wraps output data with destination addresses as packets, which enter the packet-switched network. We choose a deflection-routed, packet-switched network since it is lightweight on modern FPGAs [62]. Specifically, we use Butterfly Fat Tree (BFT) topology [77, 61], which can be parameterized to generate versions with different internal bandwidths according to Rent's Rule [71].

3.3.2 Network Interface

The operators of the application communicate through a streaming interface. A page interface is used to connect the operators to the deflection-routed packet-switched network shown in Figure 3.5. The page interface includes input and output FIFOs that are used to receive packets from BFT or send packets to the BFT. These FIFOs decouple the network and user logic into 2 different clock domains so that they can run at different frequencies. The minimal BFT does not deal with flow control nor the in-order transfer of the packets. Therefore, we add sequence numbers to packets for ordering and design the page interface logic the store the data from the BFT in order into FIFOs. The windowed acknowledgment scheme [39] is adopted for flow control to avoid overwhelming the BFT with traffic congestion. Since the page interface resides in the PR regions along with the user logic, it can be tuned to adapt to use logic with different requirements, such as the number of input/output ports, the depth of the FIFOs, and the window size for flow control. The packets include the destination address, the port identifier, the sequence number, and the payload (data). In this work, we use BFT with a packet width of 48 (5 bits for address, 4 bits for ports, 7 bits for sequence, and 32 bits for payload).

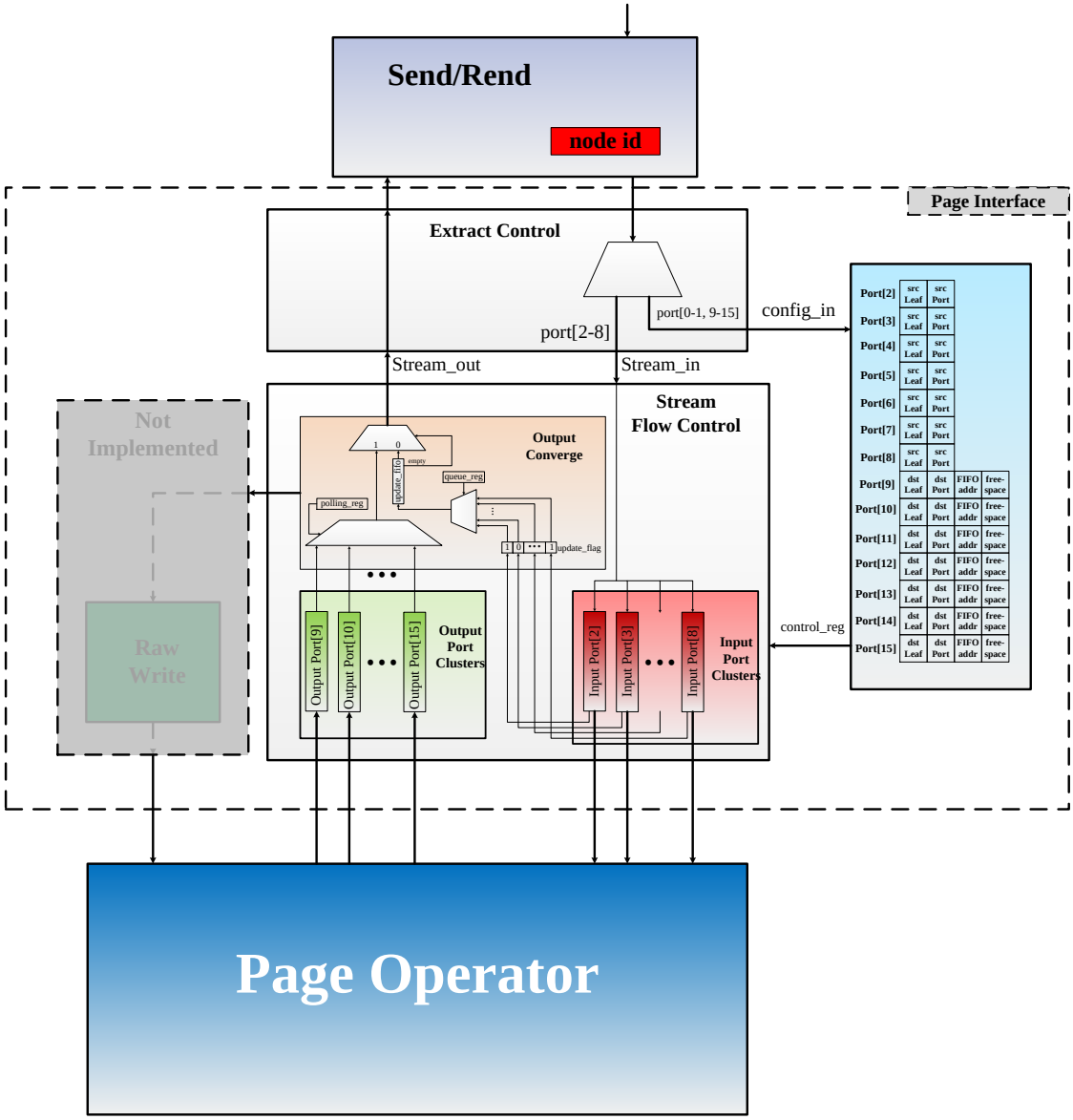


Figure 3.5: Page Block Composition and Interface

3.3.3 Management, Processor, and Memory Interface

In addition to the pages for user logic, we also include host CPU and memory interfaces connected to the BFT. For our previous implementations [123] on the Zynq device, one node is connected to the on-chip ARM processor and another node is connected to the memory interface (DMA engine). Currently, We use X86 CPU as a configuration controller to replace the previous ARM processor. After all the partial bitstreams are loaded onto the PR regions, the X86 host processor sends the configuration packets to each page to configure the designation address for each output of the page. This process acts as the linkage step for our framework.

3.4 Vendor Tool Characterization

Intuitively, we expect the implementation (placement and routing) time to be driven by the number of the logic (LUTs, BRAM, DSPs) to be mapped and the physical layout (pblocks in Vivado) the logic is mapped to. As mapping problems are NP-hard, heuristic algorithms are usually adopted, which makes the mapping more complicated. By characterizing the commercial EDA tool (Vivado) as a black box, we will see how the implementation time is relative to the logic size and layout size. We initially assume that if the pblock size is small (e.g., maybe 1-2 % of the total chip size), the implementation time would be reduced commensurately.

We will see that Vivado is not designed perfectly to deliver the full benefit of separate compilation with PR techniques. In this section, we will characterize the behavior of the commercial EDA tool (Vivado), and use these rules to guide our PRflow mapping strategy. For these experiments, we use Vivado 2022.1 running on a compute server equipped with two 2.7 GHz Intel E5-2680 CPUs and 128 GB of RAM.

3.4.1 Page Size

We first define a PR region and sweep the PR region size from 3840 to 15360 LUTs for ZCU102 Board (XCZU9EG), and 5,000 to 30,000 for Alveo U50 Data-center Card (XCU50) to generate corresponding overlays. Next, we map designs of different sizes (mainly LUTs

Table 3.1: Implementation Time vs. Design and Pblock Size On XCZU9EG (Unit: Seconds)

Design	Size	Pblock Size (LUTs)				
		3840	5760	7680	10080	15360
Shifter Register	970	296	296	298	299	308
	1977	314	309	310	311	320
	2980	318	325	320	323	332
	3980	X	340	333	336	342
	4985	X	348	344	347	355
	5989	X	X	353	356	364
	6992	X	X	364	370	382
	7983	X	X	X	379	395
	8984	X	X	X	391	407
	10005	X	X	X	X	413
MicroBlaze Cores	940	315	315	321	318	322
	1879	333	337	330	340	348
	2818	347	360	353	357	363
	3757	378	375	361	373	387
	4696	X	380	380	390	401
	5635	X	420	397	397	427
	6574	X	X	415	413	440
	7513	X	X	445	437	462

utilization) to one PR region of various sizes. In Tables 3.1 and 3.2, we see the compile time is mainly driven by the size of the logic (LUTs number). When the pblock is large enough compared with the mapped logic, increasing the pblock size does not have a big effect on the compile time. For this experiment, we use two types of logic to map the PR region: one is a shift register of various lengths; the second has a number of cascaded MicroBlazes.

From Figure 3.6, we see when pblock size increases to 15K, the compile time is not close to the other cases. From Figure 3.7, we see when pblock size increases to 30K, the compile time is not close to the other cases. Therefore, for device XCZC9EG, PR regions with a size of around 10 K LUT are good candidates. For device XCU50, PR regions with a size of 20 K LUTs are good candidates.

3.4.2 Partial Reconfiguration Compile

We call physical implementation within a PR region in-context implementation since some of the routing resources in the PR region can be occupied by static logic. Before vision

Table 3.2: Implementation Time vs. Design and Pblock Size On AU50 (Unit: Seconds)

Design	Size	Pblocks Size(LUTs)			
		5568	9744	19952	30616
Shifter Registers	1974	928	918	929	1142
	4974	950	924	978	1193
	5974	X	943	969	1179
	8974	X	997	1012	1243
	9890	X	X	1008	1231
	14911	X	X	1099	1291
	19911	X	X	1205	1348
	20911	X	X	X	1344
	24911	X	X	X	1370
	29911	X	X	X	1448
MicroBlaze Cores	1947	972	957	979	1189
	5703	1111	1058	1107	1290
	6642	X	1086	1090	1294
	7581	X	1124	1112	1329
	8520	X	1148	1147	1381
	9459	X	1201	1150	1391
	10398	X	X	1174	1392
	15093	X	X	1317	1507
	19788	X	X	1451	1569
	20727	X	X	X	1600
	24483	X	X	X	1661
	25422	X	X	X	1680

2020.2, Vivado needs to load the database for the full chip and all the pre-implemented logic in the static region, even though no decisions need to be made on where to place that static logic and routing nets. If we are mapping the logic to one PR region, we should assign minimum dummy logic to the other PR region, as all the dummy logic will be loaded later. Figure 3.8 shows an experiment where we implement all the leaves with one 940 LUT MicroBlaze [128] processor. We move various numbers of leaves to the static region to make different overlays. Finally, we define all the logic pages and the BFT block as PR regions in case 4.

For the device XCZC9EG mapping time, we only map one 940 LUT MicroBlaze processor again to Page 7. As we see in Figure 3.9, the mapping time decreases by moving move the logic from static region to PR regions; the mapping time for the case 4 overlay is the

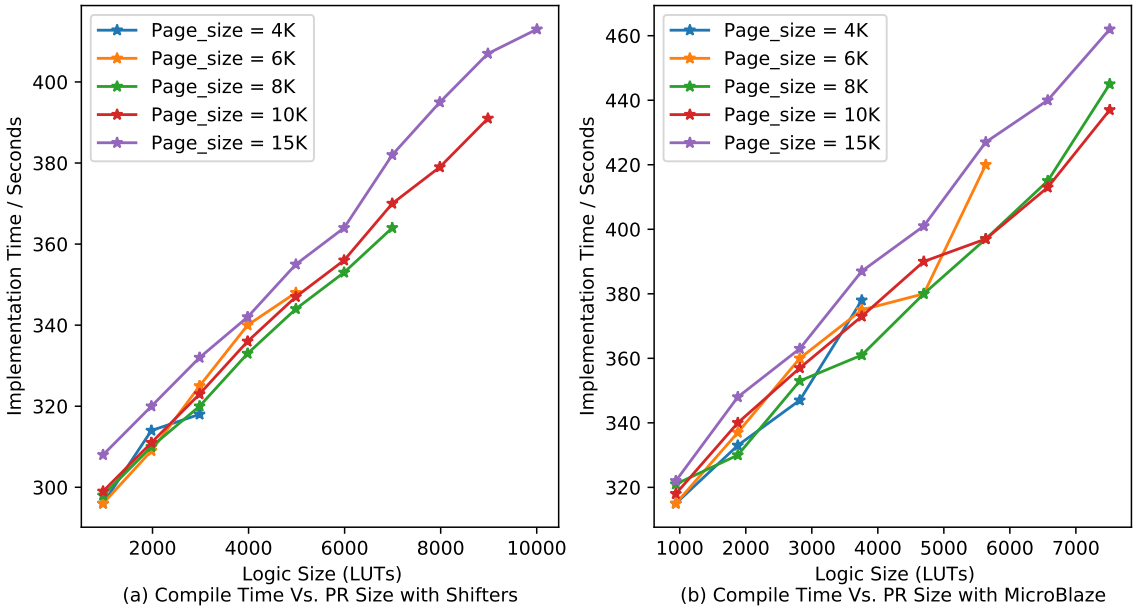


Figure 3.6: Impact of PR Region and Logic to Map (XCZU9EG)

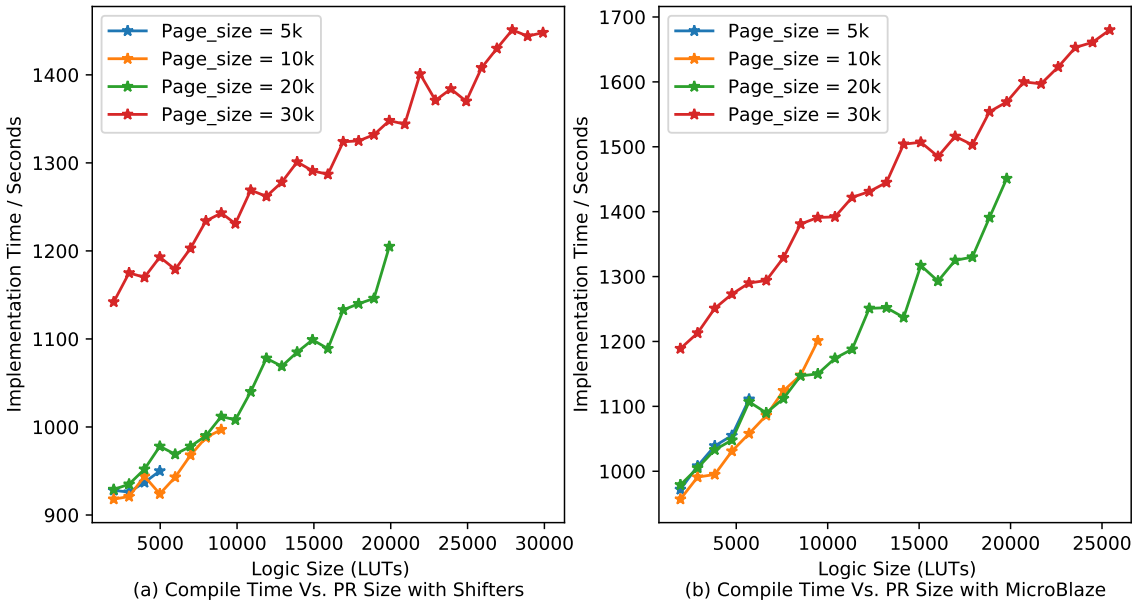


Figure 3.7: Impact of PR Region and Logic to Map (XCU50)

least one.

For the device XCU50 mapping time, we only map one 1,947 LUT MicroBlaze processor again to Page 7. As we see in Figure 3.10, the mapping time decreases by moving the logic

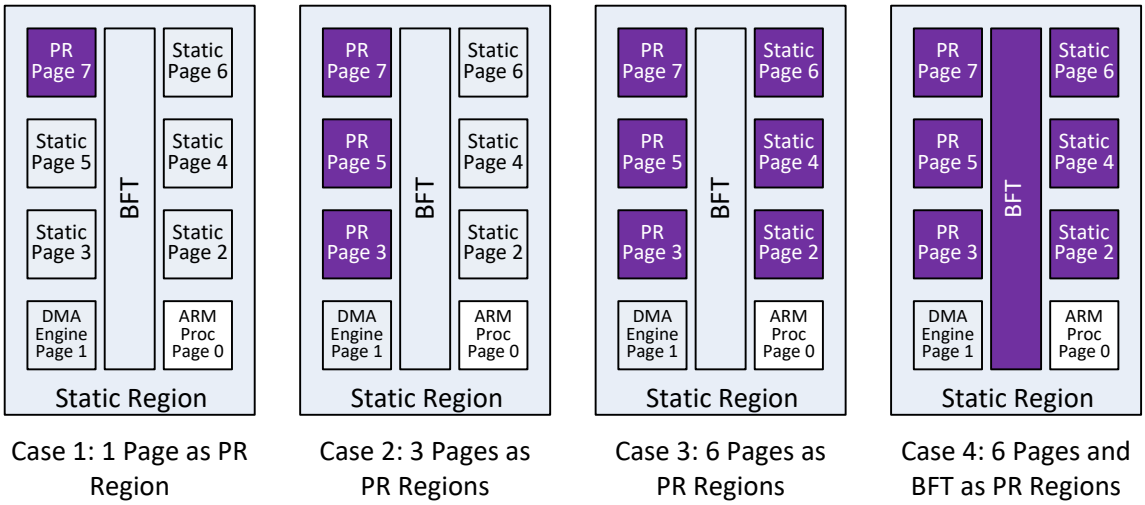


Figure 3.8: Impact of Static Region size on PR Mapping Time

from the static region to PR regions; the mapping time for the case 3 overlay is the least one. We did not implement case 4 for device XCU50, because it is too complicated to manually decompose the PSNoC on this larger device. For the XCU50 device, we see the implementation time increase when we put more pages in the static by using the traditional PR overlay, which seems contradictory to XCZU9EG. We think this is because we can only use a level-1 PR region in XCU50 instead of a clean FPGA device. As Xilinx pre-implements some firmware logic in the device, loading this logic dominates the compile time over the benefit by putting more pages into PR regions. For the abstract shell overlay, we see compilation time is much shorter than the traditional overlay, and decreases slightly from 5 to 6 in Figure 3.10.

In summary, we should put as much logic as possible into the PR region to reduce the logic to map for a single PR region implementation when using the traditional non-abstract_shell technique. Abstract shell technique can reduce the negative impact of static logic.

3.4.3 Abstract Shell Variance

From Chapter 3.4.2, we see the abstract shell technique can strip away the majority of the static logic and accelerate mapping time for a specific PR region. However, compile time is

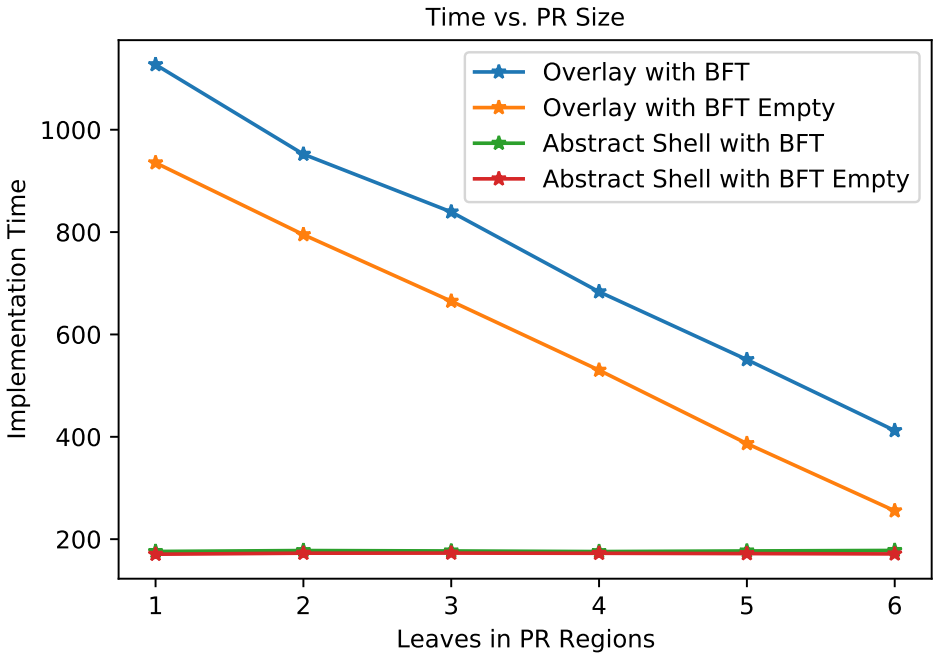


Figure 3.9: Mapping Time with Different Static Size (XCZU9EG)

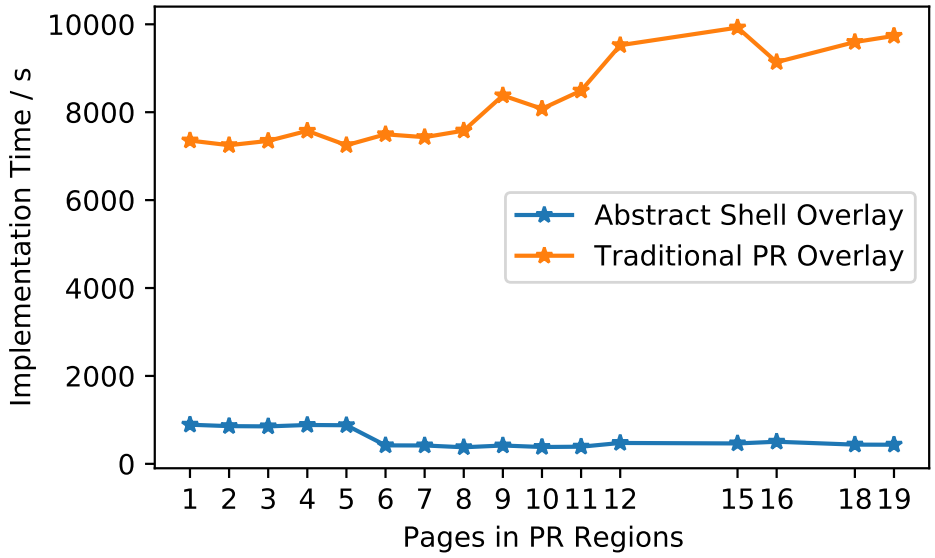


Figure 3.10: Mapping Time with Different Static Size (XCU50)

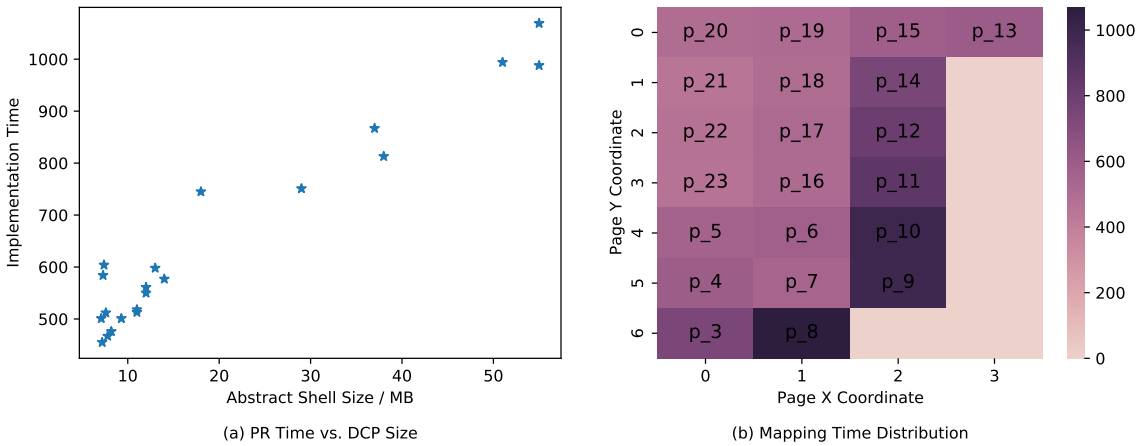


Figure 3.11: Mapping Time on Abstract Shell with Different Sizes

still possible to be affected by the static region, especially for those PR regions, which are close to the heavily-congested static region. For this experiment, we map the same operator coloringFB_top with the size of 3910 LUTs to different PR regions on corresponding abstract shell checkpoints. From Figure 3.11 (a), we see the mapping time is roughly proportional to the abstract shell size. From Figure 3.11 (b), we see the pages close to the static shell include more related logic and wires and have higher mapping time (Pages 8, 9, 10, 11, 12, 14).

Therefore, we should avoid using or defining dummy pages adjacent to the static shell regions when we define the overlay with PR regions.

3.5 Toolflow

Figure 3.12 shows the toolflow for PRflow, which mainly includes host executable generation and compilation coordination around Vitis_HLS, Vivado RTL synthesis, and implementation routines.

3.5.1 Prepare Synthesis

PRflow uses separate OoC (Out-of-Context) synthesis runs to compile each of the operators into a netlist design checkpoint (.dcp) file. Our tool sets parameters and generates the appropriate page interface (Chapter 3.3.2) for the operator based on its need (e.g., number

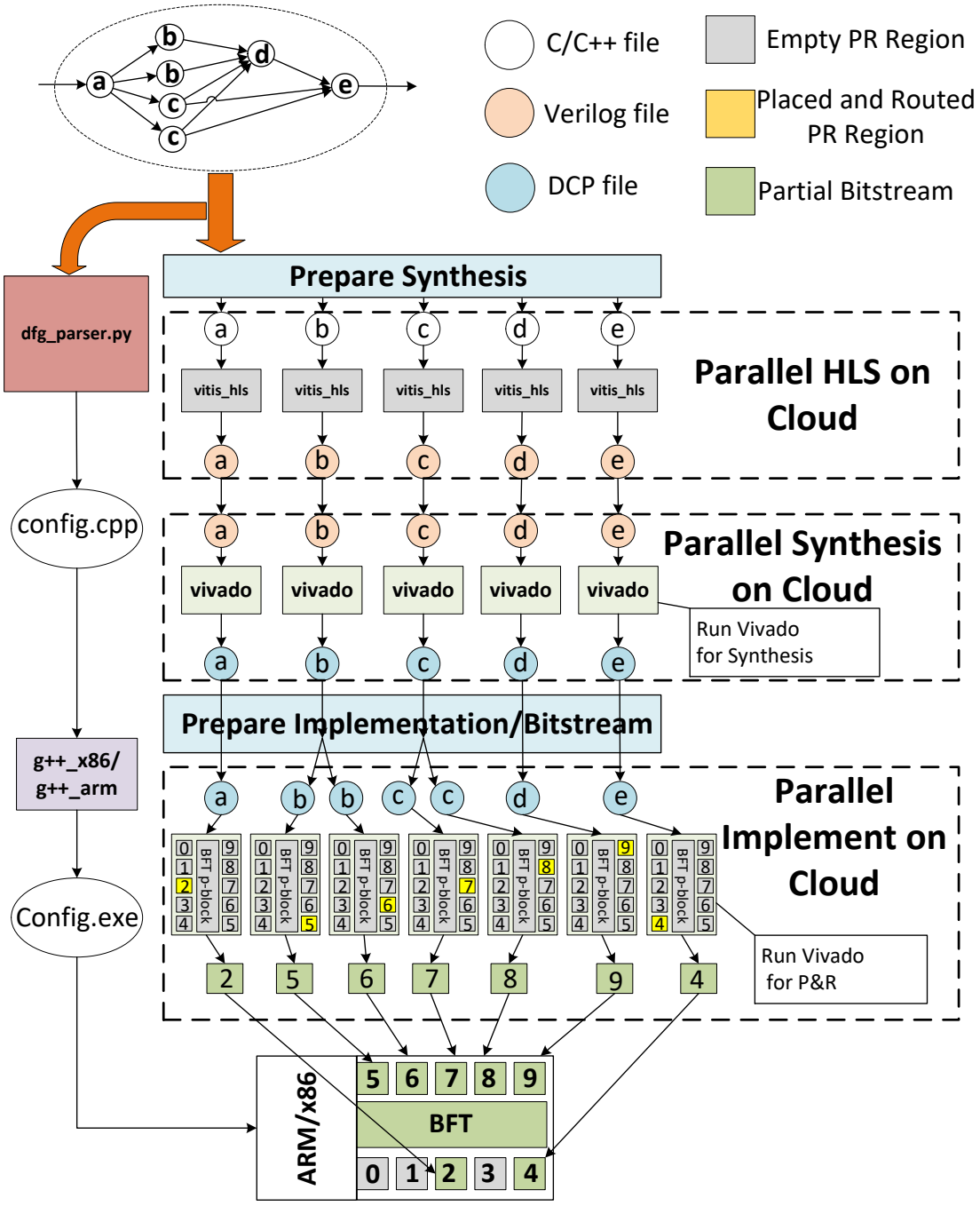


Figure 3.12: PRflow framework

of input and output streams). Overall operation is coordinated with a Python script. The main steps executed in this phase are listed below.

1. Configuring the appropriate page interface;
2. Wrapping the user logic in C or Verilog with the page interface;
3. Creating TCL scripts to control synthesis runs;
4. Spawning the separate synthesis routines to cloud servers;
5. Result is a design checkpoint file.

3.5.2 Prepare Implementation

After the page blocks have been synthesized to netlist design checkpoints, our second phase directs the separate physical implementation of each page block to a particular page pblock.

Based on a page assignment from the pragmas, a page implementation tool packages up the design for implementation, including setting the pblocks that are not being mapped in this separate compilation to dummy designs. It creates the TCL scripts to control the implementation runs and spawns the implementation runs to cloud servers. The result of the synthesis is a partial reconfiguration bitstream for the target page pblock.

3.5.3 Prepare Host Driver

Concurrent with the partial bitstream generations, we also prepare the host code, which will be executed by an ARM processor (embedded platforms) or x86 CPU (data-center platforms). Designs are expressed in C/C++ in units of operators (a, b, c, d, e), shown in Figure 3.12. A top.cpp, which calls all the operator functions, is parsed by a python script (dfg_parser.py) to generate an OpenCL host driver code (host.cpp). The driver code is responsible for sending configuration packets to configure the BFT NoC, setting up DMA engines, launching app kernels, and interpreting the processed results.

For each stream link, the ARM/x86 processor sends control messages over the BFT to configure the source and destination ports in the respective page interfaces, so that they know each other's location and port identifiers to construct packets.

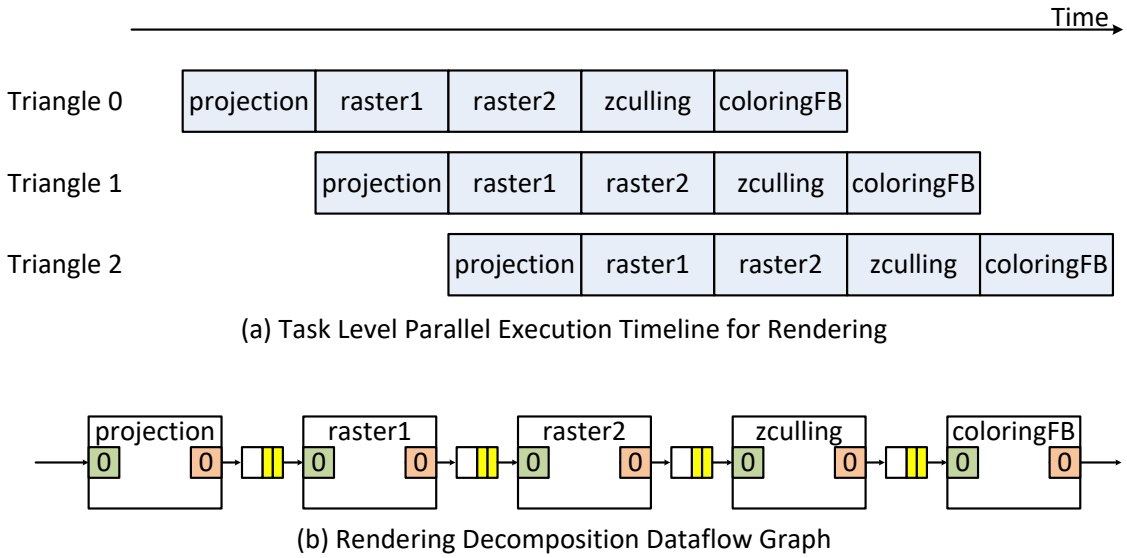


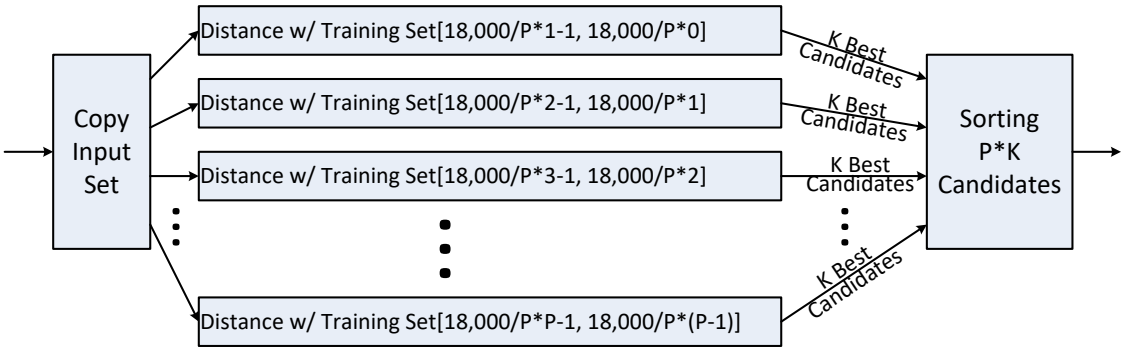
Figure 3.13: Rendering Benchmark Decomposition

3.6 Benchmark Set

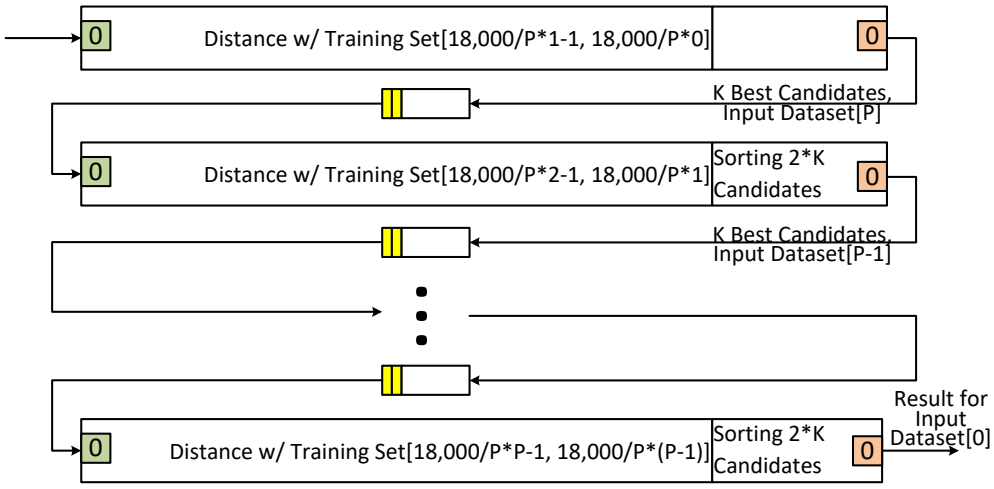
To evaluate our PRflow framework, we use the Rosetta HLS Benchmark suite [146], which contains a diverse range of applications, including machine learning and image processing. The original benchmarks are written for monolithic HLS compilation. We refactor the code in the form of the latency-insensitive style in 3.2: we decompose the application into separate operators and connect them by streaming links. The detail of each benchmark is explained below.

3.6.1 Code Refactor

Rendering – This application renders 256×256 8 bits 2-D images from 3-D triangle coordinates (3192 triangles per image). It mainly includes 5 pipeline kernel stages: projection, raster1, raster2, zculling, and coloringFB. By applying `dataflow` HLS pragma in Vitis_HLS, these five stages can be executed in a Task-Level Parallel (TLP) fashion as shown in Figure 3.13(a). Intuitively, we decompose the rendering benchmark into five operators by only changing the interface between different stages to stream links shown in Figure 3.13(b),



(a) Data Level Parallel Execution Timeline for Hamming Distance Calculation



(b) Decompose Digit Recognition to Systolic Array for Task Level Parallelism

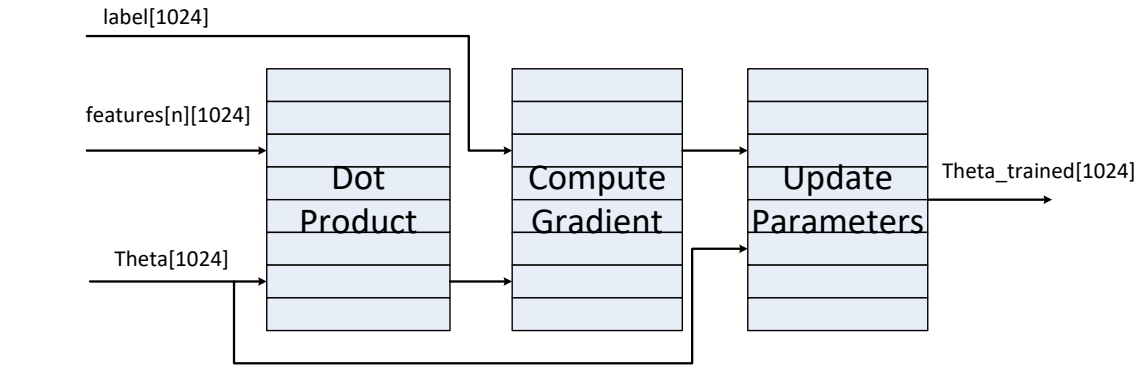
Figure 3.14: Digit Recognition Benchmark Decomposition

where all the operators are straightforwardly running in a TLP fashion.

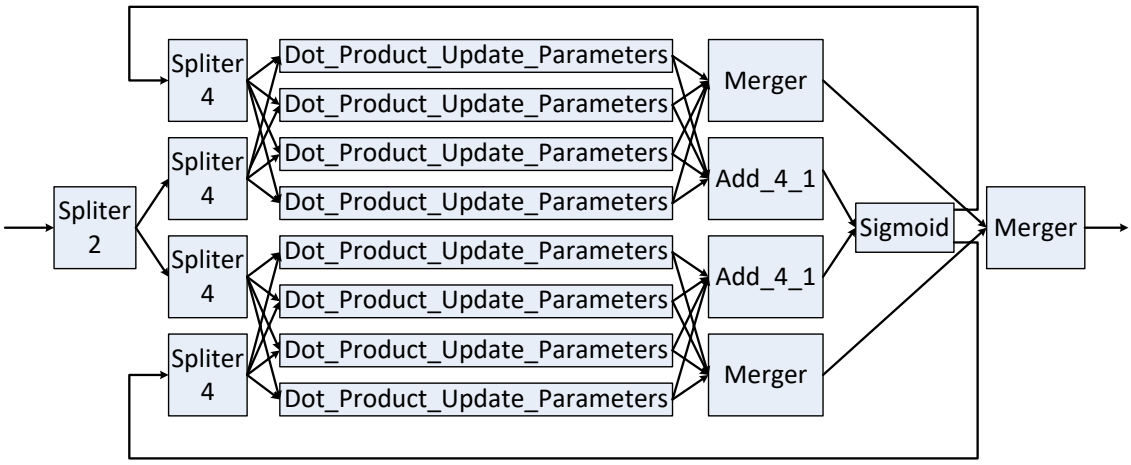
Digit Recognition – This application is based on K-Nearest-Neighbour (KNN) algorithm. A subset of MNIST database [33] was downsampled to 18,000 training and 2000 test samples were stored as 196-bit unsigned integers per image. 18,000 196-bit images are stored on-chip by BRAMs, and for each test image a Hamming distance is calculated for each training sample. K training samples with the smallest Hamming distance are voted to decide the final result. The Hamming distance calculation between the test input and each training

sample dominates the workload. The original can perform XOR of 196 bits by one clock cycle. However, calculating 18,000 times of Hamming distance per test input sequentially is inefficient. To exploit data parallel, the original code partitions the training set by a factor called `PAR_FACTOR` (P) to calculate one input set with P training samples in parallel. Finally, a voting module will sort K*P candidates from P partition results. Increasing P can decrease the distance calculation cycles proportionally but increase the complexity of the final sorting module. For our decomposed version, we still split the training set to P operators. Yet, we connect the operators in a systolic array fashion. For the first operator, it only outputs the best K candidates and transfers the input set to its consumer. For the second operator, it still calculates the best K candidates with respect to its local training samples. But a small sorting operation will be performed over 2*K candidates (extra K candidates from its producer operator). For the final operator, its sorting module still works on 2*K candidates and outputs the best result. Different operators are running with task-level parallelism and the sorting module does not scale with partition factor P.

Spam Filter – This benchmark is a Logic Regression (LR) training model based on Stochastic Gradient Descent (SGD). The dataset contains 5,000 emails, 4,500 for training and 500 for testing. Each email is represented by a 1024-element vector of 16-bit fixed-point values. The training goal is to adjust 1024 32-bit fixed-point parameters by 5 epochs of 4,500 training samples. As the dot_product operators are suitable for data parallelism, the original code partition the features and parameters by a `PAR_FACTOR` (P). As shown in Figure 3.15(a), if we partition the features and parameters into 8 groups, 8 multiply operations can be executed concurrently. For the decomposed version, we use to split the dotProduct and paraUpdate modules into 8 groups. We store the parameters in 8 operators and split the input features into these 8 operators to perform dotProduct. The intermediate results will be added together to compute the gradient. Then the gradients are sent back to the 8 operators to update the local parameters. Therefore, there is a feedback loop from Sigmoid to Dot_Product_Update_Parameters operators, which would affect the performance when the latency is high.

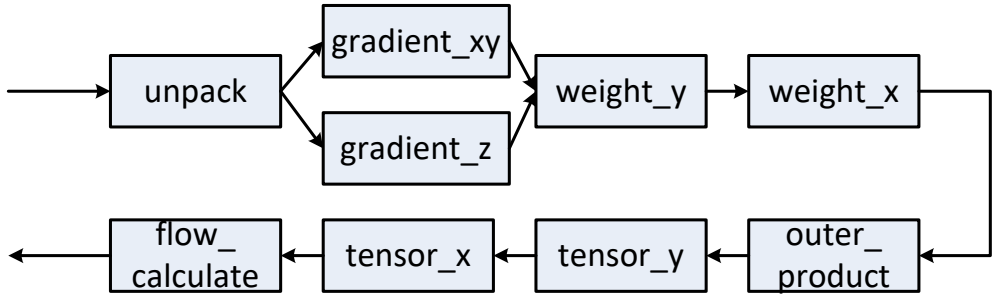


(a) Spam Filter Dataflow Graph

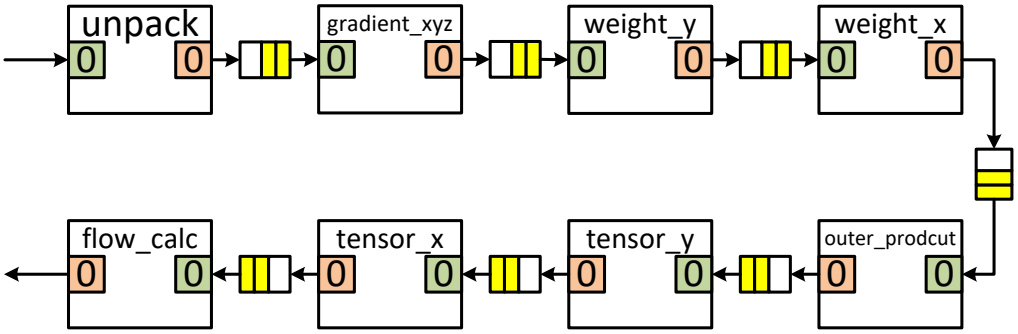


(b) Spam Filter Decomposition Dataflow Graph

Figure 3.15: Spam Filter Benchmark



(a) Hardware Diagram for Optical Flow



(b) Decomposition for Optical Flow

Figure 3.16: Optical Flow Benchmark

Optical Flow – This benchmark compute graph is similar to Rendering as both can benefit from Task-Level Parallelism (TPL) and it is straightforward to decompose in Figure 3.16(a). We change the interface from array to stream links. As for PRflow, we have input buffer and output buffer between pages and NoC, we do not need to explicitly add buffer between different operators, shown in Figure 3.16(b). As gradient_xy and gradient_z have similar operators, we merge these to into gradient_xyz operator.

Face Detect – This benchmark is based on the Viola-Jones algorithm to detect faces from a given 320×240 8-bit grey-scale image. It mainly includes 2 parts: an image scaling function and cascaded classifiers. The image scaling function generates scaled images from an image pyramid. Then a series of scaled images are fed into a cascaded classifier. A line buffer containing 25 rows will be randomly accessed by different classifiers. Classifiers operate on a sliding window of a 25×25 integral image. Theoretically, all the classifiers can be executed in parallel. But the limited on-chip memory and routing resource only allow a limited number of classifiers to run in parallel. In this example, the author parallelizes the first 3 stages of the classifiers (strong filters) and pipelines the rest 23 stages (weak filters). For our decomposed version, it is important to keep the window images (e.g., Integral Images) local to the classifiers. Based on this, we partition the line buffers and window images into 5 partitions in 5 operators. Accordingly, we split the strong filters into 5 operators according to the data access. We copy the line buffers for the weak filters, and the line buffers into 5 partitions in 5 operators. Similarly, we split the weak classifiers into the corresponding operators. As we finally need to sum up the results from the 5 partitions for both strong filters and weak filters, a communications bottleneck may slow down the overall throughput.

BNN – This benchmark is a Binarized Neural Network (BNN) that originally fits small FPGAs. It is based on the open-source implementation [145], which operates on CIFAR-10 dataset [67]. It contains 17 convolutional layers and 37 fully-connected layers. The main workload of this benchmark is intensive of bit-wise logic operations. As the application is compute-bound, the parameters are accessed from off-chip memory. The convolutional layer and fully-connected layers are hardware parameterized, which means only 2 hardware

kernels are implemented. These 2 kernels can run in a different mode according to the input layer type. This is area efficient but can limit the performance. For our decomposed version, we duplicate the convolutional layers according to the on-chip resource. We copy the convolutional kernel 3 times: the first one maps the first 3 layers; the second one maps the following 7 layers; the third one maps the last 6 layers. For the full-connected layers, the execution cycles only occupy a small amount, which means there is no need to duplicate and pipeline this stage. We make all the layers run in a task-level parallelism fashion, this means different stages need to access the parameters concurrently and asynchronously. Therefore, we move the parameters onto the chip as our target is AU50, which contains a significant amount of on-chip memory to hold all the parameters. To improve the performance, we can duplicate more stages of convolutional layers at the cost of more area.

3.6.2 Lines-of-Code

Table 3.3 summarizes the lines of code (LoC) comparisons between our baseline [146] and refactored code for PRflow. We do not account for the duplicated pages, such as `docProduct_2--7` for *spam_filter* and `update_knn2-10` for *digit_reg*. For some benchmarks, the coding style is quite different from the latency-insensitive form, so we put more effort into refactoring the code, such as *rendering* and *spam_filter*. Correspondingly, the LoC is increased by higher factors. Nevertheless, some benchmarks are easy to refactor, as we only need to change the interfaces for the sub-functions, such as *optical_flow*.

Table 3.3: Interface Resource Consumption

Benchmark	Baseline LoC [146]	PRflow LoC	LoC Increase
digit_reg	141	204	1.4×
optical_flow	403	485	1.2×
rendering	245	761	3.1×
spam_filter	144	304	2.1×
face_detect	2945	4188	1.4×
bnn	611	1127	1.8×

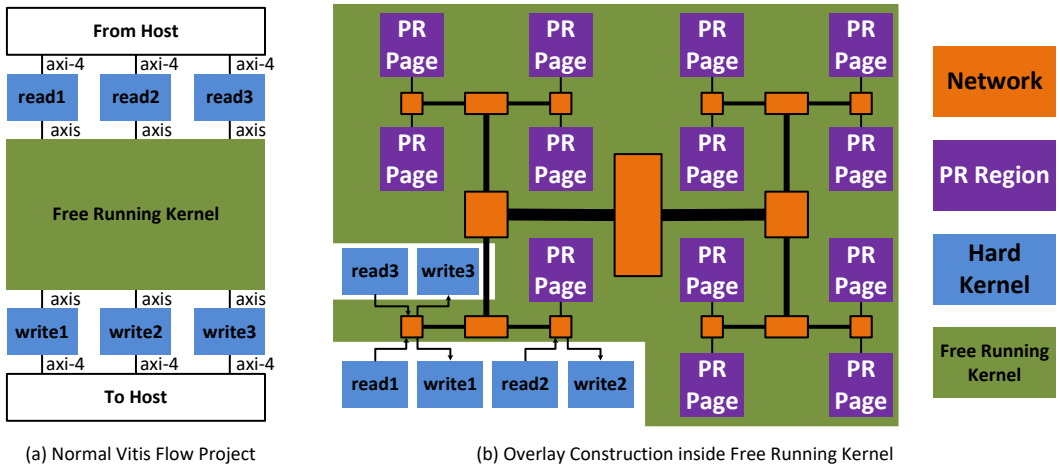


Figure 3.17: Overlay Implementation

3.7 Implementations and Experiments

To evaluate the impact of PRflow framework, we use the Rosetta High-Level Synthesis C++ Benchmarks for FPGAs [146]. We map the designs to the Alveo U50 Data Center card [135] 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 (705,520 LUTs, 2,232 18Kb BRAMs, and 4,920 DSPs). PRflow uses Xilinx Vitis 2022.1 including associated `Vivado` and `Vitis_HLS` and `XRT` as the backend. We perform the mapping on a cluster of 8 servers, each with 2.7 GH Intel E5-2680 CPUs and 128 GB of RAMs.

3.7.1 Overlay Implementation

To leverage the existing OpenCL driver from Xilinx Runtime [140], we first use Vitis to compile a project with 7 kernels defined, shown in Figure 3.17(a). One of them is called *free-running kernel*, which means that as long as the FPGA fabric is configured, it is running immediately. The other 6 kernels are *read1–3* and *write1–3*. For normal Vitis flow, it first calls HLS to compile the C++ sources to Verilog files. Next, Vitis calls Vivado to synthesize the Verilog files to one design checkpoint (DCP) file.

In Figure 3.18, we replace the Free Running Kernel with our overlay (separate process unit + network) at Verilog level, which we call free_running_new. The equivalent system

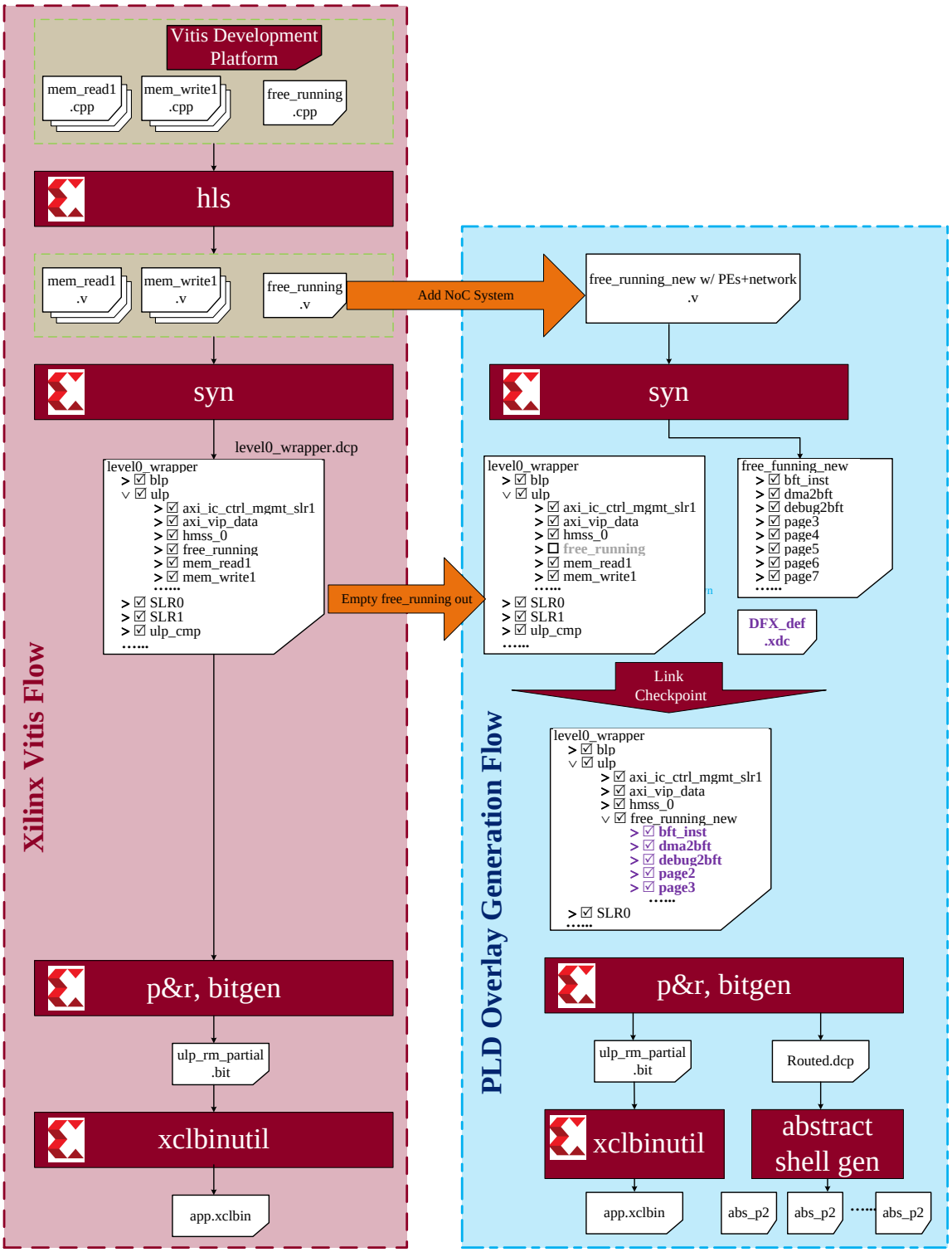


Figure 3.18: Overlay Generation flow

diagram is shown in Figure 3.17(b). Finally, the overlay diagram can be seen in Figure 3.19. Correspondingly, our host driver only needs to launch the 6 memory kernels (read1–3, write1–3) to run an application, shown in Algorithm 1.

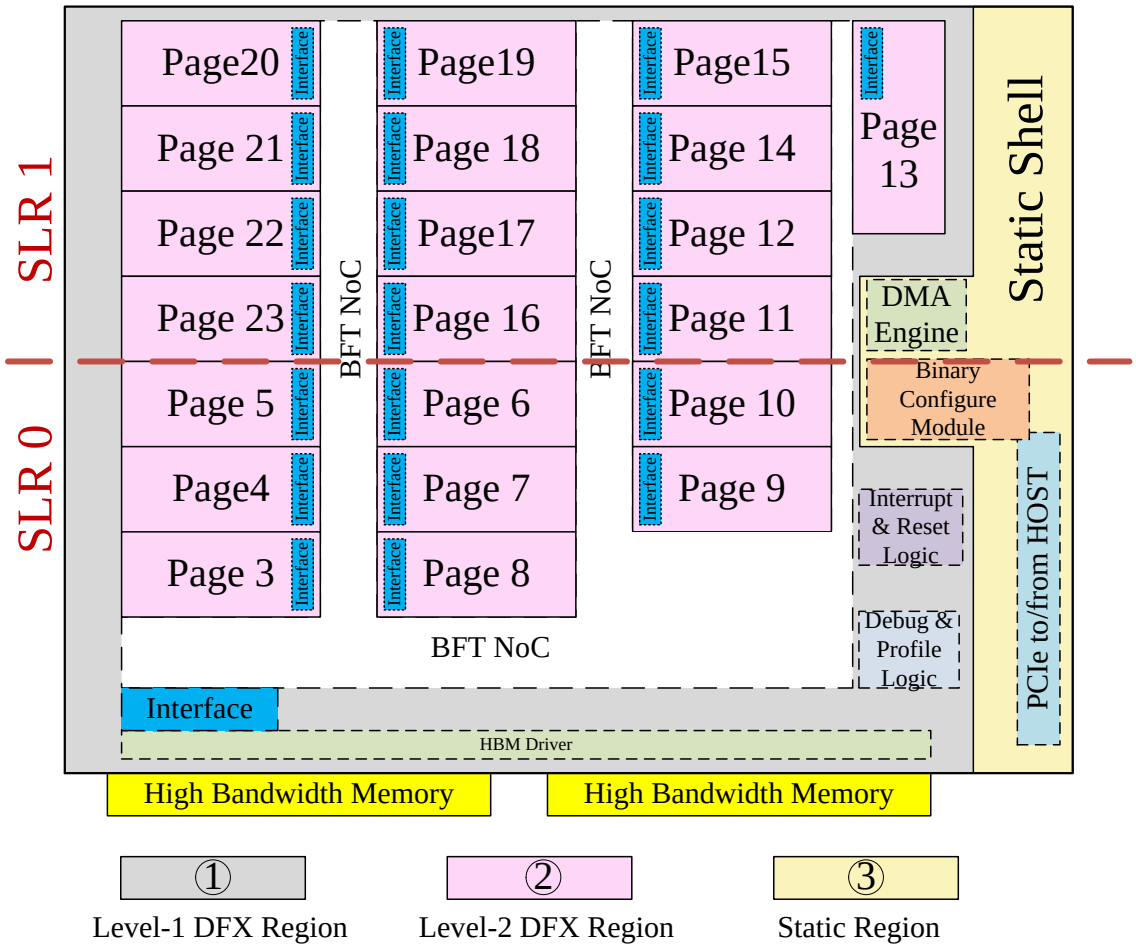


Figure 3.19: FPGA Decomposition – Pages, BFT NoC, and Support Infrastructure

3.7.2 Design Points

For AU50, we use a 32-page BFT NoC with Rent parameter p=0.5 and a datapath width of 48 supporting 32 payloads. The Network and the PR pages are all running at 200 MHz. As one input and output are used for communication between a page and the NoC, the peak bandwidth is 0.8 GB/s in each direction. One endpoint is connected to Pseudo Channel 1 (PC1) or configuring the NoC. As the number of configuration packets is at most 224

Algorithm 1 Driver Code for the Host

```
1: procedure INT MAIN(int argc, char** argv)
2:     Check Device and initiate Context
3:     Configure the FPGA with Level-1 xclbin
4:     for all xclbins do
5:         Configure the FPGA with a Level-2 partial xclbin
6:     end for
7:
8:     Create an out-of-order queue
9:     Initiate host memory and device memory
10:    Launch Kernel read1 to configure BFT NoC
11:    Launch Kernel write1 to read back info from BFT NoC
12:    Wait for all kernels to be finished
13:
14:    Launch Kernel read2 to configure BFT NoC
15:    Launch Kernel write2 to read back info from BFT NoC
16:
17:    while Not last debug Chunk do
18:        Launch Kernel read3 to request debug info
19:        Launch Kernel write3 to receive debug info from FPGA
20:        Wait for write3 to be finished
21:        Print out debug info
22:    end while
23:
24:    Wait for all kernels to be finished
25:    Is_Successful ← Check_results()
26:    return Is_Successful
27: end procedure
```

(7*2*32/2), the access range of PC1 is only 256 MB, which is the smallest range. One end node of the NoC is connected to Pseudo Channel 2 (PC2), which can access the HBM memory of 8 GB. One more endpoint is connected to Pseudo Channel 3 (PC3), which can be used for delivering debug information later.

The page interface resource requirements scale with input ports, I, and output ports, O.

$$\text{Page Interface LUTs} \approx 206 + 66I + 227O \tag{3.1}$$

$$\text{Page Int. 36Kb BRAMs} = 1 + 2I + O/2 \tag{3.2}$$

We divide the FPGAs into 21 separate PR pages as shown in Figure 3.20. Each page is connected to one endpoint of the BFT NoC. Due to the heterogeneous resource distribution of modern FPGA, all the pages have similar but not exactly the same resource. There are 4 types of pages, and the resource detail is listed in Table 3.4. The total page-blocks occupy 62 % of the LUTs, 85 % of the BRAMs, and 65 % of DSPs in customer logic (CL). PRflow uses Hoplite BFT [62, 61, 36] for the packet-switched network, running at 200 MHz with 32 data payloads. The page interface in each PR region allows the users to address by specifying the header of each packet sent by the user logic.

Table 3.4: Resource Distribution

Page Type	**Type-1**	**Type-2**	**Type-3**	**Type-4**
LUTs	21,240	17,464	18,880	24,920
FFs	43,200	35,520	38,400	49,840
BRAM18s	120	72	72	66
DSPs	168	120	144	176
Number	7	7	6	1

3.7.3 Compile Time

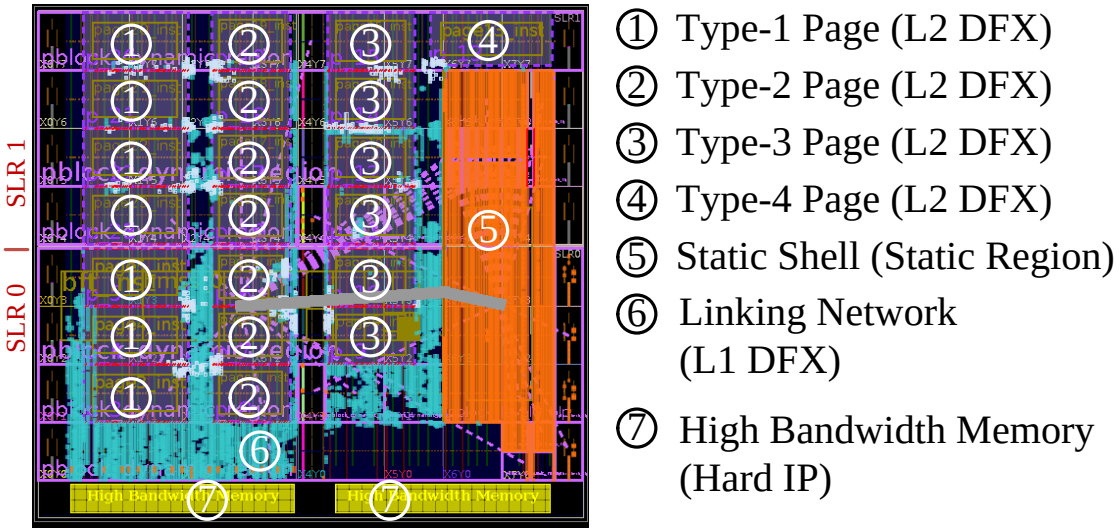
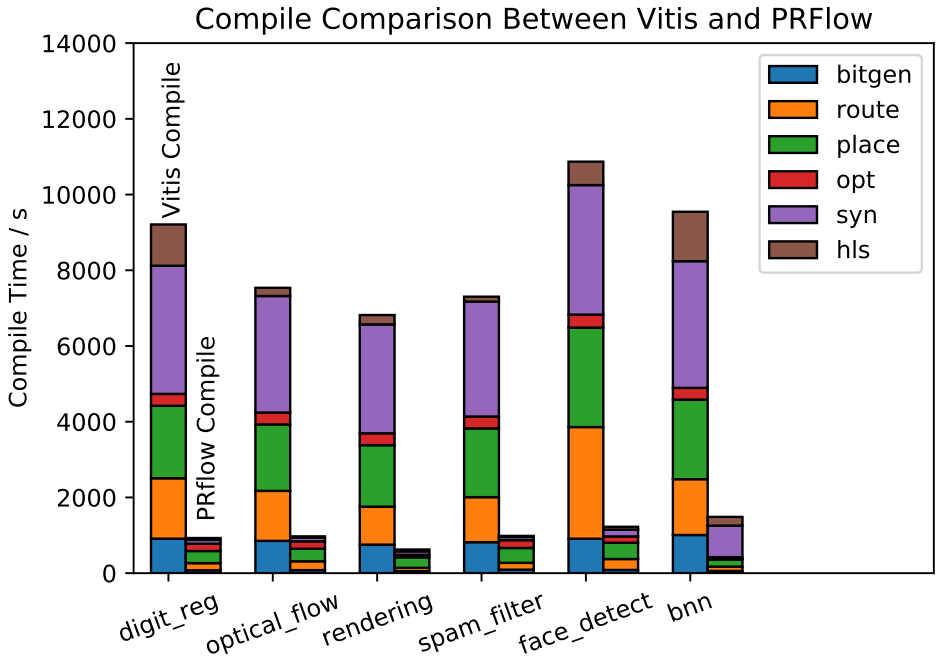


Figure 3.20: Physical Layout Floorplan



`Compile with 200MHz clock constraints for the kernel`

Figure 3.21: PR Recompile Time

Table 3.5: Rosetta Benchmark Time for all Pages(in seconds)

Benchmark	**operator**	**page**	**hls**	**syn**	**p&r**	**bit**	**total**	**speedup**
Benchmark	operator	page	hls	syn	p&r	bit	total	speedup
Continued on next page								

Table 3.5 – continued from previous page

Benchmark	operator	page	hls	syn	p&r	bit	total	speedup
digit_reg	update_knn1	13	51	123	476	55	659	13.98
	update_knn10	12	51	132	655	80	843	10.93
	update_knn2	4	51	100	428	55	584	15.77
	update_knn3	5	51	99	414	55	569	16.19
	update_knn4	6	48	100	397	56	551	16.72
	update_knn5	7	48	100	371	51	521	17.68
	update_knn6	14	49	100	604	68	758	12.15
	update_knn7	9	49	100	827	76	931	9.89
	update_knn8	10	50	104	752	78	897	10.27
	update_knn9	11	51	102	742	71	894	10.3
optical_flow	flow_calc	3	37	146	641	60	820	9.19
	gradient_weight_x	4	35	92	458	54	586	12.86
	gradient_weight_y	5	35	96	457	55	591	12.75
	gradient_xyz_calc	6	34	78	380	51	492	15.32
	merge	8	27	75	845	94	914	8.25
	outer_product	7	32	85	352	52	471	16
	tensor_weight_x	9	38	93	888	80	974	7.74
	tensor_weight_y	10	41	93	754	79	877	8.59
	unpack	11	39	86	703	68	826	9.12
rendering	coloringFB	20	47	103	308	50	462	14.76
	projection	21	20	76	284	48	385	17.71
	rasterization1	22	22	93	285	47	403	16.92
	rasterization2	23	22	82	290	50	399	17.09
	read_in	19	20	81	283	45	383	17.8
	zculling	5	49	92	484	52	627	10.87
spam_filter	Sigmoid_axi	3	33	78	548	55	659	11.08
	data_1_4_1	4	26	134	472	57	637	11.46
	data_1_4_2	5	27	132	437	55	599	12.19
	data_1_4_3	6	27	124	393	53	546	13.38
	data_1_4_4	7	27	124	376	54	531	13.75
	data_2_1	8	28	85	909	93	987	7.4
	data_in_redir	9	31	98	856	80	946	7.72
	dotProduct_1	10	31	93	752	77	866	8.43
	dotProduct_2	11	36	92	750	70	876	8.34
	dotProduct_3	12	38	90	686	87	828	8.82
	dotProduct_4	13	35	90	446	52	577	12.66
	dotProduct_5	14	39	94	645	69	781	9.35

Continued on next page

Table 3.5 – continued from previous page

Benchmark	operator	page	hls	syn	p&r	bit	total	speedup
	dotProduct_6	15	36	92	463	45	590	12.38
	dotProduct_7	16	36	92	362	51	494	14.78
	dotProduct_8	17	39	93	357	50	492	14.84
face_detect	imageScaler_bot	9	18	83	835	85	899	12.09
	imageScaler_top	4	19	87	419	57	532	20.43
	sfilter0	5	78	335	548	60	964	11.27
	sfilter1	6	83	151	632	60	869	12.51
	sfilter2	7	87	156	559	60	806	13.49
	sfilter3	3	77	181	793	66	1055	10.3
	sfilter4	10	76	178	986	85	1228	8.85
	strong_classifier	11	38	168	949	74	1149	9.46
	weak_data	23	19	79	298	53	402	27.04
	weak_process_new	12	31	244	830	85	1107	9.82
	wfilter0	13	68	185	607	57	867	12.54
	wfilter0_process	14	62	130	690	71	883	12.31
	wfilter1	15	68	167	616	54	854	12.73
	wfilter1_process	16	64	130	426	59	624	17.42
	wfilter2	17	68	171	593	59	838	12.97
	wfilter2_process	18	64	132	408	53	605	17.97
	wfilter3	19	68	174	590	53	833	13.05
	wfilter3_process	20	64	132	382	59	587	18.52
	wfilter4	21	68	211	602	62	891	12.2
	wfilter4_process	22	63	133	401	58	605	17.97
bnn	bc0_gen_0	15	26	71	336	45	433	22.05
	bc1_gen_0	14	25	80	543	70	656	14.55
	bc1_gen_1	11	26	81	671	69	775	12.32
	bc2_gen_0	19	28	82	284	48	397	24.05
	bc2_gen_1	9	27	85	841	81	913	10.46
	bd_gen_0	6	30	73	290	50	398	23.98
	bd_gen_1	7	29	83	294	52	413	23.11
	bd_gen_2	22	28	83	298	50	414	23.06
	bd_gen_3	23	28	85	292	54	413	23.11
	bd_gen_4	4	29	84	391	54	508	18.79
	bd_gen_5	3	28	81	521	58	631	15.13
	bd_gen_6	18	26	86	303	47	416	22.95
	bd_gen_7	17	26	85	318	49	432	22.1
	bd_gen_8	16	30	85	314	50	431	22.15

Continued on next page

Table 3.5 – continued from previous page

Benchmark	operator	page	hls	syn	p&r	bit	total	speedup
	bd_gen_9	5	27	83	361	50	472	20.22
	bin_conv_0	20	231	832	419	58	1488	6.42
	bin_conv_1	21	204	714	387	55	1310	7.29
	bin_conv_2	7	191	264	443	54	900	10.61
	bin_dense	10	51	102	768	80	915	10.43
	data_transfer	12	26	84	685	80	803	11.89
	fp_conv	13	44	101	452	53	603	15.83

We initially compile the Rosetta benchmarks by only changing the interface to fit the AU50's high bandwidth with normal Vitis flow as our baseline. Then we compile the modified source code with our PRflow framework and we summarize the results in Table 3.6 and Figure 3.21. The detail for all the pages of the benchmark set is tabulated in Table 3.5. For the original Vitis flow, we set the Vitis compile option `vivado.synth.jobs` and `vivado.impl.jobs` to 32, equal to the threads number of one server to make the best use of the server's threads. We see the compile time takes 2–3 hours for all the benchmarks. Noting that it still takes more than 2 hours even for small benchmarks on large FPGAs (e.g., rendering and spam filter), as Vitis needs to load the database for the full chip, which is not necessary to map small applications.

For our PRflow, we see the compile time reduces to 11–24 minutes, a factor of 6.4–10.9 $\times$ speedup. The compile time varies over different pages. Figure 3.22(a) shows the distribution of the mapping time over all the pages for different benchmarks. Applications with the worst mapping time of 24 minutes also have short mapping time for other pages (e.g., 8 minutes). Therefore, the real incremental compile time is determined by which pages the designers are tuning up. Designers can also deliberately separate the logic they are interested in into a small operator, therefore accelerating the edit-error-debug turn. From Figure 3.22(a), we see the median values for all the benchmarks are around 12 minutes. This means it is highly possible that the users only need 12 minutes for every incremental compilation.

In Table 3.6, we can see the compile time speedups vary across different designs, and we see better speedups for small benchmarks (e.g., rendering and digit recognition). This

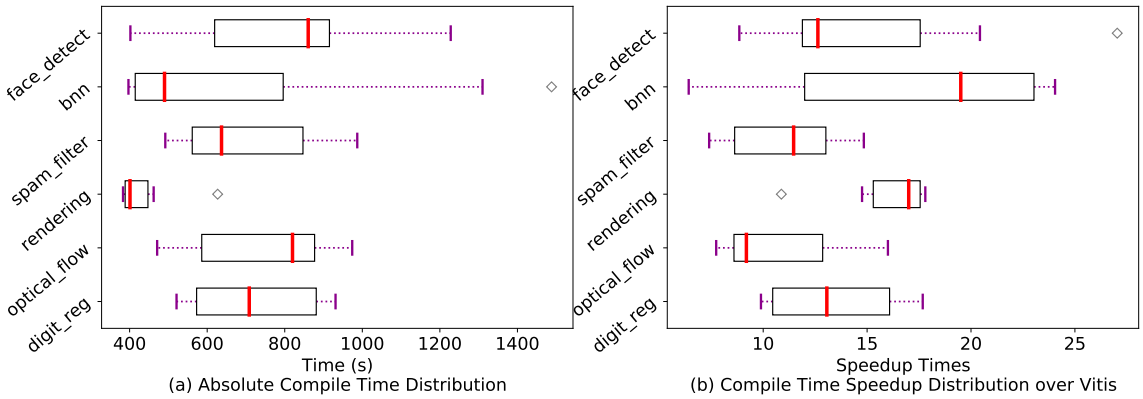


Figure 3.22: Operator Mapping Time for PRflow

is because the long fixed overhead to map large chips can dominate the compile time for small benchmarks, while our PRflow overlay does not have such overhead. Meanwhile, we expect to see similar compile times for both small and large designs. However, we see large designs also take a longer time in PRflow. This is due to the uneven decomposition of the benchmark.

Table 3.6: Rosetta Benchmark Compile Time (in seconds)

	Vitis Flow with 32 Threads					PRflow with 8 Threads for Each Operator					
	hls	syn	p&r	bit	total	hls	syn	p&r	bit	total	Speedup
digit_reg	1091	3385	3823	911	**9210**	49	100	827	76	**931**	9.9
optical_flow	216	3079	3389	853	**7537**	38	93	888	80	**974**	7.7
rendering	248	2875	2941	754	**6818**	49	92	484	52	**627**	10.9
spam_filter	130	3036	3319	818	**7303**	28	85	909	93	**987**	7.4
face_detect	618	3422	5920	909	**10869**	76	178	986	85	**1228**	8.9
bnn	1306	3346	3886	1008	**9546**	231	832	419	58	**1488**	6.4

3.7.4 Performance

Table 3.7 lists the performance between normal Vitis flow and PRflow. By separating compile with PRflow, applications can run 1.1–11 × slower than the normal Vitis flow. Most slowdowns come from the limited bandwidth between pages and the BFT NoC. This is partially due to our general overlay being designed to map a wide range of applications

tuned for quick compilation and verification over high performance. For example, for the optical flow benchmark, each operator can be compiled by the HLS tool with II equal to 1. This means any communication bandwidth loss can cause performance degradation. Out BFT NoC can only provide 0.8 GB/s (32-bits@200MHz), while it needs 7.2 GB/s (288-bits@200MHz) between outer_product and tensor_weight_y operators. This theoretically slows down the performance by a factor of 9 (11 × slowdown by measurement).

Table 3.7: Rosetta Benchmark Performance

	Vitis Flow		PRflow		x86 g++	Vitis Emu
	Freq (MHz)	Runtime per input	Freq (MHz)	Runtime per input	Runtime per input	Runtime per input
digit_reg	200	2.9 us	200	3.3 us	81.3 ms	90.6 s
optical_flow	200	2.4 ms	200	26.6 ms	2.8 s	70.0 s
rendering	200	1.7 ms	200	2.6 ms	16.1 ms	671.0 ms
spam_filter	200	16.9 ms	200	72.3 ms	354.8 ms	12.3 s
face_detect	200	19.1 ms	200	125.0 ms	6.9 s	57.8 s
bnn	150	5.1 ms	200	7.1 ms	27.8 s	2074.0 s

We also compare our performance with Vitis Emulation and X86 simulation. We can see the pure X86 simulation is much faster than Vitis Emulation. This might be due to Vitis Emulation's needing to mimic the OpenCL driver, which adds another layer on top of the C++ compiles, while X86 can optimize the C++ program with the -O3 option for fast execution. PRflow achieves substantial speedup over the X86 simulation or Vitis Emulation. This provides quick running information for the hardware execution with around 10 minutes compilation.

Figure 3.23 shows how the performance and compile-time compare among different platforms and options. This shows PRflow provides more trade space between pure x86 software implementation and raw FPGA implementation. The developers have new control options with fast edit-compile-debug turns with single source code.

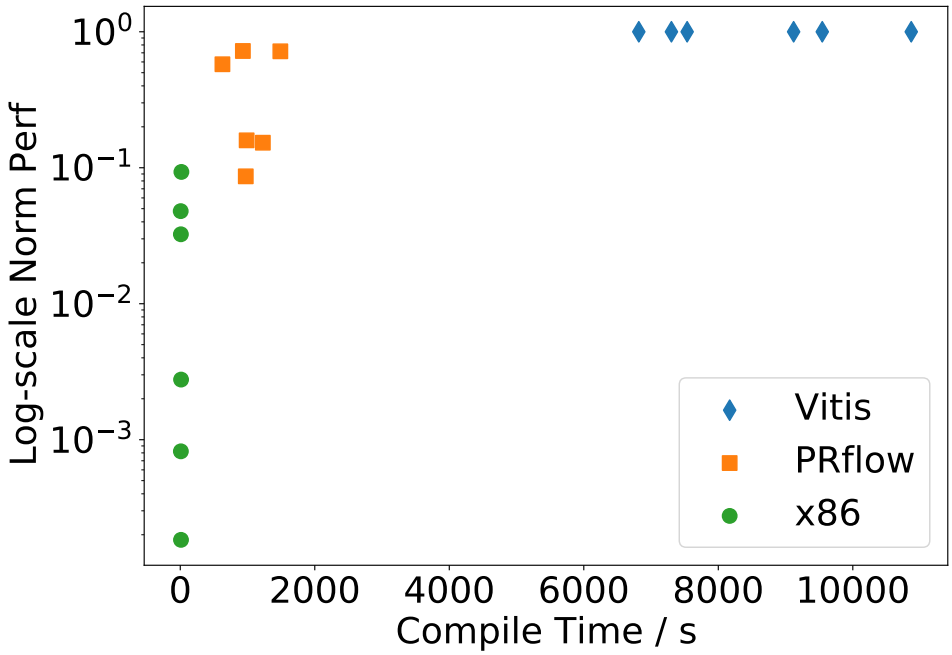


Figure 3.23: Performance vs. Compile Time

3.7.5 Area Evaluation

The area comparison breakdown between Vitis flow and PRflow is listed in Table 3.8. For PRflow, we add up all the operators' resource utilization, including the page interface and the overlay size (around 40K LUTs). We see the resource utilization of Vitis flow has a higher LUT utilization than Vitis flow since PRflow needs FIFOs at the input and output ports. This can consume more BRAMs to implement the FIFOs and more LUTs to construct the supporting logic. One promising solution is to decrease the FIFO depth until Vivado uses distributed LUTs to implement these BRAMs. We can also use Relay Station [17, 19] or Skid Buffer [98] to connect those operators instead of FIFOs. However, users need to be careful when using streaming links with small depth because this might introduce deadlock in dataflow graph applications.

3.8 Discussion

From the preliminary results in Chapter 3.7, we see the divide-and-conquer strategy can effectively reduce the compile times by factors of 6.4–10.9×. However, several limitations still

Table 3.8: Rosetta Benchmark Area Consumption

Benchmark	Vitis Flow			PRflow			
	LUT	B18	DSP	LUT	B18	DSP	PAGE#
digit_reg	56293	368	1	82971	413	1	10
optical_flow	24202	192	286	68455	199	286	9
rendering	12293	112	13	50909	153	9	6
spam_filter	15262	40	224	90795	257	256	15
face_detect	53672	194	97	213179	349	145	20
bnn	51626	1132	7	104675	1249	4	22

exist, such as manual intervention in source code, fixed page size, and low area utilization.

3.8.1 C++ Level Automation

To leverage the quick compilation by PRflow, users need to re-write the code into separate operators connected by latency-insensitive links. This might leave some burden on the users when a large-scale software project has been finely tuned by enough tuning. HeteroRefactor [72] is an excellent framework that can analyze the C/C++ code and monitors FPGA-specific dynamic invariant to generate C/C++ code efficiently for HLS and RTL implementations. Using *pragmas* or *directives* has been proven to be a potential solution [27, 114]. Instead of using hours to transfer a giant loop to a systolic array, our target is to guide the HLS tool to quickly decompose more generic applications into separate operators for later parallel compilations.

3.8.2 Bandwidth Bottleneck

From table 3.7, we see the performance is degraded by various factors. This is due to the fixed limited bandwidth between pages and the NoC (0.8 GB/s). If some benchmarks need high inter-page bandwidth (optical flow) or have more than one input/output ports, the data are decomposed and queued up in units of 32-bits wide. We will propose several solutions: Direct Wires in Chapter 4 and HiPR in Chapter 6.

Table 3.9: PRflow Fragmentation for Rosetta Benchmark

	Page#	Design Logic			Page Logic			Utilization		
		LUT	B18	DSP	LUT	B18	DSP	LUT	B18	DSP
digit_reg	10	43k	360	1	194k	804	1460	22.1%	44.7%	0.0%
optical_flow	9	28k	146	286	172k	792	1296	16.5%	18.4%	22.0%
rendering	6	11k	100	9	123k	672	960	8.9%	14.8%	0.9%
spam_filter	15	50k	204	256	288k	1212	2132	17.6%	16.8%	12.0%
face_detect	20	173k	296	145	391k	1764	2924	44.2%	16.7%	4.9%
bnn	22	65k	1196	4	429k	1956	3212	15.1%	61.1%	0.1%

3.8.3 Fragmentation

As we use fixed pages to map separate operators for all the benchmarks, fragmentation does exist as it is hard to utilize all the resources on one page. Table 3.9 shows the detail of the fragmentation. While it consumes 11–173 k LUTs for the designs, it consumes 123–429 k LUTs by using fixed pages to map. The utilization ratio varies from 8.92 % to 44.26 %. One possible solution to solve this is to use a customized overlay to map different designs, which will introduce some overhead for the initial compilation. Nevertheless, the fragmentation issue can be solved to some extent still with a short mapping time.

3.8.4 Application-specific Overlay Automation

Currently, PRflow can map separate operators at the C/C++ level to fixed PR regions at the layout level. However, the application can only be mapped when the operators are smaller than the PR region. This does not fit the initial implementation scenario, where the priority goal is to get an application to run and tune it up for later incremental refinement. As the global bandwidth is limited by the NoC, merging certain operators can move global data transmission to local internal data transfer. However, it is hard for a fixed page to hold the merged giant operator. To overcome this limitation, we will introduce HiPR in chapter 6.

Table 3.10: Compile Time Comparison between Grid Severs and Modern Workstations (in seconds)

	2.7 GHz Intel E5-2680		3.7GHz AMD Ryzen 9-5900X	
	Vitis	PRflow	Vitis	PRflow
hls	216	38	80	8
syn	3079	93	817	42
opt	314	197	120	57
place	1755	330	725	191
route	1320	234	634	61
bitgen	1320	234	634	61
total	**7537**	**972**	**2733**	**385**

3.8.5 Compilation Time on Modern Workstations

In Chapter 3.7.3, we compile the benchmarks on our servers, which are not the most advanced modern machines. We also compile *Optical Flow* benchmark on a more advanced workstation equipped with 3.7GHz AMD Ryzen 9 5900X 12 Core CPU with 24 processing threads and 128 GB of RAM. Table 3.10 and Figure 3.24 show the compilation time comparisons between server machines and modern workstations. We see modern workstations can accelerate both Vitis and PRflow by around 2.7×. PRflow can finish the compilation in around 6 minutes and still outperform standard Vitis flow by 7×.

3.9 Chapter Summary

PRflow offers users more design points between pure software simulation and raw FPGA implementation. To complement the hours-long compile time, PRflow provides fast, native-FPGA compile options that consume 20 minutes. The divide-and-conquer idea comes from the separate compilation and linkage that are widely used by software compilers. PRflow uses the streaming dataflow compute model, which abstracts away the detailed timing and logic implementation, and an NoC, which dynamically links separate physical blocks together without physical placement and routing. Combined with program deceptions in C/C++ compiled by modern high-level synthesis tools, this provides a more familiar coding environment for software developers. They can refine the C code, which can run both with

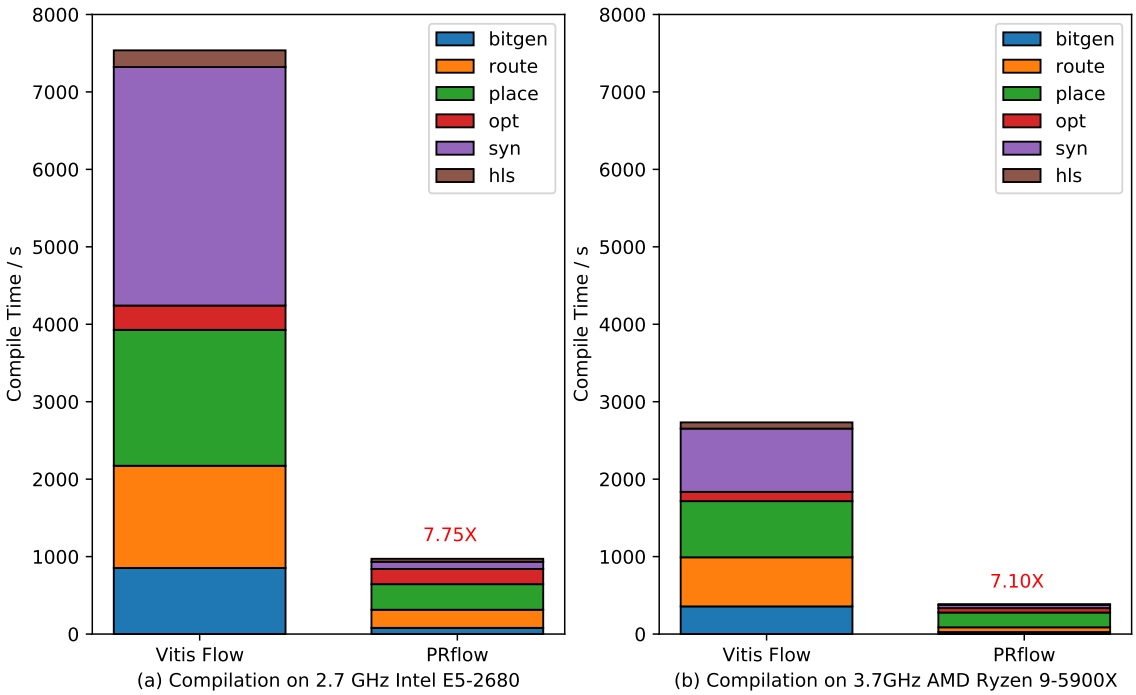


Figure 3.24: Compile Time Comparison Between Grid Severs and Modern Workstations

x86 platforms and FPGA platforms. With PRflow -O1 option, developers can flexibly move part of the entire program into FPGAs to evaluate the performance within 24 minutes. This provides an essential ramp-up between pure software and complete hardware implementation. Furthermore, the short edit-compile-debug turns to enable more trial times so that users are able to explore larger design space to obtain better performance. PRflow works with modern data-center FPGAs and interfaces to hide the low-level details of physical implementation, architecture, and CAD flow, providing an API closer to software compilation and linkage for CPUs and GPUs.

Chapter 4

More Bandwidth: Direct Wires

Direct point-to-point wires can be used to replace the Packet-Switched Network-on-a-Chip (PSNoC) for fast linking of the split FPGA blocks, offering higher inter-page bandwidth to improve the performance with less area overhead at no cost of compile time provided by PRflow. In chapter 3, we show that separate compilation for FPGAs by using a pre-compiled overlay can save compile time by 6.4–10.9 ×. A PSNoC is used to connect the separate compiled FPGA blocks dynamically without hardware placement and routing. Nevertheless, the lightweight PSNoC cannot meet the inter-page bandwidth for some critical links. In this chapter, we will show how the bandwidth issue can be solved by direct wires (DW), where the producer ports and consumer ports are connected directly by wires to take advantage of abundant on-chip interconnect wires while preserving the quick compile by pre-defining PR regions. The direct wires can reside in some PR regions compiled simultaneously with the compute pages. Adjacent pages can be connected directly by fast links with low latency, mapped directly to the pre-routed wires during the overlay generation stage. By mapping the same benchmark as Chapter 3.7, DW can offer 1.1–10× performance improvement and consume less 28–77% (LUTs) interface area overhead than PRflow.

4.1 Motivation

In this section, we will profile and quantify two issues in PRflow with PSNoC overlay: *bandwidth* wastage and *interface sharing area overhead*. Next, we will characterize the wire density of modern FPGAs by sweeping the number of wires to be routed over certain

boundaries under different timing constraints. This can be regarded as an up-bound of routing capability as routing is a global problem that can be easily affected by the real logic to be mapped [103]. Along this line, we introduce our direct wire (DW) idea and the basic techniques to support the implementation of DW (e.g., Skid Buffer/Relay Station, partition pins, etc.).

4.2 Limitations of PRflow

4.2.1 IO Bottleneck

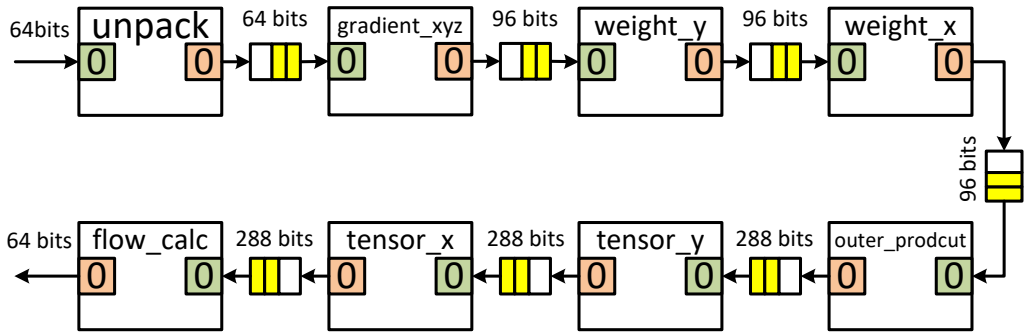


Figure 4.1: Optical Flow Decomposition with Datawidth Labeled

For the PRflow, we use a Packet-Switched Network-on-a-chip to connect separate FPGA blocks (PR pages) together. A uniform streaming bus is provided at each endpoint of the PSNoC for each page. As the bus between PSNoC and pages can only provide a fixed bandwidth of 0.8 GB/s (32 bits@200 MHz), it partially explains the performance loss in Table 3.7 for PRflow compared with normal Vitis Flow. When an operator has more than one input port or output port, they have to share the IO throughput of 0.8 GB/s. Moreover, IO with large datawidth also causes data transmission to be queued up by 32 bits per cycle.

Figure 4.1 shows the *optical flow* design from Rosetta Benchmark [146]. The datawidth of different links between operators varies depending on the specific computation. Some pages can consume or produce more than 32 bits per cycle. However, due to the fixed 32 bits data bus, it takes multiple cycles to send or receive data but only consumes one cycle to process the data. By profiling the consumed cycles for IO operations and compute

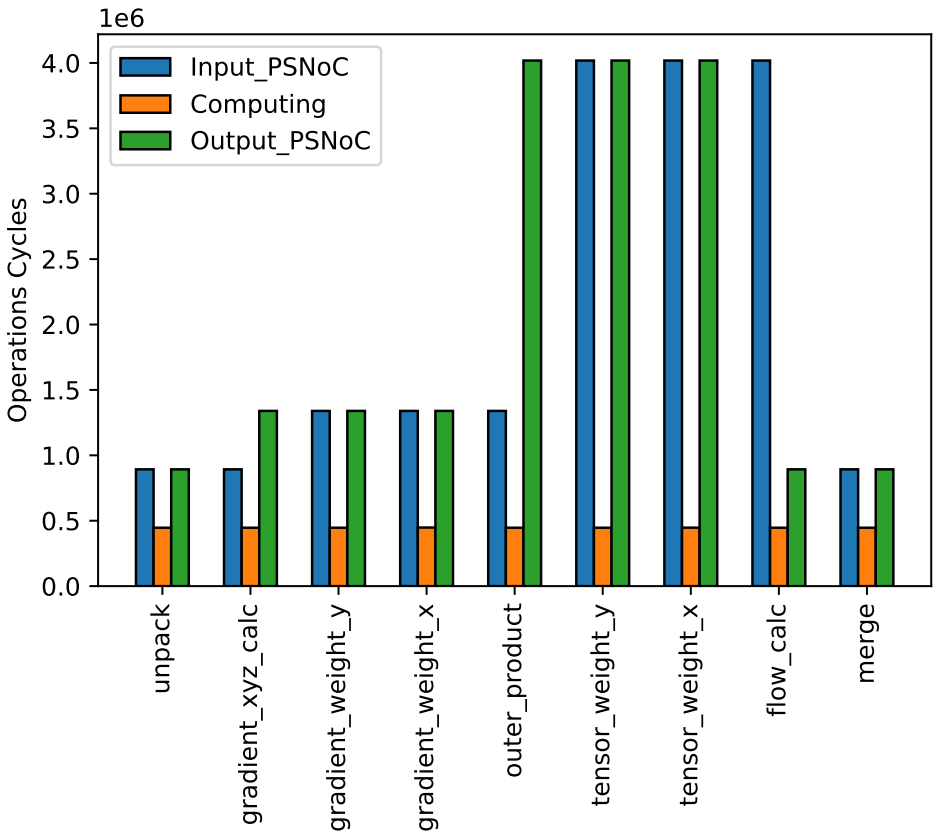


Figure 4.2: Optical Flow: IO and Compute Operation Cycles

operations in Figure 4.2, we see the IO and compute operators are unbalanced, and in the worst case, it can slow down the overlay performance by 9 ×.

In fact, we can profile the IO operations and compute operations for the benchmarks. Theoretically, the overall performance of an application is determined by the maximum of memory operations cycles and compute operation cycles. We use the normalized IO cycles and compute cycles by equations 4.1 and 4.2. For one benchmark, we first find the maximum compute operation cycles $Max\{AllComputeCycles\}$ from all pages. Then we normalized the compute operation cycles for all the pages by dividing this maximum one. We also divide the IO operation cycles by this maximum compute operation cycle.

We profile the ratio between $NormIoCycles$ and $NormComputeCycles$ in Figure 4.3 for all the pages from the benchmark set. The black line means that the IO cycles are

equal to the compute cycles for a page, which can be interpreted as the boundary between compute-bound and memory-bound. When the points are above the black line, the overall performance is limited by the slow IO operations on that page. It is especially worth noting the cases where the normalized compute cycles are close to 0, but the normalized IO cycles are high. This means the performance is significantly degraded by the IO operations (e.g., *face detection* and *spam filter*). Of course, we should also address the cases where the ratio is significantly high (e.g., *optical flow*). Nevertheless, we see for most pages, the IO and compute ratio is around 1 (Figure 4.4). This means only a limited number of pages have such unbalanced IO and compute operation cycles, and there is no need to increase the inter-page bandwidth for most cases.

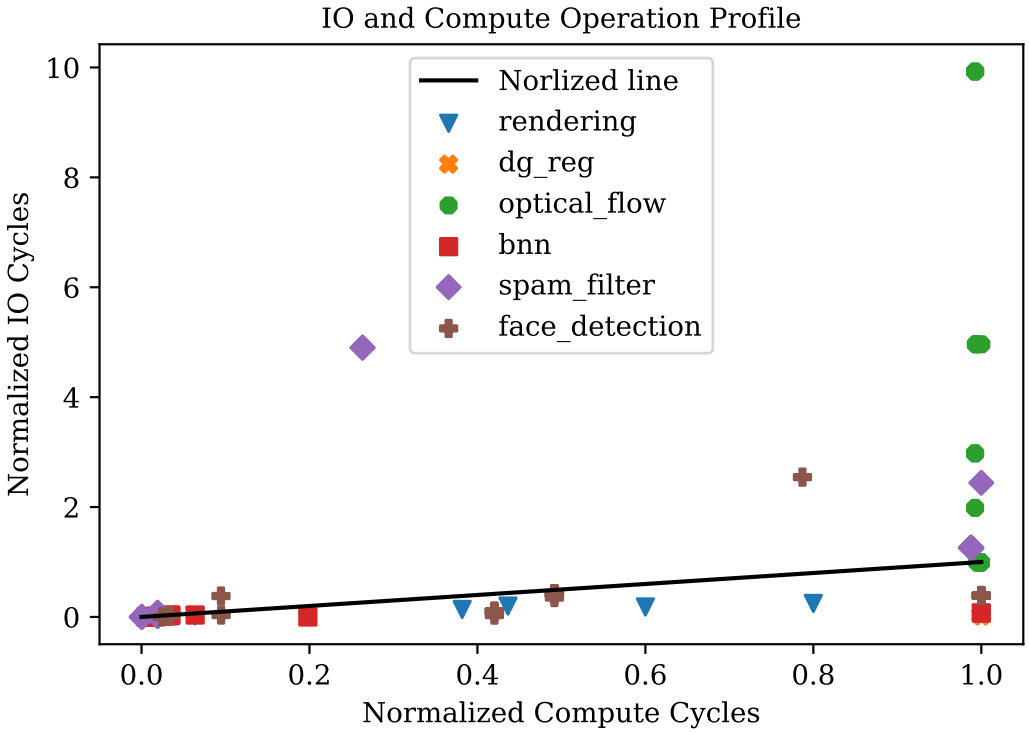


Figure 4.3: IO and Compute Operations for all Benchmarks)

$$NormComputeCycles = \frac{ComputeCycles}{Max\{AllComputeCycles\}} \tag{4.1}$$

$$NormIoCycles = \frac{Max\{InputCycles, OutputCycles\}}{Max\{AllComputeCycles\}} \tag{4.2}$$

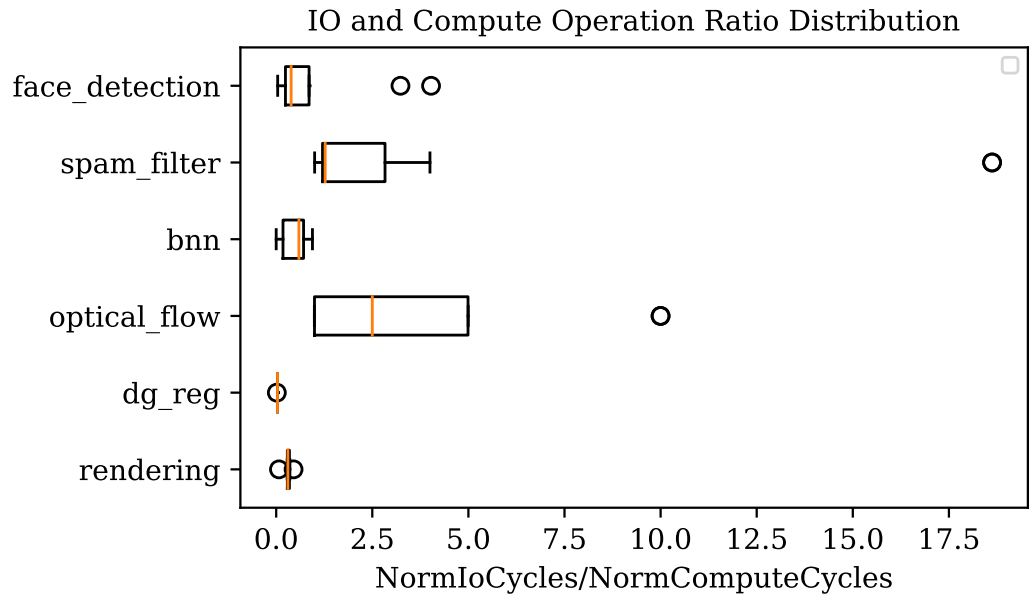


Figure 4.4: IO and Compute Cycles Ratio Distribution)

4.2.2 Area Wastage

The protocol in PRflow is similar to TCP/IP [39], with which data are transmitted in packets. Window buffer and the corresponding control logic are needed on both the output and input sides for the user operators to support windowed acknowledgment discipline [26] to guarantee that the PSNoC is not overwhelmed by too much traffic congestion. Moreover, the packets may be received out-of-order from PSNoC, special order labels are added as part of the header for each packet and re-order logic is necessary to store input packets into the input buffer. PRflow uses 48 bits to deliver 32 bits payload data, consuming one-third of the PSNoC bandwidth on the headers. As shown in Figure 3.5, the output ports need dedicated FIFOs to buffer the output data before injecting them into PSNoC. A round-robin multiplexer is used to share the only output bus to the PSNoC. For the input ports, a de-multiplexer is adopted to split input packets to the buffer of each port. Table 4.1 lists

the resource utilization with different IO numbers.

Table 4.1: Interface Resource Consumption

Overlay Type	Instance	LUTs	FFs	BRAM18s
PSNoC	Page_interface_1	1076	526	7
	Page_interface_2	1962	822	11
	Page_interface_3	2764	1106	15
	Page_interface_4	3360	1390	19
	Page_interface_5	4121	1674	23
	Page_interface_6	4923	1963	27
	Page_interface_7	5746	2247	31
DW	Relay_Station	37	66	0
	Stream_shell	71	38	1

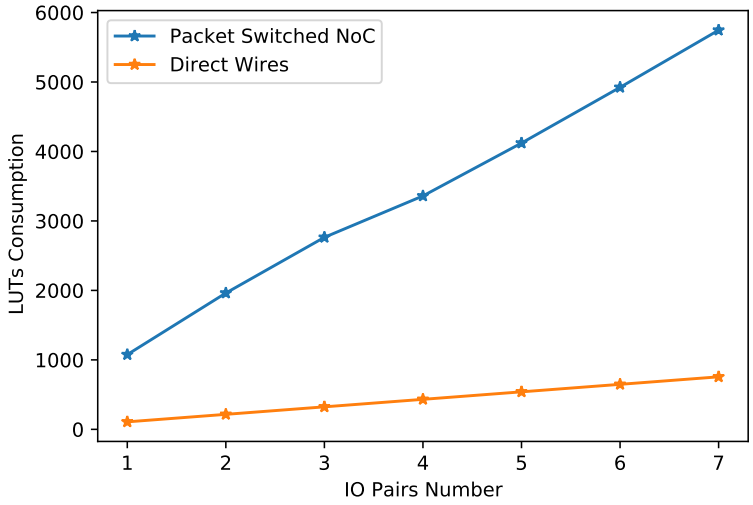


Figure 4.5: Interface LUTs Utilization for PSNoC and Direct Wires

Page_interface with 1–7 inputs and outputs are listed in the first 7 rows. For direct wire overlay, one pair of input and output needs one relay_station [17, 19] and one stream_shell listed in the last two rows. In Figure 4.5, we can see the interface LUTs consumption is roughly proportional to the IO pair number.

The PSNoC in Chapter 3 needs flow control logic to prevent packet loss by always reserving enough space on the receiving buffer for incoming data. Therefore, links are always initiated in pairs (one output port in the producer and one input port in the consumer).

We propose to use direct wires to initiate these links. This might consume more resources when IO numbers are high.

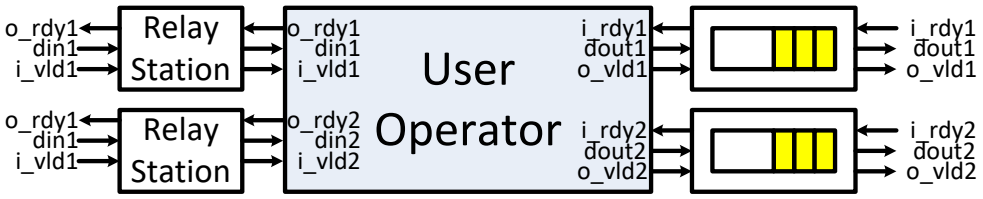


Figure 4.6: User Operator Wrapped with Relay Stations and Stream FIFOs

Instead of wrapping the user operator with a page interface, we propose to pack one user logic with relay stations and stream FIFOs. For example, Figure 4.6 shows an operator with 2 input and 2 output ports.

4.3 Direct Wires

We propose to use pre-compiled overlays with universal pages with direct wires stitched together as shown in Figure 4.7. To map a dataflow graph with operators, we pack each operator input port with a repay station to pipeline the input data and connect each output port with a stream fifo. Each operator is mapped to a uniform page by a coarse grain placer to meet the resource constraints. A coarse grain router is called to connect different operators under limited route resources between different pages. As we use the latency-insensitive model to prepare our application, extra relay stations can be added along the links when they route through different uniform pages. For example, page c and page d are connected across 2 extra pages. On each page, we add 2 relay stations to improve the time. The idea of merging interconnect and logic can date back to system design [34] and reconfigurable devices [9, 125, 79, 142]. We apply this idea to modern data-center FPGAs by Partial Reconfiguration technique.

This interconnect-and-logic sharing scheme may need more pages to be recompiled as different extra route-through logic may be inserted into a page even if the mapped operator in that page is not modified. Nevertheless, we still admit parallel compilation, given enough threads or cores by the cloud servers.

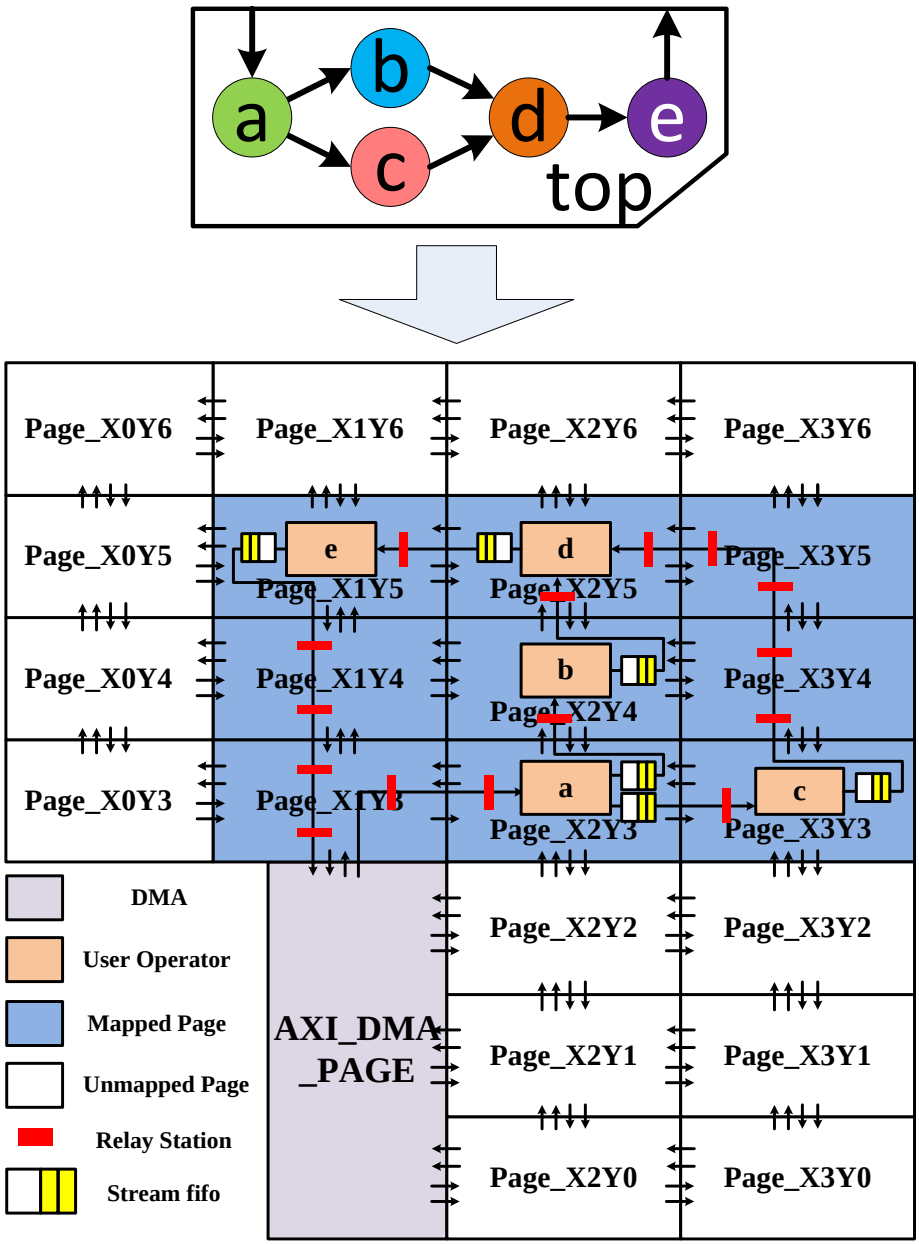


Figure 4.7: Direct Wire Overlay

4.4 Toolflow

The toolflow of direct wires is shown in Figure 4.8. The operators' C source files are compiled into Verilog files. Then an RTL synthesis is launched to compile the Verilog files

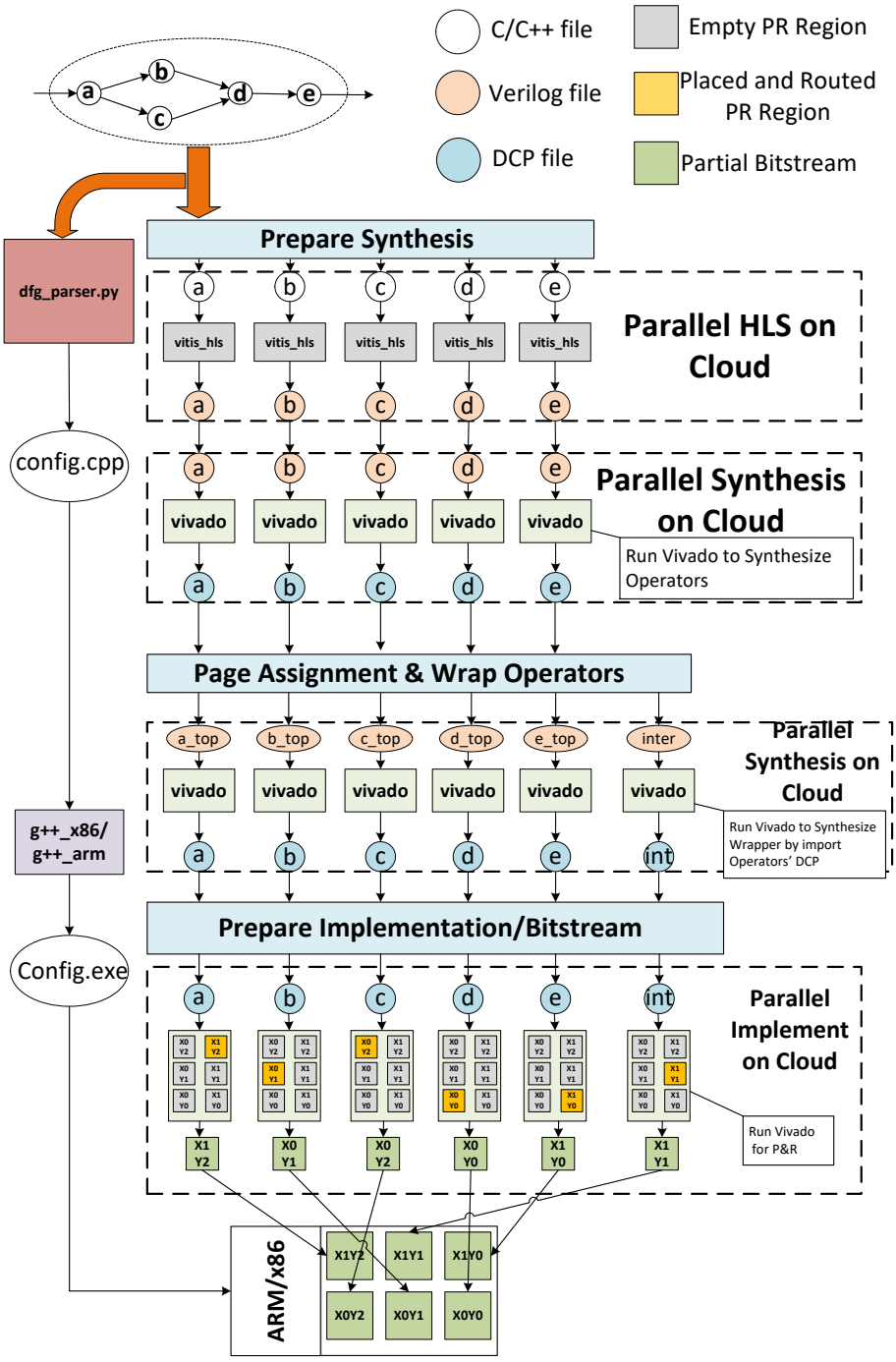


Figure 4.8: Direct Wires Toolflow

to the design checkpoints. At this point, all the resource requirements for the operators are available. The resource requirements and interconnect information are fed into our page assigner to place the operators on the proper pages. Iterative routing is performed to make sure the routing resource between two pages is not over-utilized.

We construct our page placer as Algorithm 2. We use Simulated Annealing to find feasible places for all the operators. The variables and constants for the algorithm are listed below.

$PR :=$ set of PR pages with $< x, y >$ coordinates;

$OP :=$ set of operators to be placed;

$W :=$ width of the device in units of pages;

$H :=$ height of the device in units of pages;

$T :=$ set of tile types considered (CLB, BRAM, URAM, DSP);

$r_t :=$ number of type t resources ($t \in T$);

$L :=$ set of all the links between 2 PR functions;

$x :=$ column coordinate for an operator or a page;

$y :=$ row coordinate for an operator or a page;

$op :=$ an operator ($op \in O$);

$l_{op_i,op_j} :=$ number of interconnect wires between PR regions pr_i and pr_j ($pr \in PR$, $l \in L$);

The target is to find a unique pair of coordinates for each operator in equation 4.3.

$$\begin{gathered} \forall i \in \{0, 1, .., |OP| - 1\}, \forall x \in \{0, 1, ..., W - 1\}, \forall y \in \{0, 1, ..., H - 1\} \\ op_i \rightarrow < x, y > \end{gathered} \tag{4.3}$$

For the Simulated Annealing, We randomly select two pages for each Simulated Annealing iteration and swap the mapping status. If both pages are empty, we skip this iteration; if both pages are mapped with operators, we swap the location of the 2 operators; if one is empty and another page is mapped with an operator, we move the operator to the empty page. The cost function is shown in Equation 4.4.

$$min: \ OverUsedResource + OverUsedWires \tag{4.4}$$

The $OverUsedResource$ is easily calculated by subtracting the operator resource from the assigned page resource. If it is below zero, we set it to zero. To calculate the $OverUsedWires$, we need to route all the streaming links. Fortunately, the scale of the operators is orders of magnitude smaller than the bit-wise routing. Figure 4.9 illustrates a naive router the placer uses for each iteration.

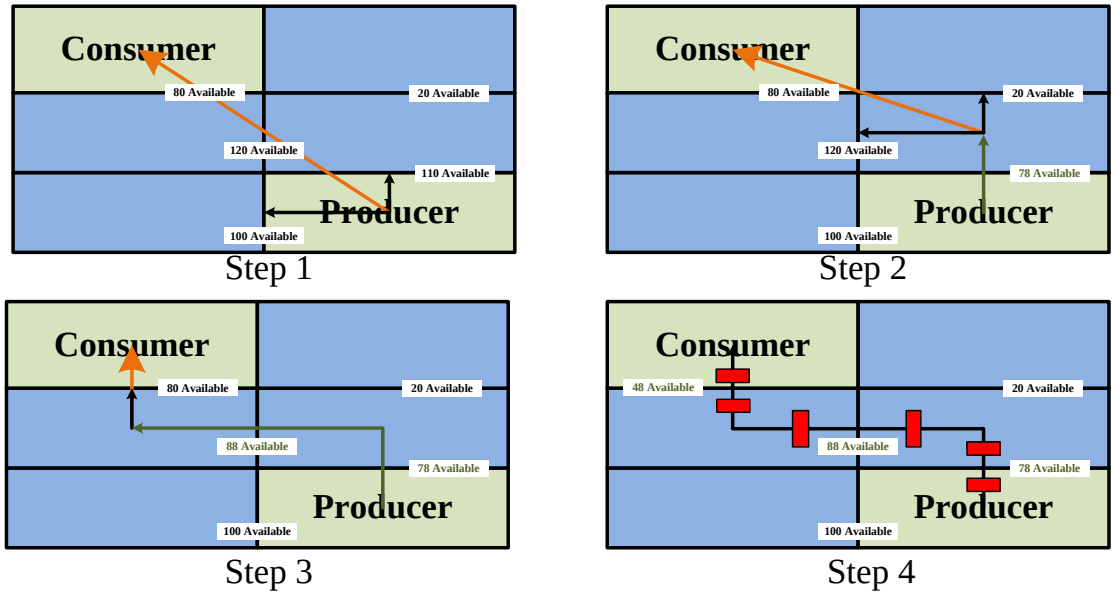


Figure 4.9: Naive Router

Algorithm 2 Direct Wires Page Placer

```
1: procedure PagePlacer(Overlay Parameters, Resource Requirements, Interconnection)
2:     for operator in operators_set do
3:         Randomly generate < x, y > for operator
4:     end for
5:     T ← T_0
6:     CurrCost ← CostFunction()
7:     MinCost ← CurrCost
8:     while CurrCost > 1 do
9:         i ← 0
10:        while i < TRIAL_NUM do
11:            Randomly select 2 page (page_j, page_k)
12:            Swap the state of page_j and page_k
13:            GreedyRoute(all operators)
14:            df = CostFunction() - CurrCost
15:            if df < 0 then
16:                CurrCost ← CostFunction()
17:                if CurrCost < MinCost then
18:                    CostMin ← CurrCost
19:                    Best Set of Operator Shape ← Current Set of Operator Shape
20:                end if
21:            else
22:                if exp(-df/T) > random_possibility then
23:                    CurrCost ← CostFunction()
24:                else
25:                    Swap the state of page_j and page_k
26:                end if
27:            end if
28:            i ← i + 1
29:        end while
30:        T ← η×T
31:    end while
32:    if MinCost < 1 then
33:        return The Best Set of Operator Coordinates and Routing Graph
34:    else
35:        return Fail
36:    end if
37: end procedure
```

4.5 Evaluation

We implement Direct Wires framework on top of PRflow in Chapter 3. We map Rosetta High-Level Synthesis C++ Benchmarks for FPGAs [146]. We map the designs to the Alveo U50 Data Center card [135] with a Virtex UltraScale+ XCU50 FPGA and 8 GB HBM. We perform the mapping on a cluster of 8 servers, each with 2.7 GH Intel E5-2680 CPUs and 128 GB of RAMs by running Vitis 2022.1.

4.5.1 Overlay Design

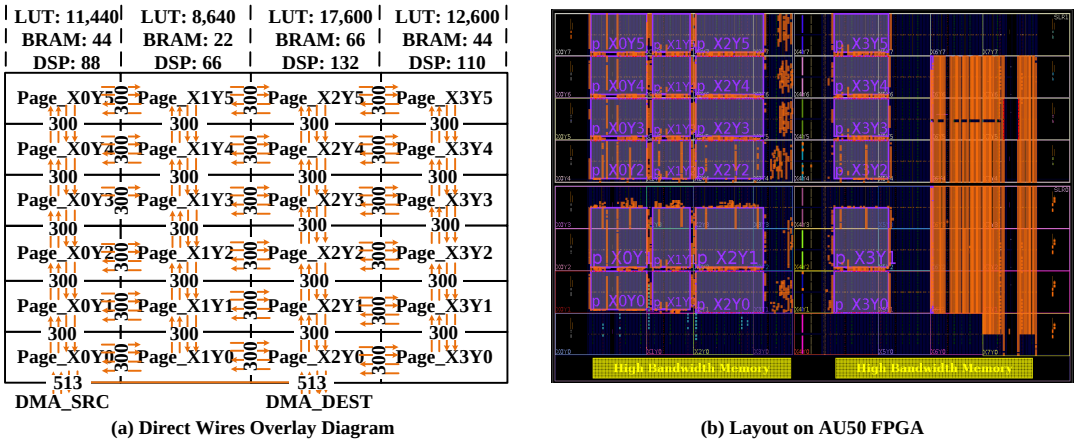


Figure 4.10: Direct Wires Overlay Implementation

Figure 4.10(a) illustrates the detail of the direct overlay. The adjacent pages are connected by two sets of 300 wires. The two sets of wires have opposite directions. The biggest challenging part is choosing the number of wires between pages. The IO bandwidth of a page is 240 Gbps (300×200MHz×4). Due to the heterogeneous resource distribution of FPGAs, we define 4 types of pages. The page resource detail is shown in Figure 4.10(a). The layout of the implementation on AU50 is shown in Figure 4.10(b). The left orange part is the firmware used by Xilinx.

Table 4.2: Rosetta Benchmark Performance on XCU50

	Vitis Flow		PRflow		Direct Wires	
	Freq (MHz)	Runtime	Freq (MHz)	Runtime	Freq (MHz)	Runtime
digit_reg	200	2.9 us	200	3.3 us	200	3.0 us
optical_flow	200	2.4 ms	200	26.6 ms	200	2.5 ms
rendering	200	1.5 ms	200	2.6 ms	200	1.5 ms
spam_filter	200	16.8 ms	200	72.3 ms	200	19.9 ms
face_detect	200	19.1 ms	200	125 ms	†	†
bnn	150	5.1 ms	200	7.1 ms	200	4.4 ms

† `Face Detect cannot map due to routing congestion on a new device with a new version of Vivado. It can be mapped with the previous ZCU102 platform in [120].`

4.5.2 Performance

Mapping the Rosetta HLS Benchmark [146] to direct wires overlay, we tabulate the performance benefits in Table 4.2. The application runtime is listed in column fifth and seventh for PRflow and Direct Wires framework. We can see Direct Wires overlay can improve the performance by 1.1–10 ×. For *optical flow*, we increase the bandwidth between pages, which improves the performance by 10 ×.

4.5.3 Parallel Compile Time

Table 4.3: Rosetta Benchmark Compile Time on XCU50 (in seconds)

Benchmark	Vitis Flow	PRflow		Direct Wires		
		Logic Page	Speedup	Logic Page	Wire Page	Speedup
digit_reg	9120	931	9.8	929	758	9.8
optical_flow	7537	974	7.7	922	507	8.2
rendering	6181	627	9.8	700	410	8.8
spam_filter	7303	987	7.3	675	574	10.8
bnn	9546	1488	6.4	1829	472	5.2
face detect	10869	1228	8.8	†	†	†

† `Face Detect cannot map due to routing congestion on a new device with a new version of Vivado. It can be mapped with the previous ZCU102 platform in [120].`

Table 4.3 tabulates the compilation time for different cases. We use Vitis as our baseline. We see the DW compilation time is slightly longer than PRflow. This is because we placed more IOs around the PR pages, which leads to more partition pins around the bounder. This can add more placement and routing load for PR implementation. Nevertheless, we can still see 5.2–10.8 × speedup compared with standard Vitis Flow.

4.5.4 Area Overhead

Table 4.4: Application Resource Comparison – PSNoc Vs. DW on AU50

Benchmark	Resource	PRflow			Direct Wires				
		User	Interface	Totale	User	Interface	Routing	Total	Inf Save†
digit_reg	LUTs	36282	10206	46488	46285	1342	3226	50853	55.2%
	FFs	35204	5386	40590	47481	760	2560	50801	38.4%
	B18s	300	60	360	334	24	0	358	60.0%
	DSPs	1	0	1	1	0	0	1	-
optical_flow	LUTs	17550	10744	28294	17031	1213	6507	24751	28.1%
	FFs	15788	8218	24006	19245	340	6379	25964	18.2%
	B18s	84	62	146	92	52	0	144	16.1%
	DSPs	286	0	286	286	0	0	286	-
rendering	LUTs	8728	10465	19193	8214	718	2612	11544	68.2%
	FFs	9836	6011	15847	9840	263	2052	12155	61.5%
	B18s	64	58	122	64	31	0	95	46.6%
	DSPs	18	0	18	18	0	0	18	-
spam_filter	LUTs	15099	35485	50584	15552	1073	6780	23405	77.9%
	FFs	17030	16955	33985	18051	564	5017	23632	67.1%
	B18s	10	194	204	10	65	0	75	66.5%
	DSPs	256	0	256	256	0	0	256	-
bnn	LUTs	37449	24318	61767	36888	1699	5560	44147	70.1%
	FFs	28983	11656	40639	28887	798	4609	34294	53.6%
	B18s	1046	142	1188	595	63	0	668	55.6%
	DSPs	4	0	4	4	0	0	4	-

† `For Direct Wires, the interface overhead is calculated by adding interface and routing resources. The interface of DW is compared with the interface of PRflow.`

Table 4.4 lists the detail of the resource consumption for PRflow and Direct Wires framework. We can see the area overhead for the interface is reduced by 28–77% (LUTs).

This means as we leave more space for the user logic instead of the interface.

4.6 Discussion

4.6.1 Fixed-Wiring Limitations

The pure DW gives up the complete virtualization of communication provided by the PSNoC. That is, the DW solution depends on designs not requiring more user ports than the overlay can support. One solution is to serialize lower bandwidth ports (e.g., provide a logical 32b port with an 8b physical, DW path) to fit within pre-defined wiring constraints. Another is to consider a hybrid network with a PSNoC for fallback after exhausting high-speed DW capacity for the high throughput links.

When we encounter designs that require more ports than any overlay can support, this suggests the need for a new overlay. Given an iterative development style, we can fall back to the PSNoC, and this new overlay can be generated to support later revisions of the design. Ideally, custom overlay generation would be automated, so that a new, suitable overlay would become available in a few hours.

4.6.2 Higher clock Frequencies

For our overlay, we demonstrate our idea by using the clock frequency of 200 MHz for our overlay. The clock frequency is determined by sweeping the inter-page wire numbers and clock frequencies to get the highest inter-page bandwidth without timing violation similar to [120]. Modern FPGA fabrics are claimed to be able to run at high frequencies (e.g., 600 MHz). However, we find our benchmarks can seldom achieve the best performance under the highest clock frequency. This is because vendor HLS compilers will add more pipeline stages to meet higher clock frequencies which may make the execution time of an application even worse. For example, an application can run at 200 MHz by producing one output per clock cycle, and it can also run at 250 MHz by producing one output per 2 clock cycles. In this scenario, lower clock frequency generates better performance. In the future, we can generate different overlays with various clock frequencies and inter-page wires for users to choose from, so that users can get the best performance for their designs.

4.7 Conclusion

In this chapter, we show how to use the Direct Wires framework to exploit separate compilation and linking for FPGA design. We show the link can be mapped by using real wires instead of using Packet-Switched NoC to virtualize the communication. This can improve inter-page bandwidth without degrading the performance due to the IO bottleneck by PSNoC. As separate operators are compiled independently on their own pages, we can still benefit from parallel compilation without sacrificing the application performance.

Chapter 5

Fit Size Early: Soft Cores

We realize the divide-and-conquer compilation strategy in Chapter 3 and list two limitations of PRflow: 1) the IO-bottleneck issue and 2) the size-fit issue. In Chapter 4, we present Direct Wires to address the IO-bottleneck issue. In this chapter, we will show how to partially resolve the size-fit issue by using soft cores.

5.1 Motivation

With the rise of high-quality C-to-gates HLS compilation, it is becoming viable to craft high-performance FPGA-accelerated designs in C. However, just as arbitrary Verilog is not synthesizable or efficient, arbitrary C won't necessarily produce efficient and high-performance FPGA accelerators. Nonetheless, it should be possible to incrementally refine an application written in C to tune it properly for HLS to FPGA mapping. While incremental refinement with frequent recompilation and testing is a solid strategy for software development, slow (hour-long) monolithic FPGA compilation discourages or prevents rapid, incremental development and tuning. To avoid slow simulation on CPU-based workstations, logic emulation is an alternative for debugging and verification. [109, 4] propose to decompose the designs into split components and partial crossbars or time-multiplexed networks to link the separate components back together. However, these logic emulators did not achieve fast compilation as our Direct Wires framework and sacrificed a magnitude of performance compared with raw FPGAs. We present techniques and methodologies to allow incrementally refinement of dataflow-coordinated C computations into efficient FPGA-mapped computations. In

particular, we support (1) separate compilation and linkage of stream-connected dataflow operators to avoid the need to recompile unchanged parts of the FPGA and (2) fast compilation of the *same* C source to softcore processors that can be linked in place of the FPGA logic and communicate uniformly over compatible stream links. C can be mapped to softcore processors quickly (<4 seconds) with no limitations on operator sizing beyond memory capacity, allowing continuous, rapid validation of functional refinements [108]. Operators can be moved to softcore for rapid, in-situ functional testing while FPGA logic is being recompiled. As operators are refined and sized for the FPGA logic, they can be incrementally moved to the FPGA with short (10–20 minutes) compiles, even if parts of the applications are still running on soft cores [44, 45].

5.2 DIRC: Dataflow Incremental Refinement of C

Based on the PRflow framework in Chapter 3, we take a further step to overcome the size-fit issue, providing a path for **D**ataflow **I**ncremental **R**efinement of **C** (DIRC) code that allows the developer to gently migrate applications from pure software to hybrid or pure hardware running on the FPGA. Development can proceed like software, retaining the ability to make quick, incremental changes and always run the code after every change by mapping all the operators to pre-compile RISC-V cores connected by our PSNoC, shown in Figure 5.1(a). Here we call this compilation strategy `-O0`.

Next, we can move some operators into hardware pages, as shown in Figure 5.1(b). If the operator is too big to fit in a target partial reconfiguration region, the developer can refactor the operators to split the operation into multiple operators. If the operator becomes the performance bottleneck, the developer can replicate the operator to increase parallelism. These changes can all be performed as incremental refinements to the operator C graph; graph nodes can always run on the softcore processors, with some nodes running on the FPGA as the operators are sized appropriately. Finally, we seamlessly finish the transition to PRflow implementation in Chapter 3, shown in Figure 5.1(c). Here we call this compilation strategy `-O1`.

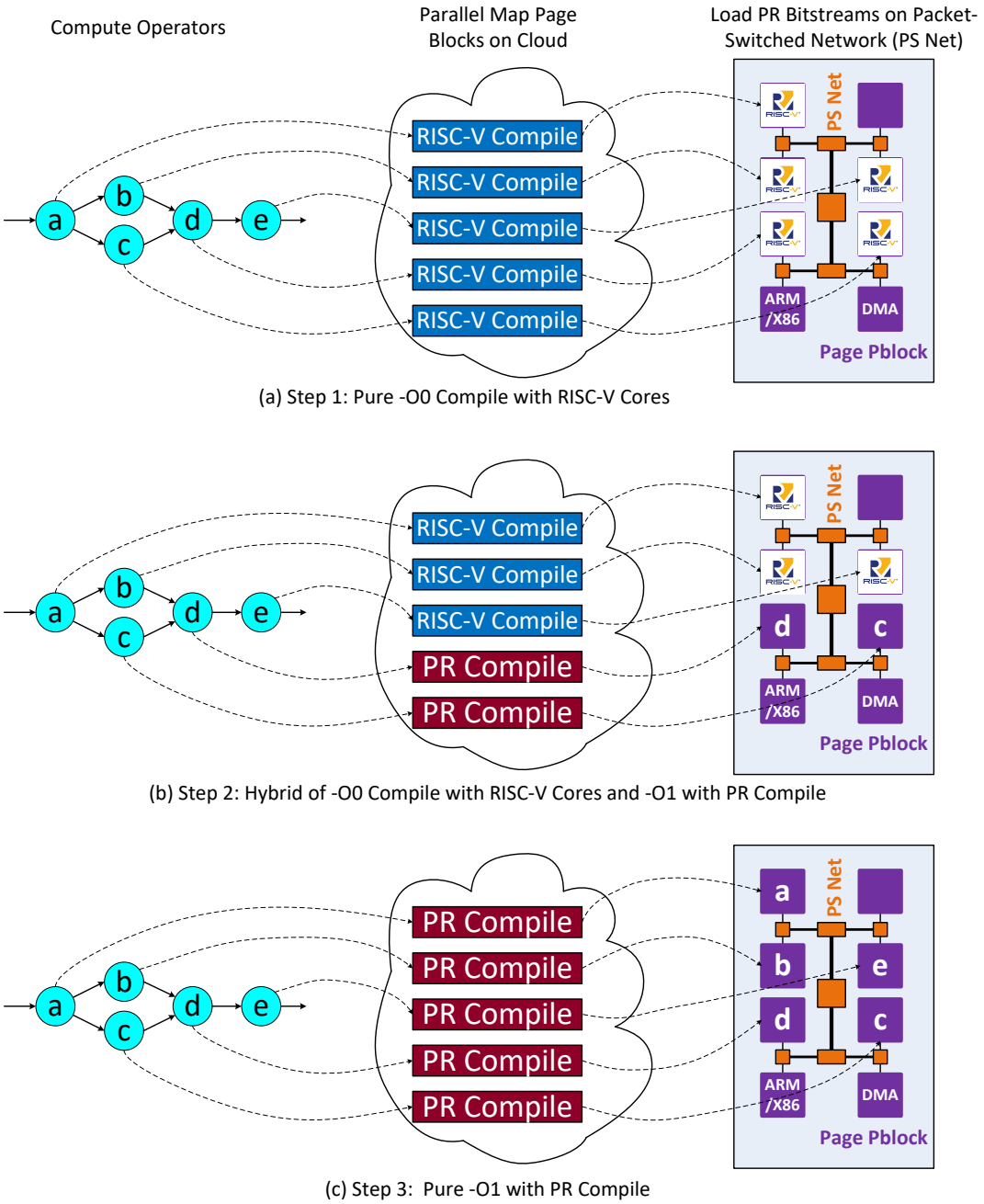


Figure 5.1: Fast Incremental Mapping Strategy

This methodology provides the missing `-O0` compilation strategy, familiar with software that offers a quick, but possibly low-performance, compilation to allow testing and debugging to proceed rapidly. It also provides separate compilation and linking familiar with the software development. We further show that the ability to remap streams and operators to processor cores enables more software-like instrumentation and debugging (Chapter 5.4).

We provide an open-source release for our dirc_riscv implementations on github[4].

5.3 Operator Discipline

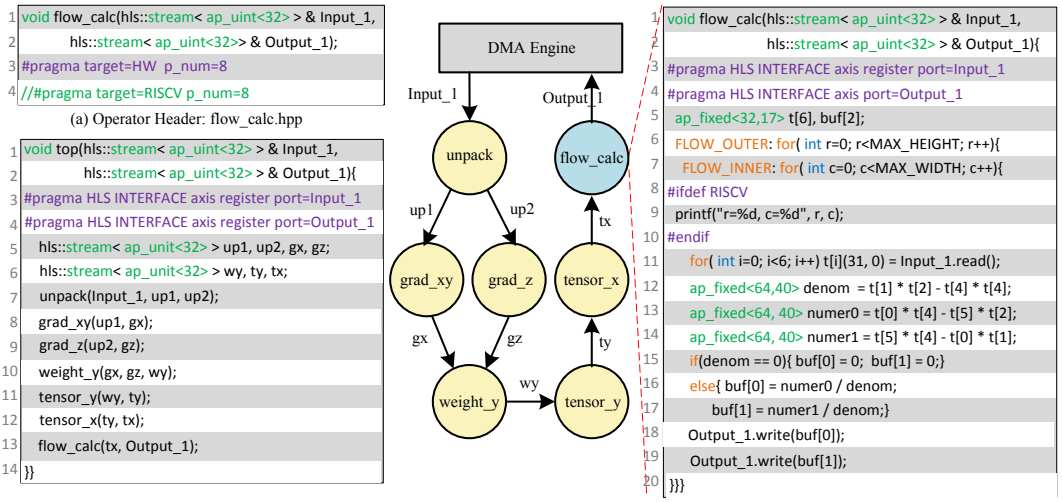


Figure 5.2: Code Discipline Example

There is some discipline required to design operators that are mutually compatible with the processor cores and the FPGA fabric. Our discipline includes:

- `hls::streams` and associated API operations (Chapter 5.3.1) are used for all communication.

- Operators are limited to 6 input streams and 6 output streams; this is not fundamental, but a particular system will need to pick some MAX_STREAMS to support in

[4]https://github.com/icgrp/pld2022.git

the hardware design. This limit impacts both the packet headers for the BFT and the buffers and control allocated for the softcore processors.

- Operators should obey standard HLS prohibitions such as no allocation or recursion; if you want to exploit processor-only operations when they are on the processor, like `print`, it should be guarded by suitable `ifdef` software macro guards (Figure 5.2(d), Lines 8–10).

- Operators use a standard set of datatypes with compatible implementations for processor and FPGA (e.g., `ap_int`, `ap_fixed`).

5.3.1 Streams

We provide a processor implementation of Vivado's `hls::stream` [133] so that operators can use a single, common stream API independent of where they are mapped. Our processor streams are latency insensitive, similar to the Vivado versions. Code compiled to the FPGA using native Vitis_HLS `hls::stream` implementation. Figure 5.2(d) Lines 11, 18, and 19 show how to use this class type to read and write data.

5.3.2 Application Composition

We support a stylized discipline for the top-level composition of the operator graph along with pragmas that specify where each operator is mapped. Top-level operator composition instantiates operators as function calls (See Figure 5.2(b)) and can be compiled with Vitis_HLS for a monolithic compilation; alternately, it can be compiled with our tools to generate the linking graph needed to configure the PSNoC. An operator with mapping control directives is shown in Figure 5.2(a) Lines 3–4. Each operator has a line with a target specification. Changing the target will change whether the page is loaded as a native FPGA partial bitstream or a standard processor overlay loaded with a compiled processor instruction stream.

Changing the target also sets up the compiler dependencies to build the appropriate bitstream or instruction stream.

5.4 Debugging

The ability to dynamically link in processor-mapped operators enables rapid inspection and debugging [96, 53].

Initially, it allows us to move a single operator over to hardware for testing. The stream links allow the software to feed the operator data and consume its results without additional harness setup to run the operator.

If we have a mapped design where we encounter new bugs, we can move an operator back to software for inspection and add `printf` statements just as we might use in software debugging, shown in Figure 5.2(d) (Lines 8–10). The printed output will be wrapped as debugging packets and sent back to the ARM controller/X86 CPU, which has FIFOs to receive debugging packets from one RISC-V core. This is done without recompiling any hardware since the connectivity over the PSNoC is configured with the configuration packets over the NoC and does not require the recompilation of the neighbors of the operator to be moved to software. Similarly, it is possible to interpose an operator in an existing stream link by configuring the producer to send its data to the interposed operator and the interposed operator to send its output to the original consumer. This interposed operator can then be programmed to perform inspection and instrumentation, such as counting data tokens or printing them out. The interposed operator can perform the role of an integrated logic analyzer [130], again without the need to recompile the entire design. The interposed page can also act as a larger data buffer. These token counts and stream inspection are often useful in debugging deadlock scenarios that arise in dynamic dataflow streams [96].

5.5 Softcore Integration

We integrate RISC-V [92] processors at the BFT leaves (Figure 5.3) for our softcore processor targets. The RISC-V core configurations are pre-compiled into a partial reconfiguration bitstream for each page position, so it is only necessary to load the page configuration, load the instruction memory with the compiled RISC-V ELF contents (Chapter 5.5.5), and send the associated page interface stream-linking configuration commands (Chapter 5.5.4) to re-

link a freshly compiled processor-mapped operator. We typically remap operators between native FPGA and softcore processor overlays to the same partial reconfiguration page so that no changes or remapping is needed for the rest of the operators.

5.5.1 Processor Prototype Implementation

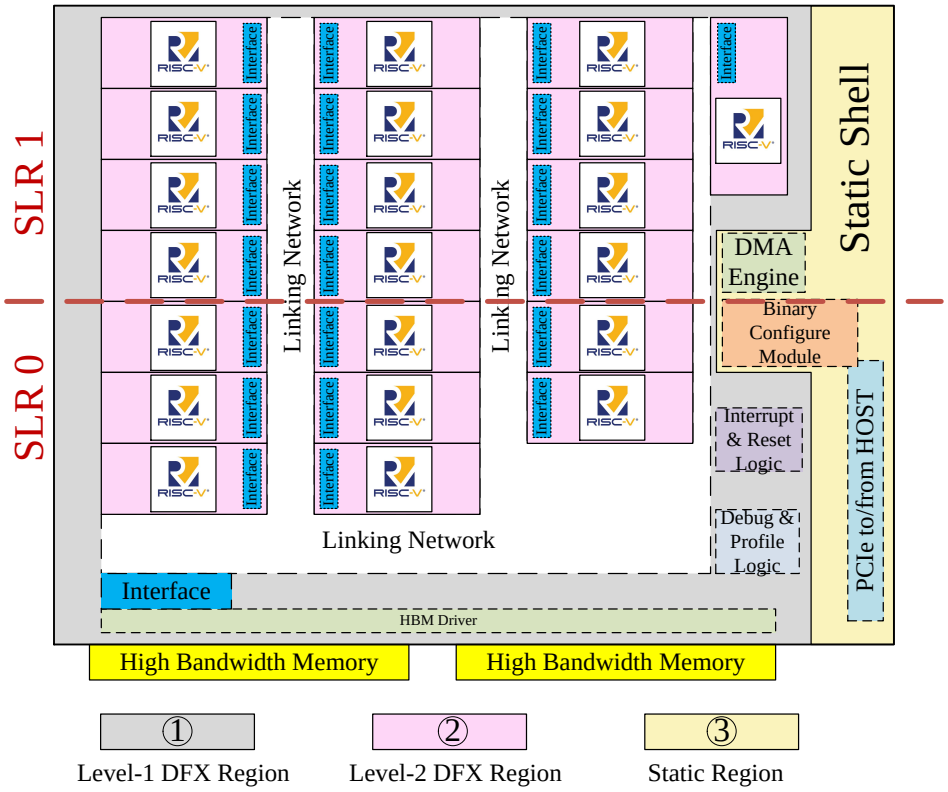


Figure 5.3: RISC-V Overlay

We start with a PicoRV32 soft processor [118] for the processor softcore overlay target. We use the `picorv32` with the integer multiplier enabled. This PicoRV32 provides a simple native memory interface, running at 200 MHz to match our BFT frequency. In this configuration, the PicoRV32 needs 2,504 LUTs, which easily fits in our partial-reconfiguration pages along with page interface logic. The PicoRV32 uses unified instruction and data memory. We can equip the RISC-V core with different memory sizes according to our page size and operators' IO numbers. We support at most 192KB (96 BRAM18s) of instruction memory.

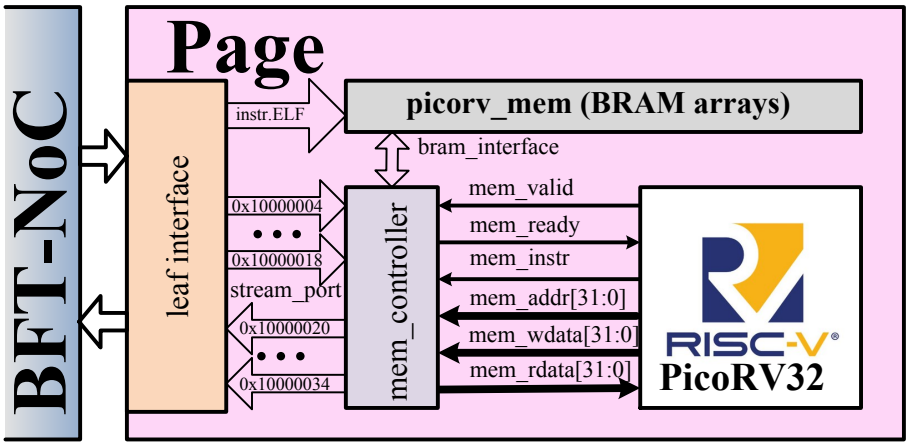


Figure 5.4: RISC-V Integration for One PR Page

5.5.2 Processor Integration

Using the native memory interface for RISC-V, we can easily manipulate the memory data bus. We define memory address 0-0x0FFFFFFF as the instruction memory and data memory. We define memory address 0x10000004-0x10000018 as stream_in ports and 0x10000020-0x10000034 as the stream_out ports. The *mem_valid* and *mem_ready* from the native memory interface can be utilized as handshake signals. By adding a hardware transfer module, the 6 stream_in/out ports can be used for the page_interface to communicate with the RISC-V cores (Figure 5.4).

5.5.3 Software Libraries

To take advantage of customization features, FPGA HLS C++ code often uses datatypes from `ap_unit` and `ap_fixed` libraries. These arbitrary-precision variables can be mapped to FPGA logic using minimum LUTs, rather than 32b or 64b datapaths on processors. However, Xilinx `ap_unit` and `ap_fixed` libraries use more than the minimum number of bits to represent these types, which can be a challenge when our partial reconfigurable pages only have 48 or 96 BRAM18s. Therefore, we develop our own, more memory efficient, `ap_unit` and `ap_fixed` libraries that are compatible with the existing Xilinx HLS C++ code. Mapping all the operators from Rosetta Benchmark, the ELF size is typically 32–64

KB, consuming 16–32 BRAM18s.

5.5.4 Page Stream Linking

The overlay floorplan is shown in Figure 5.3. We use the same method to link the separated PR pages back together as in Chapter 3. The embedded ARM processor controller/X86 CPU is responsible for sending configuration packets, which are generated according to the *top* function shown in Figure 5.2(b). For each stream link, the ARM processor/X86 CPU will send the corresponding configuration packets over the BFT NoC to configure the destination and source ports in the respective page interfaces.

5.5.5 Program Loading

For each RISC-V core on the partially reconfigurable page, we can download the ELF file into the instruction memory via our BFT NoC. Our tool can automatically parse the header file for each operator. As shown in Figure 5.2(a), if we enable Line 4, DIRC will compile the RISC-V ELF file and convert the ELF file to C++ arrays, which will be included by the ARM controller/X86 host CPU. It takes less than one second to send all the ELF arrays to all the RISC-V cores.

5.6 Benchmark Evaluation

We map Rosetta High-Level Synthesis C++ benchmarks for FPGAs [146]. We map the designs to the Alveo U50 data center card [135] with an UltraScale+ XCU50 FPGA and 8 GB HBM. We perform the mapping on a cluster of 8 servers, each with 2.7 GH Intel E5-2680 CPUs and 128 GB of RAMs by running Vitis 2022.1.

We divide the chip into 21 user logic pages, a dedicated packet-switched network, and one page that combines a portion of the packet-switched network with 1 DMA Engine (Figure 5.3). We use a Hoplite BFT [61, 62] for the packet-switched network, running at 200 MHz with 32b data payload. The interface page logic in each partial-reconfigurable page allows the configuration of the consumer address by packets on the BFT network [123].

We map the full set of designs from the Rosetta Benchmark Suite [146]. Figure 5.5 shows the compilation time for soft-cores distribution over all the pages from all full benchmark

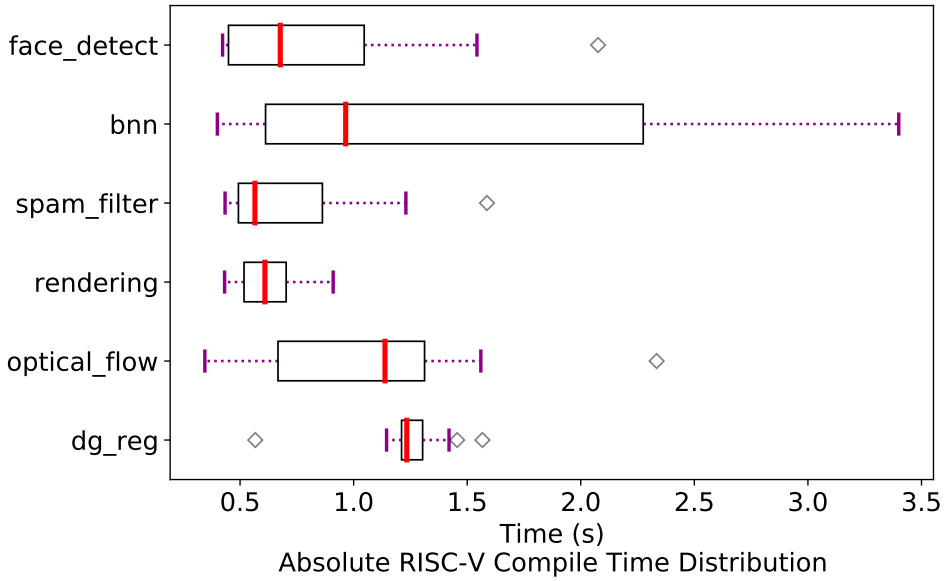


Figure 5.5: Softcore Page Mapping Time

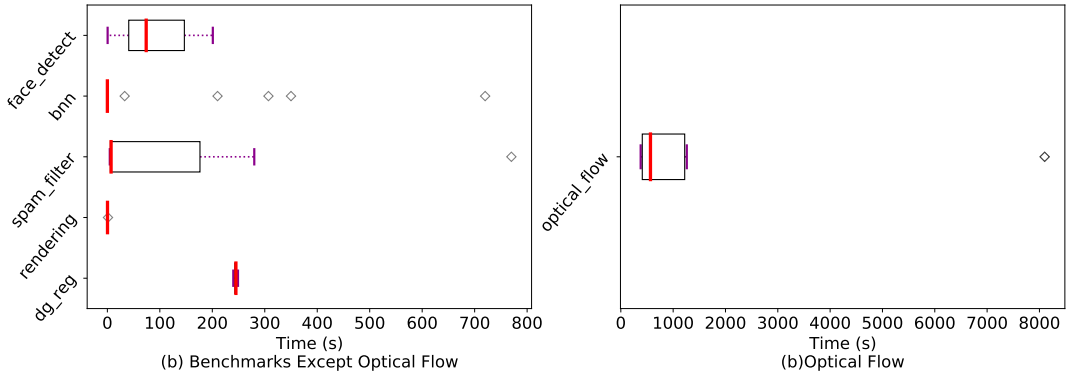


Figure 5.6: App Execution Time with One Page Mapped to RSIC-V Core

Table 5.1: Rosetta Benchmark Applications

	Vitis Flow		PRflow		Soft Cores		x86	Vitis Emu
	Freq (MHz)	Runtime	Freq (MHz)	Runtime	Freq (MHz)	Runtime	Runtime	Runtime
digit_reg	200	2.9 us	200	5.4 us	200	137 s	81.3 ms	90.6 ms
optical_flow	200	2.4 ms	200	26.6 ms	200	10935 s	2.8 s	70s
rendering	200	1.7 ms	200	2.6 ms	200	3 s	16.1 ms	671.0 ms
spam_filter	200	16.9 ms	200	72.3 ms	200	752 s	354.8 ms	12.3 s
face_detect	200	19.1 ms	200	125.0 ms	200	527 s	6.9 s	57.8 s
bnn	150	5.1 ms	200	7.1 ms	200	983 s	27.8 s	2074s

sets. We can see all the pages can be compiled into executable-format files within 4 seconds.

Table 5.1 reports the basic characterization of the benchmarks. Mapping all the operators of the application to pure soft cores, we see the runtime is worse than X86 and Vitis Emulation. This is predictable since the RISC-V cores we adopt are area-optimized, not performance optimized: the CPU is not pipelined, and no floating-point multipliers are available. *Optical Flow* suffers from a slow software floating-point implementation. In Chapter 3.7.3, we see native FPGA page region mappings typically take 10 minutes, with some taking up to 24 minutes. Figure 5.5 shows the ability to get up and running with soft-core operator compiles in seconds and the ability to migrate individual operators to native FPGA mappings in tens of minutes. The FPGA mappings can run in parallel with debugging and regression on the softcore-mapped operator. Figure 5.6 shows the distribution of throughput when mapping a **single** operator to the softcore processor and leaving the rest of the operators mapped to the separately-compiled partial reconfiguration leaves. Except *Optical Flow*, most cases can be executed with around 200 seconds, represented by the median values in Figure 5.6(a). For *Optical Flow* benchmarks, the median value is around 500 seconds. 2 pages are the extreme cases where a large number of floating-point multipliers are needed. Figure 5.6 also illustrates several things. Functionally, this illustrates the interoperation of mixed designs, with part of the design on the processor cores and part on native FPGA logic. The results directly represent fast remap debugging scenarios once you have a complete design. The throughputs here are much closer to the full, separately-

compiled partial reconfiguration page cases, since they are only bottlenecked by a single processor-mapped page. Combined with the all-processor mapped case in Table. 5.1, this provides a bracket on the performance one will see with mixed designs during development where some operators are on the softcore processors and some have already migrated to native FPGA implementations.

5.7 Discussion

This chapter shows how to use RISC-V cores to map operators to realize software first for incremental development. However, the performance of the RISC-V cores is still poor compared with X86 or FPGA accelerators. For the *optical flow* benchmark, we see the total runtime is even longer than the FPGA compilation time. This is because our RISC-V is a simple, lightweight version. We can speed up the execution by pipelining the RISC-V cores as general CPUs. Additionally, some common hardware cores (e.g., floating point multipliers, vector units [141], etc.) are also missing in our RISC-V CPUs. We believe the performance of pure `-O0` in this dissertation can be greatly improved by supporting these common acceleration methods in CPU.

In addition, great efforts have been put into reducing the size of the ELF file to fit the limited size of the on-chip BRAMs in PR regions. However, we cannot completely address the issue when an array with a big length is utilized in an operator. One possible solution is to use the on-chip BRAMs only as cache and use the PSNoC to access the big off-chip DDR memory or HBM memory, which are usually several Gigabytes.

5.8 Conclusions

With a suitable dataflow coordination discipline, combining C-to-gates compilation, soft-core processors, and separate compilation and linking, we can provide a better FPGA development experience—one that is no longer dominated by long wait times for FPGA synthesis-placement-and-routing. This supports a familiar incremental refinement design style, where a functional design is always running and can be improved through a series of small changes and fast edit-compile-debug turns. The key capability we add to enable this

discipline is a producer/consumer-agnostic communication streaming discipline that supports both separate compilation and heterogeneous architectures (e.g., FPGA, Processor) on either side of the communication stream. This allows us to compile C for the operator quickly to processors to give an immediate (< 4 seconds) integration of a refined operator and to compile C to a small partial reconfiguration region quickly (10 minutes) to migrate to the FPGA. Operators mapped to processors can run with the separately-compiled regions already mapped to the FPGA. In an incremental compilation and refinement development style exploiting separate compilation and linking, once an operator has been migrated to the FPGA, it need not be recompiled as other operators are developed and modified. The methodology provides the missing `-O0` and `-O1` compiler optimization options to complement the existing best-effort option provided by monolithic FPGA compilation.

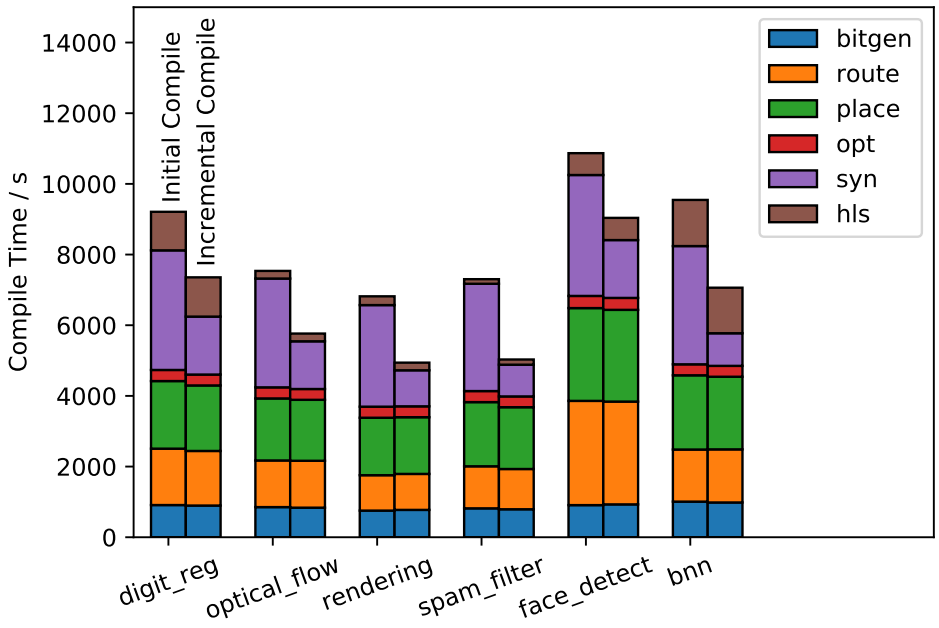
Chapter 6

More Flexibility: Customizable PR Size and Interconnect

Partial Reconfiguration (PR) is a key technique in the application design on modern FPGAs. However, current PR tools heavily rely on the developers to manually conduct PR module definition, floorplanning, and flow control at a very low level. In addition, the existing PR tools do not consider High-Level-Synthesis (HLS) languages either, which is of great interest to software developers. In Chapter 3, we propose PRflow, which is able to accelerate the compilation by a factor of 6.4 – 10.9 ×. However, several limitations still exist: 1) benchmarks cannot map unless the separate decomposed operators are small enough to fit the fixed page sizes; 2) the split operators can only be connected by a BFT NoC with limited bandwidth of 0.8 GB/s, sometimes degrading the overall performance.

In this chapter, We propose HiPR[5], an open-source framework[6], to bridge the gap between HLS and PR. HiPR allows the developer to define partially reconfigurable C/C++ functions, instead of Verilog modules, to accelerate the FPGA incremental compilation and automate the flow from C/C++ to bitstreams. We use a lightweight Simulated Annealing (SA) floorplanner and show that it can produce high-quality PR floorplans an order of magnitude faster than analytic methods. By mapping Rosetta HLS benchmarks, the incremental compilation can be accelerated by 3.5–7.6× compared with standard Xilinx Vitis

[5]The initial version of HiPR has already been published as [121].
[6]https://github.com/icgrp/hipr



`Compile with 200MHz clock constraints for the kernel`
`For the incremental case, we set -vivado.impl.strategies as Flow_Quick`

Figure 6.1: Initial-Compile vs. Incremental-Compile with Vitis

flow without performance loss.

6.1 Design Requirements

Table 6.1: Initial-Compile vs. Incremental-Compile with Vitis (Seconds)

	Initial-compile					incremental-compile					Reduction
	hls	syn	p&r	bit	total	hls	syn	p&r	bit	total	
digit_reg	1091	3385	3823	911	9210	1111	1640	3707	897	7355	20.14%
optical_flow	216	3079	3389	853	7537	219	1350	3358	837	5764	23.52%
rendering	248	2875	2941	754	6818	216	1020	2931	774	4941	27.53%
spam_filter	130	3036	3319	818	7303	147	898	3193	792	5030	31.12%
face_detect	618	3422	5920	909	10869	628	1637	5843	930	9038	16.85%
bnn	1306	3346	3886	1008	9546	1289	923	3866	984	7062	26.02%

To make FPGA more accessible to software developers, vendors have been developing versatile tools, such as Vitis [131], SDSoC [129], and OpenCL [54], to decrease the coding difficulties by supporting high-level languages (C/C++). While these solutions can improve

coding productivity, the most time-consuming place-and-route step does not benefit from these HLS tools. In fact, these long P&R step is not only necessary for the final few iterations but along for every iteration over the entire developing stage. However, during the initial developing stage, users tend to have more quick trials to get evaluations on different optimization strategies. In a sense, developers may need only a small portion of the design to be refined and re-compiled, and it is highly inefficient to re-compile the whole design through bit-level placement and routing. Fig. 6.1 profiles the compilation time breakdown to implement Rosetta benchmarks [146] on a data center card (Alveo U50) [135]. Synthesis usually takes more time for the initial compile (purple blocks) as some peripheral modules are compiled once and can be reused in later incremental compile. However, by changing one source file, we only see 17–32% reduction in the incremental compile times listed in Table 6.1 column 12; 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 recompiled. 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?*

By using PRflow (Chapter 3), the incremental compile can be partially realized by moving the logic to be refined to a page, so that we can perform partial recompile to quickly get some runtime results. However, the limited inter-page bandwidth (0.8 GB/s) can easily be a design bottleneck. Additionally, the fixed page size is not suitable for design space exploration when more area is needed to explore better performance. For example, applying `unroll` or `pipeline` pragma to a design can easily increase resource utilization by several orders. The fixed-page overlay in PRflow cannot fit various resource requirements for different designs.

Based on the factors above, we identify two design requirements.

(1) High-Level Definition of Function to be Refined

Developers should be able to signify the logic to be refined at a higher level (C/C++), which software programmers are familiar with. This high-level support can simplify the design steps so that developers only need to modify the C/C++ code instead of manipulating Hardware Description Languages (HDL), which usually needs expertise in RTL design. `Pragmas/directives` are potentially good solutions to guide the CAD tools to P&R on FPGAs better. For the Xilinx Vitis tool, it only allows the users to signify whether a certain function will be moved to FPGA for implementation but does not let the users control physical implementations detail (e.g. layout shape, location, or reconfigurability). We argue that there should be more connectivity between high-level languages and physical implementation. Specifically, users should be able to signify whether some functions are reconfigurable and even provide shape constraints to guide the CAD tools for later layout implementation. With this support, users can signify which functions are partially-reconfigurable, so that only PR recompilation needs to be performed for later refinement.

(2) Customizable Overlay Automation

To overcome the low bandwidth and fixed-size page limitations in PRflow, the new framework should have the ability to generate different PR overlays for different designs or requirements. For example, if one function is Partially Reconfigurable (PR), the new framework should generate an exclusive pblock with the proper size to fit the corresponding hardware logic. Additionally, the pblock definition should be elastic enough for later refinement. This means it is possible to define a pblock that finally has 10 times LUTs than the initial requirements for later tuning up. The users can provide some parameters which can guide CAD tools to reserve more space for later refinement. When multiple PR functions are needed, the framework should also be able to handle the coarse-grain floorplan for the PR functions and non-PR functions to generate area-efficient implementation under specific timing constraints. All these supports should be automatic with the least manual intervention.

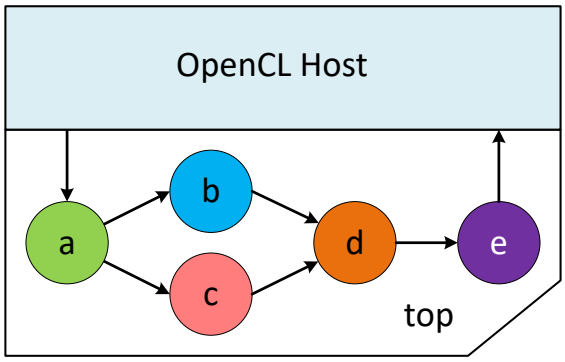


Figure 6.2: Dataflow Graph

6.2 Strategy

To satisfy the above requirements, we propose HiPR, which allows the developers to identify the PR functions by using PR pragmas. HiPR assigns specific PR regions to meet the PR function resource requirements, compiles interconnect and non-changing functions to the static region, and compiles each function to its own PR region. As the interconnect wires can be customized to fit the bandwidth requirement, no performance will be degraded as previous frameworks [123, 122].

6.3 Compute Model: Mapping Input

The dataflow computational graph model [59, 19, 122, 31] has proven effective in isolating kernels for separate compilation. For Kahn Processing Networks (KPN) [59], each kernel, called an *operator*, is described by a C function in HiPR: the operator receives inputs and sends outputs through latency-insensitive streams [17, 1]; reads to empty streams stall until data become available.

The dataflow graph in our model is illustrated in Figure 6.2: 1) the design consists of a cluster of operators; 2) different operators are connected by stream links. The code snippet in Figure 6.3(a) 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 (Figure. 6.3(c) Line 1-4). By calling the `read()`

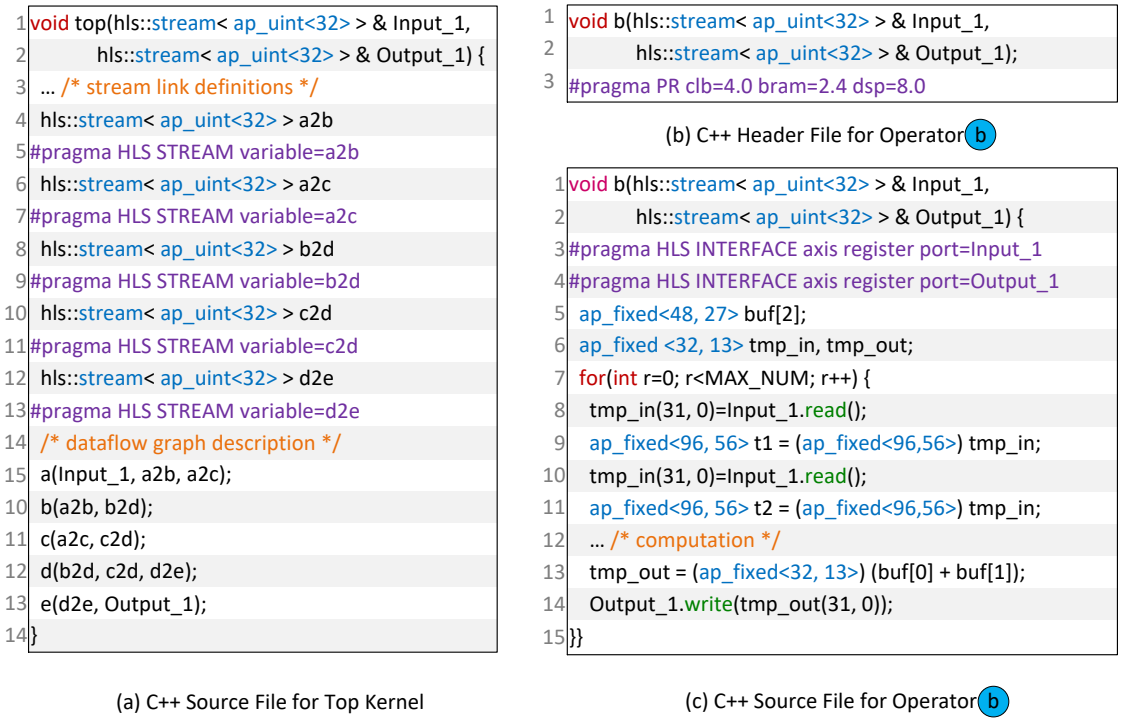


Figure 6.3: Top and Operator C++ Code Prototype

function (Figure. 6.3(c) 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 (Figure. 6.3(c) Line 14).

6.4 Fragmentation

By creating custom pages for each operator, HiPR can avoid some of the fragmentation inherent in the one-size-fits-all, fixed-size pages of prior work [48, 123, 120, 122]. Pages can be sized to include only the resources needed, avoiding the fragmentation that comes from trying to allocate adequate resources to handle a wide variety of operators. For example, operators that do not need DSPs can be given pages with no DSP columns, and pages can be customized for operators that need a large number of DSPs. Furthermore, there is no need to allocate regions exclusively to NoCs that will make some of the LUTS, BRAMs, and DSPs inaccessible to compute pages.

However, the HiPR strategy of allocating a PR region that satisfies Xilinx's partial

reconfiguration constraints exclusively for each operator can still lead to fragmentation. Since vertically-stacked PR regions within one clock region are not allowed, the minimum granularity of resource allocation for the floorplan is one column wide and one clock region height (hereafter referred to as a tile). Consequently, BRAMs must be allocated in blocks of 24 BRAM18s on the UltraScale+ architecture leading to some fragmentation when the number of BRAM18s needed is not a multiple of 24. For example, if we have an operator that needs 25 BRAMs, this can lead to a fragmentation of 48% (23/(2*24)). Similar granularity issues impact LUTS (60 LUT clock-height column quanta) and DSPs (24 clock-height column quanta). Furthermore, since we allocate PR regions as a contiguous set of columns to satisfy the constraints on LUTs, DSPs, and BRAMs for an operator, when the operator's mix of resources does not match the FPGA's local mix of resources, extra columns of the non-limiting resources may be included to get enough resources for the limiting resource.

6.5 High Level Partial Reconfiguration (HiPR)

As shown in Figure 6.1, it takes a different amount of time for initial-compile and incremental-compile. However, incremental compilation can only save 17–32 % compile time than initial-compile. The key reason is that vendor tools have to place and route the entire design monolithically. For HiPR, we also classify the compilations into 2 types: overlay-compile and incremental-compile. Different from PRFlow, which only reuses a one-size-fit-all overlay (Figure 3.4), HiPR generates customized overlays for different applications shown in Figure 6.4. Overlay-compile happens when we first map the design. Different operators are compiled in parallel (High-Level-Synthesis and RTL-Synthesis). Based on the resource report and pragmas/directives (indicating whether an operator is PR or non-PR), a floorplan tool will generate location and PR constraints, which along with the post-synthesis netlists are fed into vendor tools for overlay implementation. Consequently, a customized overlay is generated, which can be re-used for later incremental compilation. The sizes for different pages can vary depending on the initial resource and pragmas. Additionally, different operators can be connected with real physical wires instead of an NoC, the bandwidth is

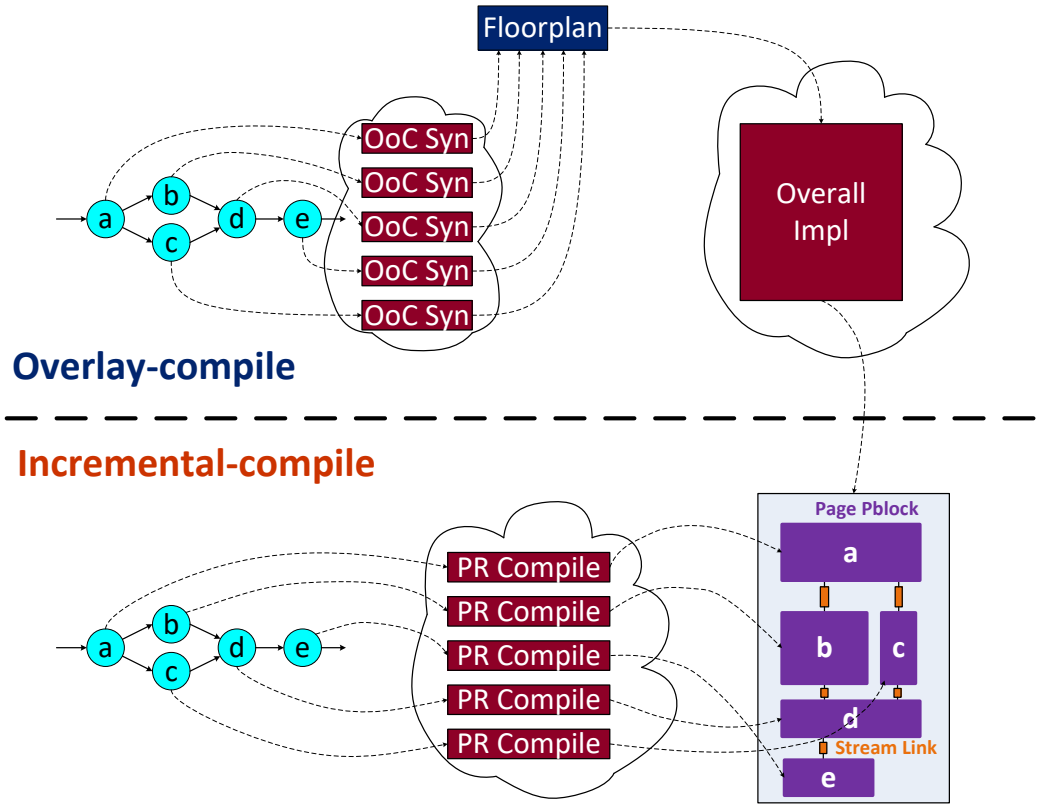


Figure 6.4: HiPR Separate Compile

achieved according to the need.

With the overlay generated before, the divide-and-conquer strategy can be applied to incremental compilation. Separate physical implementations and bitstream generations in addition to HLS and RTL synthesis are performed in parallel. As the initial size of certain pages can be arbitrarily large (limited by the real device size), developers can tune these operators up and perform partial compilation instead of implementing the whole design again. If the size of the refined operators exceeds the size of the specified PR region, an overlay compile needs to be conducted again. Also, as the interconnection between different pages resides on the static region, an overlay compile is needed when changing the interconnections as well.

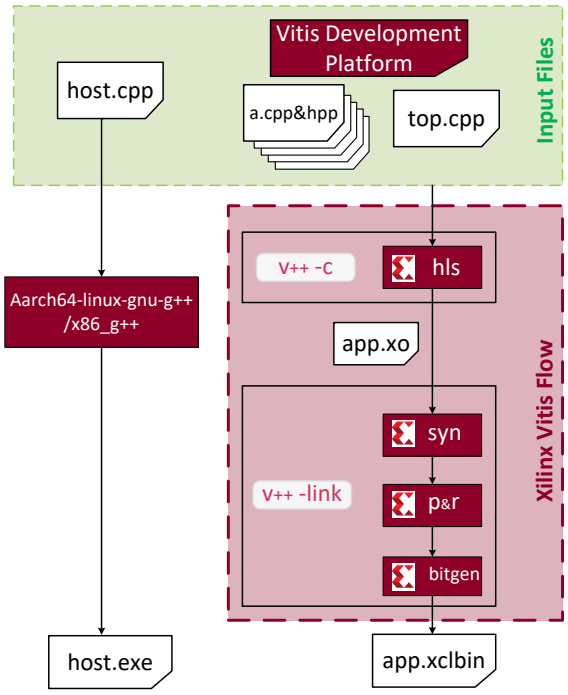


Figure 6.5: Vitis Toolflow

6.6 HiPR Toolflow

In this section, we will describe the implementation of the HiPR toolflow. We first briefly summarize the Xilinx Vitis flow in Figure 6.5 as a normal example and list several drawbacks of current vendor tools. Then we will elaborate on the HiPR toolflow in Figure 6.6.

6.6.1 Xilinx Compilation Flow

The basic inputs into Xilinx toolflow include a device deception kit, which specifies the low-level device information, C/C++ source files for hardware kernels, and OpenCL source files for host drivers.

Taking in all the C/C++ files as the inputs, `vitis_hls` is called to generate `app.xo` file. By executing the command `v++ -link`, the Xilinx objective file `app.xo` can be compiled to an FPGA-loadable file `app.xclbin`. This step is equivalent to the combination of the traditional compilation steps (RTL synthesis, technology mapping, placement, routing, and bitstream generation), which is also the most time-consuming part. For the incremental

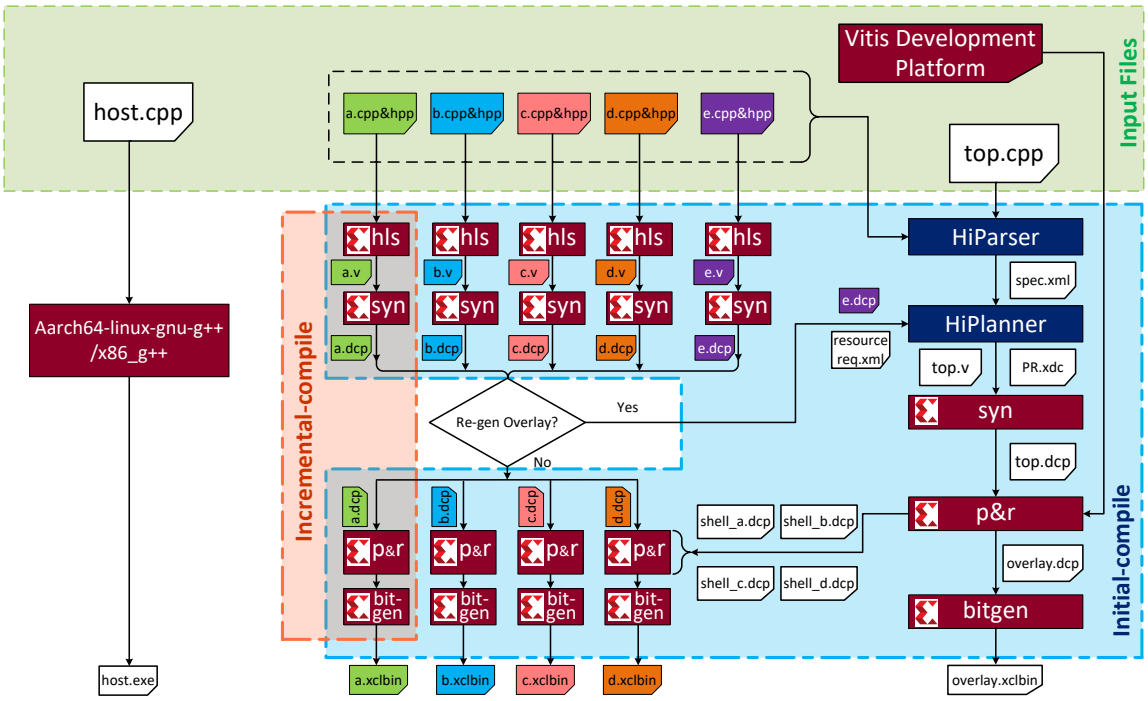


Figure 6.6: HiPR Toolflow

compile, even if only a small portion of the hardware source file is modified, it still needs to go through hls, syn, p&r, and bitgen to generate the final FPGA-loadable file. As we showed before in Figure 6.1, incremental compile only saves some synthesis time but keeps the p&r, bitgen almost the same. Unfortunately, this linkage step is not open for application developers. Therefore, it is hard to perform incremental compile with the PR technique.

6.6.2 HiPR Compilation Flow

For HiPR, it takes the same input source as Xilinx Vitis: each operator is represented by a C++ function; the PR pragma in the corresponding header file signifies whether the function is partially reconfigurable (Fig. 6.3(b) Line 3). For the example in Figure 6.6, 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).

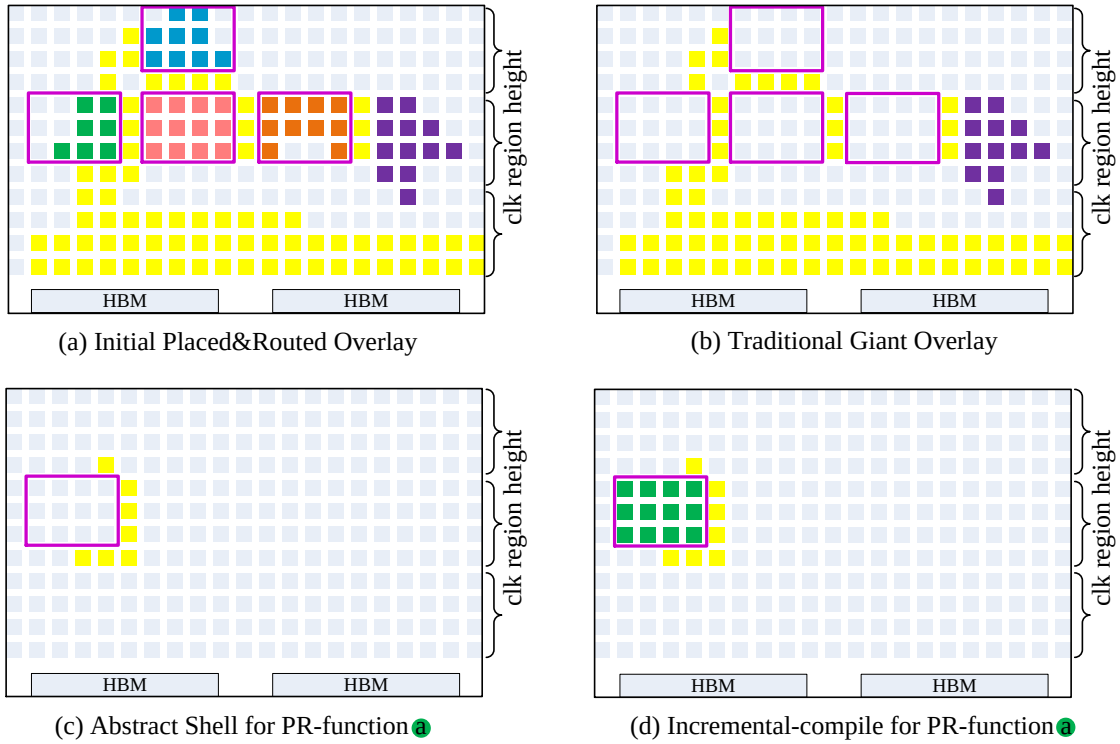
As described in Chapter 6.5, we classify the development compilation into 2 types: overlay-compile and incremental-compile. For the overlay compile, shown in the blue dashed

block in Figure 6.6, a Python-based module `HiParser` parses the `top.cpp` file and interprets the connections between different operators. The hardware header files for corresponding operators are also parsed by `HiParse` module to identify all the PR functions and all the parsed information is stored in an XML file, which will be fed into another Python-based module `HiPlanner` for floorplanning.

Concurrently, HiPR calls `vitis_hls` and `vivado` for independent compilations for the separate operators in parallel. The post-synthesis resource utilization of all the operators is collected by `HiPlanner` for floorplanning. A Simulated Annealing algorithm is adopted to floorplan the operators, and the generated `PR.xdc`, which defines the partial reconfigurable regions and partially reconfigurable modules, is then fed into `vivado` to generate a partially reconfigurable overlay. Finally, an `overlay.xclbin` is generated, which corresponds to the post-routed device layout in Figure 6.7(a). For traditional PR flow without abstract shell [134], a giant overlay (Fig. 6.7(b)), which contains the definition for all PR regions and the static logic, is generated. All the information will be loaded even when only one PR region needs re-implementation. This step can last 10-20 minutes for Alveo data-center FPGAs. This will lengthen the entire implementation as well since more elements are considered during the placement and routing steps.

With the abstract shell technique, independent abstract shell `DCP` files are generated for PR functions to perform the in-context implementation. In this example, 4 abstract shell `DCP` files are generated for the 4 PR functions (a, b, c, d). Figure 6.7(c) shows the abstract shell for PR-function a, where only the connection wires and logic (yellow blocks) related to that PR region are reserved. As a result, the post-synthesis netlists for corresponding 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 [124], the separate `xclbin` files for operators a, b, c, d, can be loaded after the `overlay.xclbin` are loaded for the final application execution.

The header files for the operators are important here as they signify whether the functions/operators are partially reconfigurable. To accommodate the logic increase for the



(a) Initial Placed&Routed Overlay

(b) Traditional Giant Overlay

(c) Abstract Shell for PR-function a

(d) Incremental-compile for PR-function a

Figure 6.7: Overlay-compile vs. Incremental-compile

later refinement on the operators, we provide resource ratio pragma to guide `HiPlanner` to assign extra resources than the operator initially needs. For example, in Fig. 6.3(b) 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 can help reserve enough space to accommodate design growth, as the developer can change functionality, add code to fix bugs, and increase parallelism. An application-specific overlay will finally be generated.

For incremental compilation, developers can refine the PR functions with quick compile shown in the dashed orange block in Figure 6.6: only function a is modified, and this function is recompiled by calling `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 Figure 6.7(d).

Based on these dependencies, we write a makefile [42], which only launches necessary compilations for the modified source files. Therefore, HiPR can run on a single machine,

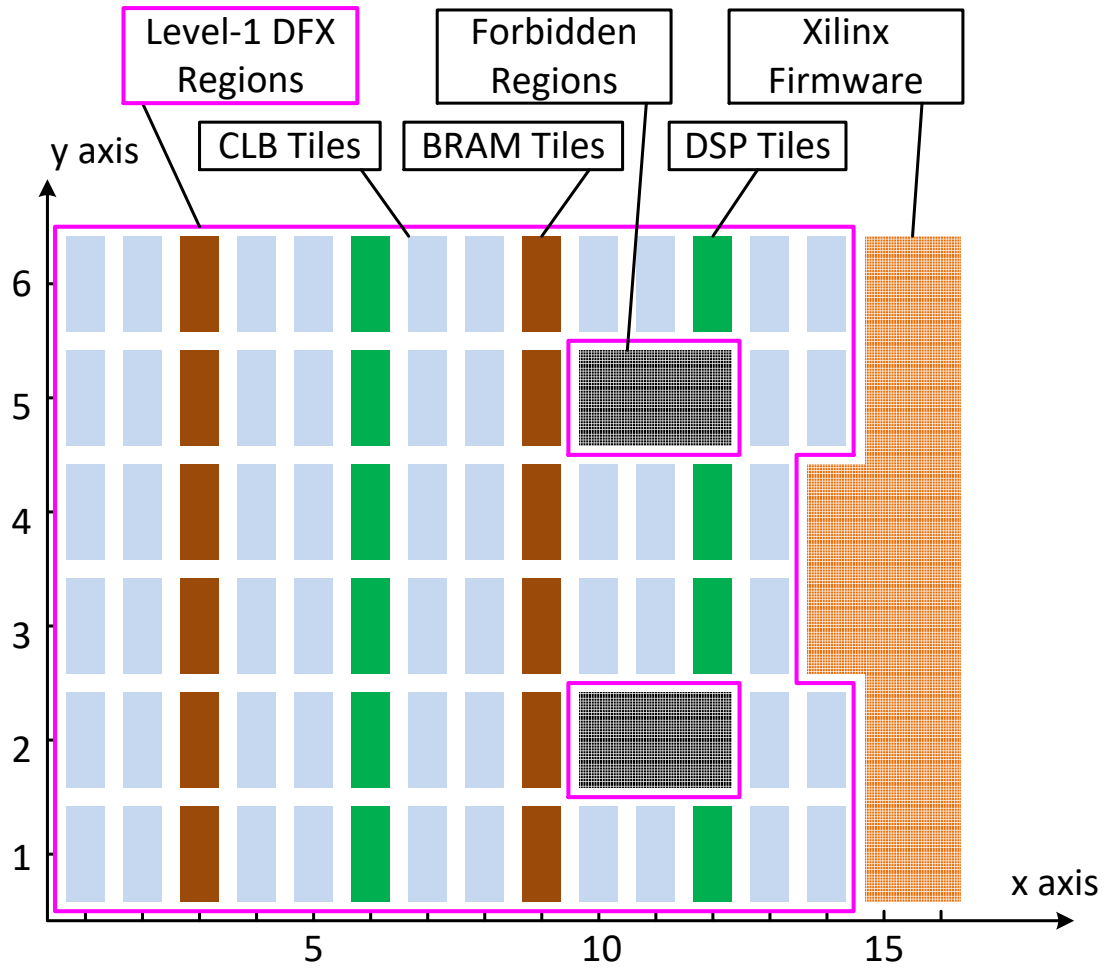


Figure 6.8: Data-center FPGA Device Architecture

on which the parallelism depends on the local cores and memory size.

The independent compile strategy also enables compilations by a cluster of servers on the cloud, while vendor tools can only run a single machine. We deploy Sun Grid Engine [94] for task scheduling. HiPR generates proper scripts according to the dependencies and submits the compilation jobs with `qsub`. 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 overlay-compile. Changing the streaming links between the operators also lead to re-launching overlay-compile as well, since it affects the interconnect

wires in static regions.

In summary, HiPR can launch two types of compilations based on needs. Initially, HiPR launches overlay-compile according to the pragmas specified by users to generate an application-customized overlay with PR regions defined for corresponding PR functions. For later incremental compilation, users can refine the PR functions with quick PR compilation. When the current overlay cannot fit the size or bandwidth requirements, users can launch the overlay-compile again with new pragmas.

6.7 HiPlanner

The C++-based floorplan module (`HiPlanner` in Figure 6.6) is the key step to bridge HLS and physical PR implementations. Various approaches have been proposed for floorplanning [111, 100, 8, 105]. We adopt the traditional Simulated Annealing (SA) as our floorplan engine since it runs faster than analytical methods [100, 105]. We also implemented the MILP-based floorplanner according to [105] for detailed comparisons in Chapter 6.9.1.

6.7.1 Problem Formulation

Modern data-center FPGA devices can be described by Cartesian integer coordinates, as shown in Figure 6.8, where the heterogeneous resource (i.e., CLB, DSPs, BRAMs, ...) are usually distributed non-uniformly over the chip. Additionally, Vendors often reserve some area for firmware implementation (shell logic in Alveo U50) and define a Level-1 DFX region for the users (Custome Logic). The basic element of the floorplan is one column wide and one clock region height (hereafter referred to as a tile). Vertically-stacked PR regions within one clock region are not supported by Xilinx FPGAs.

`HiPlanner` takes in the resource requirements from `HiParse` and a device description file, and produces a set of PR constraints (PR.xdc), which describes the size and location of the PR regions. The constraint file (PR.xdc) is fed into `vivado` along with the logic post-synthesis netlists to generate an overlay.

We model the FPGA device as a 2-dimension matrix, which contains columns of 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, URAM, 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;

$y :=$ lowermost row coordinate for a tile;

$w :=$ width of a PR region in units of tiles; w is set to no less than 4 for routability;

$h :=$ 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;

$f :=$ an area that cannot be used by PR regions ($f \in F$), such as $< 10, 2, 3, 1 >$ and $< 10, 5, 3, 1 >$ in Figure 6.8;

$r_t :=$ 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). We assume only one module drives DMA input and DMA output only drives another module;

GAP := number of columns between two PR regions when both occupy the same row.

We simplify the FGPA device based on the columnar architecture of modern FPGAs by using a W-element *resource vector* <CLB, CLB, BRAM, BRAM, ... CLB, CLB> to represent the resource distribution over one row. *number of rows* represents how many rows of resources the chip has, which is also equivalent to the number of clock regions vertically. Finally, invalid regions are represented by a set of 4-element vectors. The detail is listed in Table 6.2. `HiPlanner` first reads in the simplified device file and then processes the floorplan for all the operators with Simulated Annealing (SA) algorithm.

Table 6.2: Parameters to Describe a Device

Parameters	Description
resource vector	A vector of resource types to represent the resource distribution over one row. For the device in Figure 6.8, the resource vector is <CLB, CLB, BRAM, CLB, CLB, DSP, CLB, CLB, BRAM, CLB, CLB, DSP, CLB, CLB, CLB, CLB>.
number of rows	It represents how many rows of resources are on the chip. For the device in Figure 6.8, the number of rows is 6.
invalid area	It represents a set of invalid areas, including forbidden and firmware regions, represented by a 4-element vector. For the device in Figure 6.8, it has 4 invalid areas: <10, 2, 3, 1>, <10, 5, 3, 1>, <14, 3, 1, 2>, <15, 0, 2, 6>.

The goal of the `HiPlanner` is to find a set of non-overlapping areas $a_j :< x_j, y_j, w_j, h_j > | \ j \in \{0, .., |PR| - 1\}$ to map all the PR functions $pr_i \in PR| \ i \in \{0, .., |PR| - 1\}$.

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

$$xc_{a_i} = x_{a_i} + w_{a_i}/2 \tag{6.1}$$

$$yc_{a_i} = y_{a_i} + h_{a_i}/2 \tag{6.2}$$

The Manhattan Distance is adopted to represent the wire length between 2 areas:

$$Mdist_{a_i,a_j} = |xc_{a_i} - xc_{a_j}| + |yc_{a_i} - yc_{a_j}| \tag{6.3}$$

6.7.2 Objective Function

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

$$min:\ \alpha * WL_{norm} + \beta * RW_{norm} + \gamma * OV_{norm} \quad (\alpha, \beta \geq 0,\ \gamma \geq 1,\ \alpha + \beta = 1) \tag{6.4}$$

where α and β are weights for total wire length and resource wastage, respectively and γ is the weight for overlapping PR regions; the sum of α and β is 1; WL_{norm} is the normalized wire length; RW_{norm} is the normalized resource wastage; OV_{norm} is the normalized overlapping area in units of tiles.

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

$$WL_{abs} = \sum_{pr_i, pr_j \in PR | i<j} Mdist_{a_i(pr_i), a_j(pr_j)} \cdot l_{pr_i, pr_j} + \sum_{pr_i \in PR} Mdist_{dma, a_i(pr_i)} \cdot l_{dma} \tag{6.5}$$

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:

$$WL_{norm} = \frac{WL_{abs}}{(|L| + 2) \cdot \max\{l_{pr_i, pr_j} | l_{pr_i, pr_j} \in L, l_{dma}\} \cdot (W + H)} \tag{6.6}$$

where $|L| + 2$ represents the total link number plus one DMA input and one DMA output; $max\{l_{pr_i, pr_j} | l_{pr_i, pr_j} \in L, l_{dma}\}$ 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:

$$RW_{norm} = \frac{1}{|PR| \cdot |T|} \sum_{i \in \{0,..|PR|-1\}} \sum_{t \in T} \frac{r_{a_i,t} - r_{pr_i,t}}{r_{chip,t}} \tag{6.7}$$

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.

The normalized overlapping area is calculated as:

$$OV_{norm} = \begin{cases} 0 & \text{if } \forall\, U_{x,y} \leq 1; \\ 1 + \frac{1}{|PR| \cdot H \cdot W} \sum_{x \in \{0,..W-1\}} \sum_{y \in \{0,..H-1\}} U_{x,y} & \text{otherwise.} \end{cases} \tag{6.8}$$

where $U_{x,y}$ is equal to the number of PR functions that use $tile_{x,y}$. By Equation 6.8, the overlapped area term is normalized to (1, 2].

The sum of α and β is 1 and, γ is no less than 1. The floorplan is only legal when the cost function is less than 1, as any overlapping areas will increase the overlapping term to more than 1. Higher γ encourages our method to generate a legal floorplan more quickly.

6.7.3 Greedy PR Shape Generation

We use a greedy method to reshape the region to cover 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 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. Consequently, If $y + h$ reaches H, we set x and y all to 1 and start the previous greedy step again. This can provide access to the whole chip resources.

Since the FPGA fabric is non-homogeneous, when we move a region from one x location

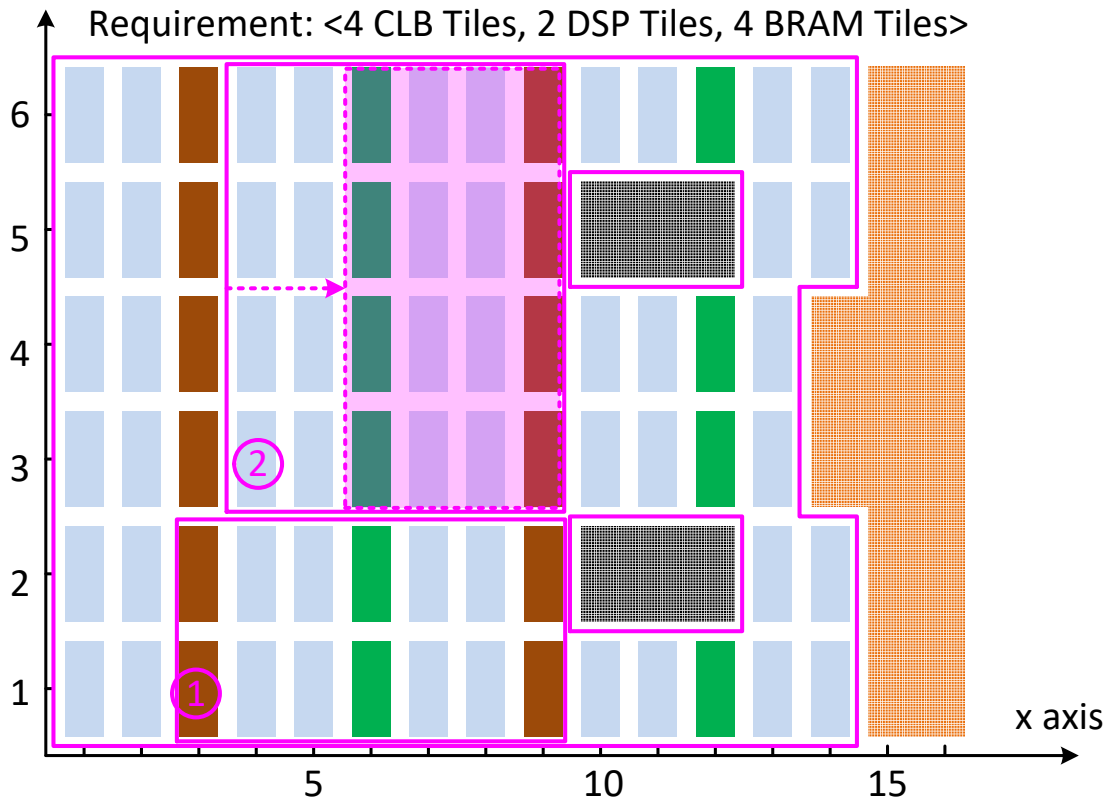


Figure 6.9: PR Region Reshape on a Given Left-bottom Point

to another, the existing w, and h may not provide the needed resources. For example, in Figure 6.9, assumes we need 4 tiles of CLBs, 2 tiles of DSPs, and 4 tiles of BRAMs. If our left-bottom tile is $< 3, 1 >$, our final shape will be $< 3, 1, 7, 2 >$. If the start tile is $< 4, 3 >$, intuitively the shape will be $< 4, 3, 6, 4 >$. Compared with the previous shape, we waste 8 tiles of CLBs, 2 tiles of DSPs adding to the fragmentation. To avoid extra resource wastage on the left boundary region, after the resource requirements are met, we will increase the x coordinate of the shape as long as the resource requirements can be met. By applying this greedy reshaping strategy, the final shape is shown as the shaded area $< 6, 3, 4, 4 >$ in Figure 6.9.

Note that we need to obey more practical constraints as well. For example, the minimum width for a PR region is 4-tiles. Also, for each PR region, we intentionally include 3 (GAP)

more columns on the right and leave these 3 (GAP) columns in the static region. This extra space is reserved to route the wires between different PR regions.

With this greedy reshaping method, we only need to use Simulated Annealing to determine the lower-left coordinates for each operator.

6.7.4 XY Simulated Annealing (XYSA)

As the main goal of our Simulated Annealing algorithm is to generate proper x and y coordinates for all the operators, we call our algorithm XYSA. As shown in Algorithm 3, the inputs to `HiPlanner` are device parameters and resource requirements for all the operators. For the initial point, we randomly generate the x and y coordinates for all the operators and perform the greedy reshaping method to generate the PR shapes (Chapter 6.7.3). We update the cost function and use this initial cost as the current cost. For the following simulated annealing steps, 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. After the shape of the operator is determined, we update the cost function with the new set of PR regions. We accepted the results if the cost function is improved or the calculated accepting possibility is greater than the random possibility. For our implementation, x and y coordinates are the simulated annealing targets since we believe this representation is simple and fast to run. By using the variables we define in Chapter 6.7.1, our implementation can easily be extended to support sequence pair [89, 111], which is another traditional representation in floorplan and placement. We will compare our results with the sequence-pair-based Simulated Annealing algorithm (SQSA) and Mixed-Integer Linear Program (MILP) in Chapter 6.8.3 and Chapter 6.8.4, respectively.

6.8 Design Metrics

In this section, we will profile the characterizations of XYSA algorithm first. Then, we will compare our XYSA algorithm with other algorithms, such as Simulated Annealing

Algorithm 3 Simulated Annealing Floorplanner (HiPlanner)

```
1: procedure HiPlanner(Device Parameters, Set of Resource Requirements for Operators)
2:     for operator in operators_set do
3:         Randomly generate lower-left < x, y > for operator
4:         Reshape(operator, < x, y >)
5:     end for
6:     T ← T_0
7:     CurrCost ← CostFunction()
8:     MinCost ← CurrCost
9:     while T > T_MIN do
10:        i ← 0
11:        while i < TRIAL_NUM × log 1/η do
12:            Randomly select an operator_j
13:            Randomly generate < x_j, y_j > for operator_j
14:            Reshape(operator_j, < x_j, y_j >)
15:            Move operator_j to < x_j, y_j >
16:            df = CostFunction() - CurrCost
17:            if df < 0 then
18:                CurrCost ← CostFunction()
19:                if CurrCost < MinCost then
20:                    CostMin ← CurrCost
21:                    Best Set of Operator Shape ← Current Set of Operator Shape
22:                end if
23:            else
24:                if exp(-df/T) > random_possibility then
25:                    CurrCost ← CostFunction()
26:                else
27:                    Reset operator_j to previous location
28:                end if
29:            end if
30:            i ← i + 1
31:        end while
32:        T ← η×T
33:    end while
34:    if MinCost < 1 then
35:        return the Best Set of Operator Shape
36:    else
37:        return Fail
38:    end if
39: end procedure
```

with Sequence Pair(SQSA) and Mixed-Integer Linear Programming method (MILP). As not all the implementations are open-source, we implement different algorithms in C++ with similar variable definitions in chapter 6.7.1.

6.8.1 Benchmark Preparation

To characterize the XYSA algorithm, we use `digit recognition` (varying BRAM utilization) from the Rosetta Benchmark Suite [146], as it is easy to tune the application resource usage up and down.

Digit Recognition

`Digit recognition` is based on K-Nearest-Neighbour (KNN) algorithm. A subset of MNIST database [33] was downsampled to N (N=18,000 in the original Rosetta Benchmark) training samples and 2000 test samples and stored as 196-bit unsigned integers per image. N 196-bit images are stored on-chip by BRAMs and, for each test image, a Hamming distance is calculated for each training sample. K training samples with the smallest Hamming distances are voted to decide the final result. For our case, we use the full database [33] of 196-bit images. We split the training set into a systolic/cascaded chain of operators. The first operator calculates the best K candidates for the input testing sample and passes the K best candidates along with the input test sample to its downstream operator. All the operators in the middle vote to choose K best candidates from their local candidates and the upstream K best candidates, and pass the best K candidates along with the testing sample to the next stage. The final operator calculates the best candidate. All the operators are running at a task-level pipelining or dataflow manner shown in Figure 6.10. By changing the parameters below, we can generate different benchmark versions with various BRAM utilization ratios of Alveo U50 in Table 6.3. In fact, a full database [33] only provides N=60,000 training samples. We can still set N=83,200 in Table 6.3 since we can instantiate more BRAMs when more training samples are ready later. Since we can only assign BRAMs to the PR function in units of tiles (24 BRAM18s per column), we can barely increase the utilization above 80%, especially when the page number is high. This is because

Table 6.3: Digit Recognition Resource Utilization

case	num	p	par	BRAM Utilization
1	1280	20	1	17%
2	1280	20	2	18%
3	2560	20	1	31%
4	4480	20	1	46%
5	4480	20	2	61%
6	8320	20	1	76%
7	8320	20	2	90%

the datawidth for the training set is 196 bits, and the granularity of BRAMs increments for each page is 11 BRAM18s ($\lceil \frac{196}{18} \rceil$). If the total BRAM numbers for each page cannot be divided by 24, some fragmentation issues will show up.

`P` : the number of decomposed pages;

`PAR` : the partition number of the training set within a page;

`N` : the total samples of the training set;

6.8.2 XYSA Characterization

Below lists the parameters for XY Simulated Annealing (XYSA) algorithm. In this section, we will use `digit recognition` case 5 in Table 6.3 to show how these parameters affect the quality of results (QoR) of XYSA.

T_0 : the initial temperature for Simulated Annealing;

`TRIAL_NUM` : the number of trials before the temperature is decreased by 10 $\times$;

`T_MIN` : the frozen temperature;

η : the temperature decay ratio;

Initial Temperature – Figure 6.11 shows the cost function with different initial temperatures for XYSA. Figure 6.11(a) shows the cost functions for all temperatures converge

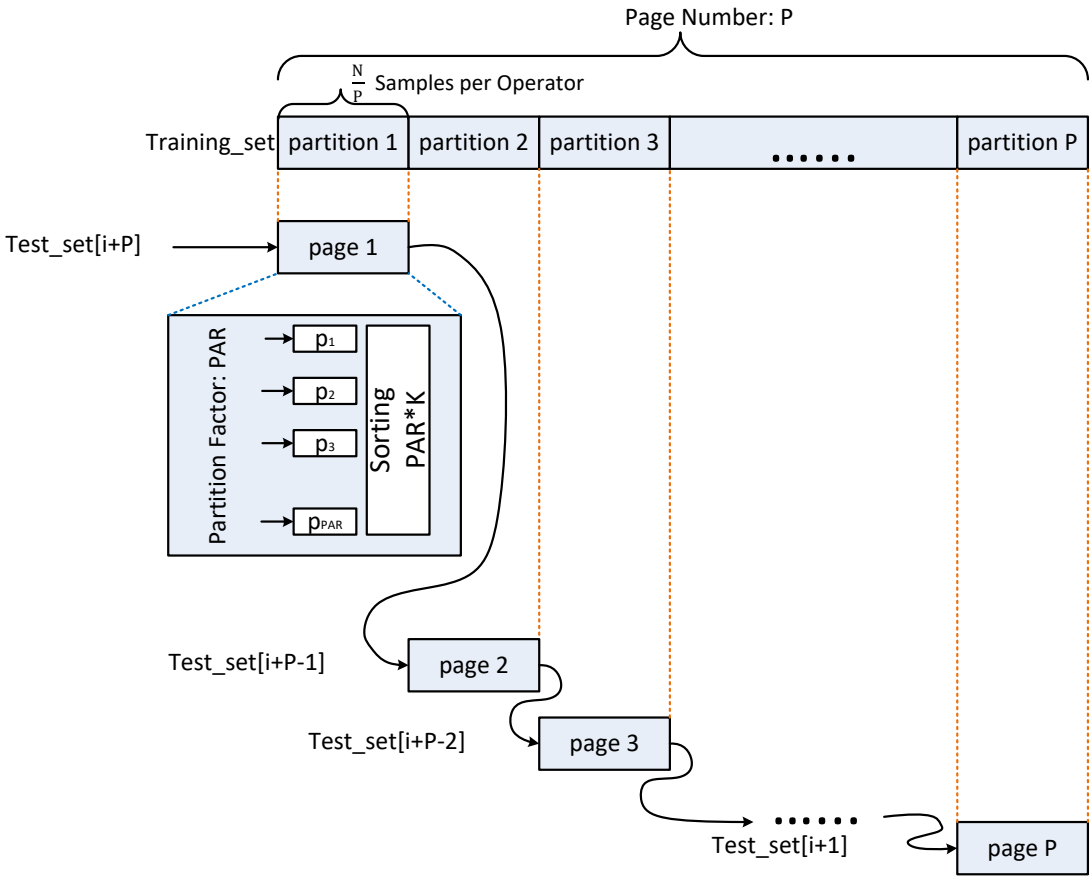
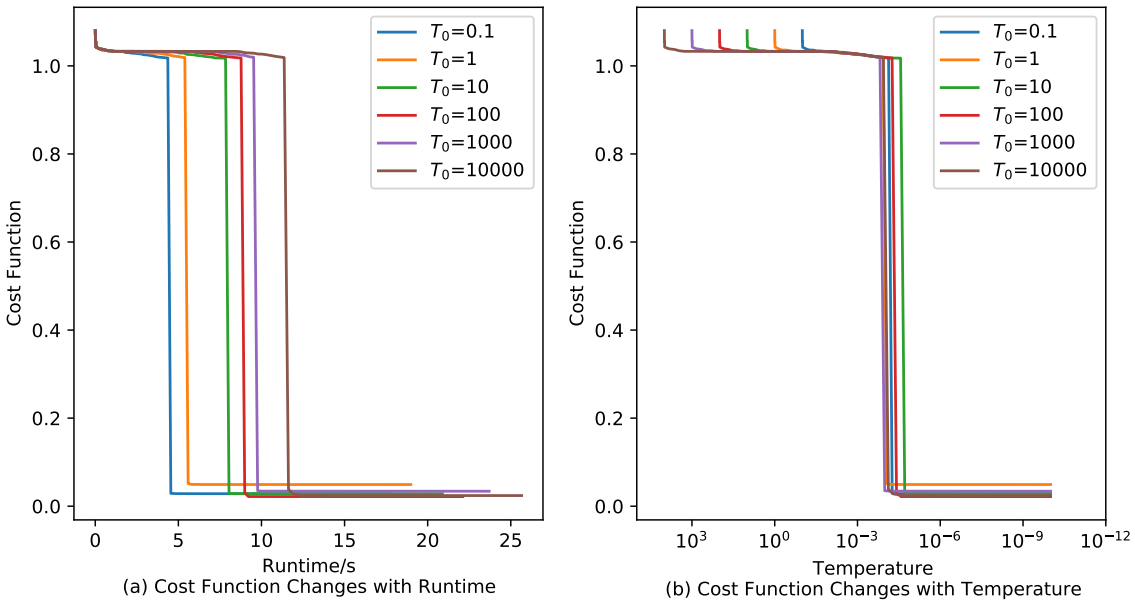


Figure 6.10: Digit Recognition Benchmark Decomposition Automation

after running for 15 seconds. However, different initial temperatures can generate different final cost functions. For Figure 6.11(b), we can see cost functions converge slowly in the temperature range [1000, 1e-3], but converge quickly below 1e-3. This means the lower temperature can accelerate the convergence for XYSA algorithm. Nevertheless, we believe a higher initial temperature is useful to avoid being trapped in a local optimum. From Figure 6.12, we can see when the initial temperature is below 1, XYSA may fail to find legal floorplan results. Therefore, to guarantee that we can finally generate legal floorplan results, we will set the initial temperature to 100 for the following experiments.

TRIAL_NUM – We sweep the `TRIAL_NUM`, the interval before the temperature is decreased



BRAM Utilization=61%, N=44,800, P=20, PAR=2
TRIAL_NUM = 100000, T_MIN=1e-10, η=0.9997

Figure 6.11: XYSA Cost Function with Different Initial Temperature

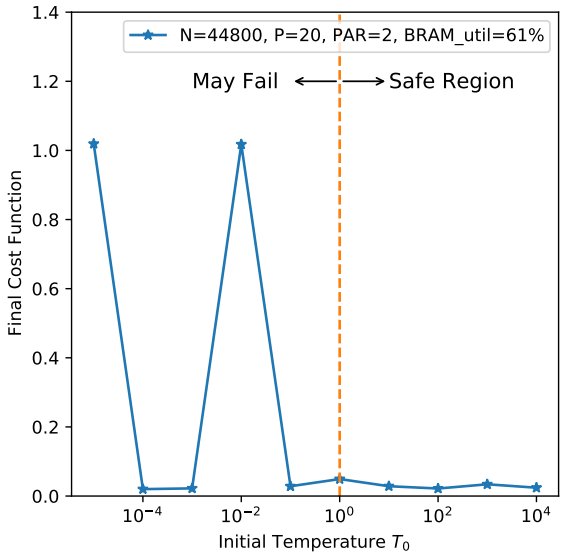
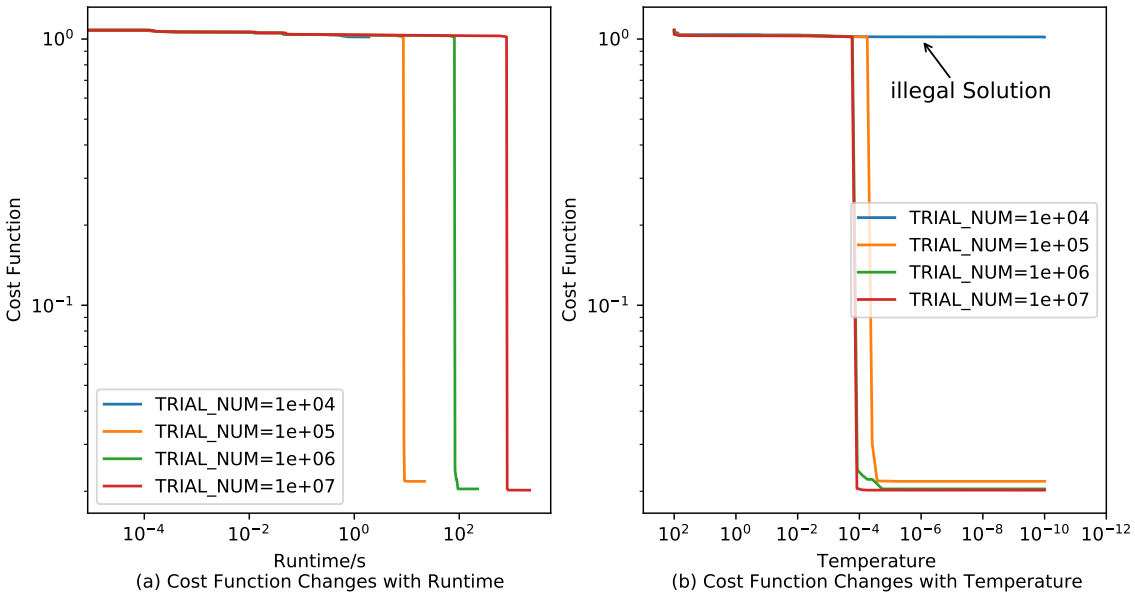


Figure 6.12: Final Cost Function with Different Initial Temperature – when the initial temperature is less than 1, XYSA may fail to find a feasible solution

by 10 × (per log-scale unit). From Figure 6.13(b), we see the cost function decreases faster with temperature when we have more trials in each temperature range. When TRIAL_NUM



BRAM Utilization=61%, N=44,800, P=20, PAR=2
T_0 = 100, T_MIN=1e-10, η=0.9997

Figure 6.13: XYSA Cost Function with Different Initial Temperature

is 1e+7, we see the cost function converges most quickly. However, when we look at the cost function versus runtime shown in Figure 6.13(a), we see the cost function converges more slowly when TRIAL_NUM is higher. This is because it takes more time at each temperature with a higher TRIAL_NUM. It is worth noting that too low TRIAL_NUM ($<$ 1e+4) may make the floorplanner fail to find a legal solution, as shown in Figure 6.13(b). By plotting the cost function with the TRIAL_NUM in Figure 6.14, we see the higher TRIAL_NUM could slightly improve the final cost function. However, we see no significant improvement when TRIAL_NUM is bigger than 1e+6. Therefore, we prefer to use trial numbers of 1e+5 to keep the floorplan process fast with a considerable low-cost function. In summary, we use T_0=100, and TRIAL_NUM=1e+5 for further experiments.

6.8.3 Sequence-Pair Simulated Annealing

Sequence Pair is a classic representation for floorplanning [89]: a positive sequence (Γ_+) and a negative sequence (Γ_-) are adopted to represent the relative locations between each pair

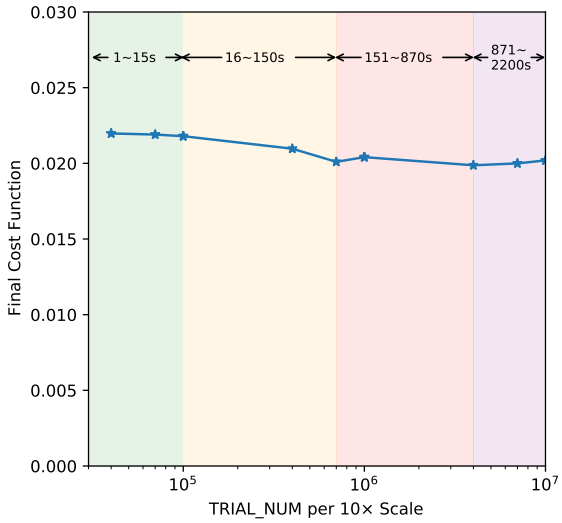


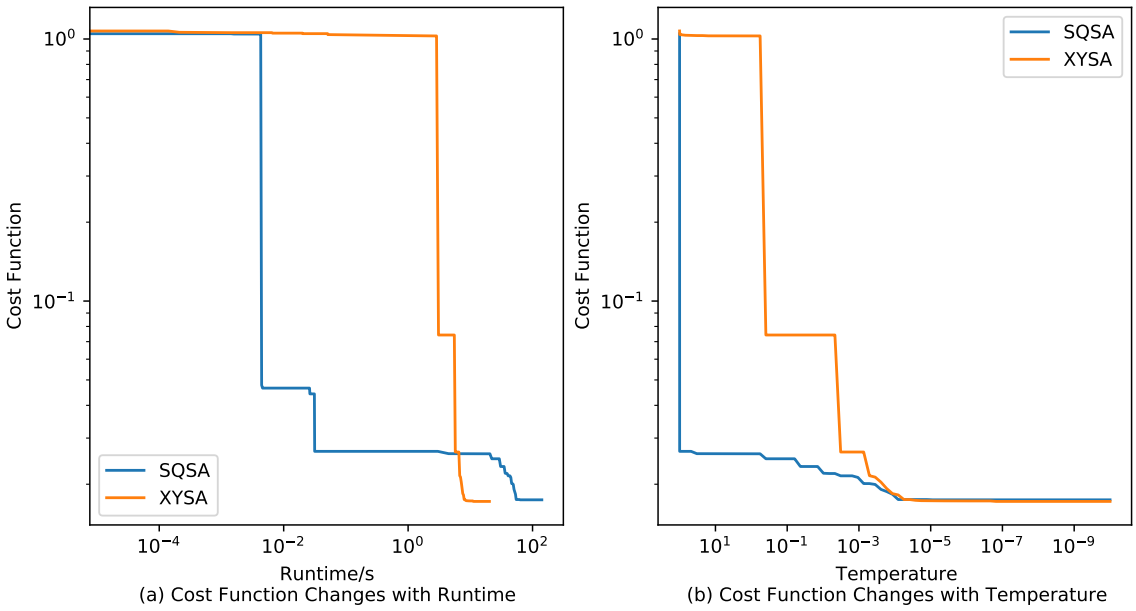
Figure 6.14: Final Cost Function with Different TRIAL_NUM – regions with different colors represent different runtime; e.g., it takes 1–15 seconds when `TRIAL_NUM` is less than 1e+5

of blocks. If block a is before block b in both Γ_+ and Γ_-, block a must be left of block b, shown in Equation 6.9. If block a is before block b in Γ_+ and after block b in Γ_-, a must be above b. Based on the relative locations for all the blocks, a directed and vertex-weighted graph called a horizontal-constraint graph (G_H) and a vertical-constraint graph (G_V) can be constructed. The weight of G_H and G_V can be viewed as the width and height of a block. By using the well-known *longest path algorithm* for vertex-weighted, directed acyclic graphs, the absolute location coordinates can be calculated for all the blocks. Readers can refer to [89] for a detailed explanation.

$$(\Gamma_+ :< ..., a, ..., b... >, \Gamma_- :< ..., a, ..., b... >) \Rightarrow a \text{ is left of } b \tag{6.9}$$

$$(\Gamma_+ :< ..., a, ..., b... >, \Gamma_- :< ..., b, ..., a... >) \Rightarrow a \text{ is above } b \tag{6.10}$$

We implemented the Sequence-Pair Simulated Annealing (SQSA) from [8] but extended it to support PR constraints and forbidden areas for modern FPGAs. We first use the *longest path algorithm* to determine the weights of G_H. For the left-most block, we use a



BRAM Utilization=61%, N=44,800, P=20, PAR=2
T_0 = 100, T_MIN=1e-10, η=0.9997

Figure 6.15: Comparison between XYSA and SQSA

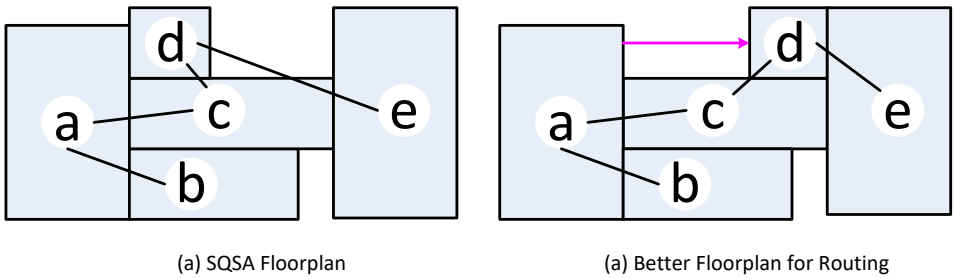


Figure 6.16: Routing Driven Floorplan

similar greedy reshaping strategy in Chapter 6.7.3 to determine the width and height of the block. The width (w) and height (h) of the following blocks are determined by the x and w of the blocks on their left. As the heights of all the blocks are determined after updating G_H, the weights of G_V are determined, by which the y coordinates of all the blocks can be easily calculated. These relative location representations can generate a more compact outlined floorplan, as the right and upper blocks are always adjacent (which is not necessarily true with XYSA) and determined by the left and lower blocks. However, this

strategy might be inadequate to represent the rich space in optimizing inter-page link length. In Figure 6.16, we can see different implementations represented by the same sequence pair ($\Gamma_+ :< a, d, c, b, e >, \Gamma_- :< a, b, c, d, e >$). We assume d and e have heavy linking wires. If we use *longest path algorithm*, we will get Figure 6.16(a) where block d is adjacent to block a. However, if we move d in the right direction by certain tiles, d and e have a shorter distance. This move cannot be represented by sequence pair but can be represented by XYSA.

Figure 6.15 shows the difference between XYSA and SQSA algorithms. We see SQSA converges more quickly than XYSA initially, as SQSA tends to generate compact floorplans. Since SQSA has a smaller design space (e.g., SQSA can not represent the floorplan in Figure 6.16(b)), XYSA will slightly outperform SQSA when the temperature is lower than 10e-5 shown in Figure 6.15. In terms of runtime, since it takes more time for SQSA to update the floorplan and the cost function, we can see XYSA converges faster than SQSA in Figure 6.15(a).

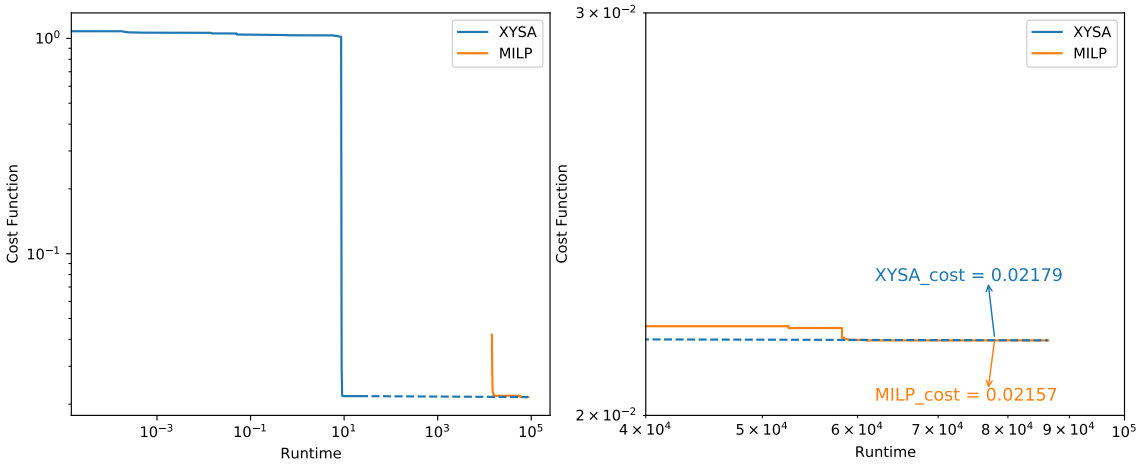
XYSA can generate cost functions with a negligible difference in a much shorter time than SQSA. In Chapter 6.8.5, we will show that XYSA and SQSA generate similar quality of results.

6.8.4 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) is another classic method in floorplanning. We implement MILP model from FLORA [105] in C++ prototype and use Gurobi 9.5.1 [49] for academia as our solver. We extend the original MILP to support modern devices in Chapter 6.7.1. We define the extra variables and constraints below. Readers can refer to [105] for the other constraints.

$a_{i,y} :=$ binary variables set to 1 if and only if PR_i occupies row y;

$g_{i,j} :=$ binary variables set to 1 if and only if PR_i is not to the left of PR_j.



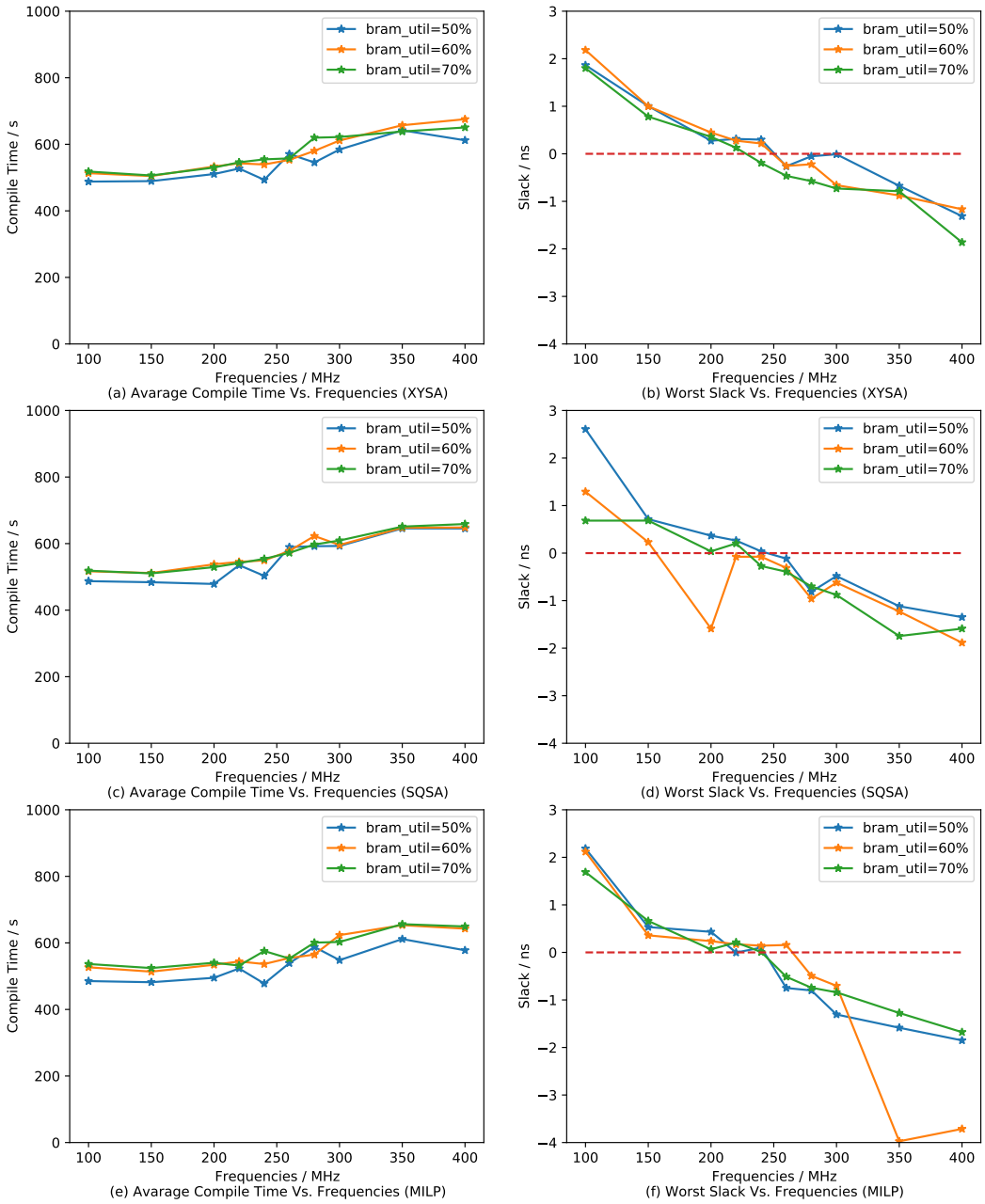
BRAM Utilization=61%, N=44,800, P=20, PAR=2
T_0 = 100, T_MIN=1e-10, η=0.9997

Figure 6.17: Runtime Comparison between XYSA and MILP

$$\forall\ PR_i,\ PR_j \mid i < j,\ i, j \in \{0, ...|PR| - 1\}$$
$$x_i + w_i + GAP \leq x_j + g_{i,j} \cdot W \tag{6.11}$$

$$\forall\ PR_i,\ PR_j \mid i < j,\ i, j \in \{0, ...|PR| - 1\}$$
$$x_i \geq x_j + w_j + GAP - (3 - g_{i,j} - a_{i,y} - a_{j,y}) \cdot W \tag{6.12}$$

where GAP represents the extra space in units of tiles between 2 PR regions when both occupy the same clock row (same as XYSA in Chapter 6.7.1). Since it usually takes significant time for MILP to reach optimal or even a feasible result, we set the maximum runtime for the Gurobi solver as 24 hours and keep the other parameters as default. We use a dashed horizontal line to extend the line after XYSA converges in Figure 6.17 to compare the cost function between XYSA and MILP. We can see XYSA can converge an order of magnitude faster than MILP. However, MILP will eventually outperform XYSA by 1 % at cost of hours runtime. We will show the improvement has a negligible impact on the quality of results in Chapter 6.8.5.



XYSA,SQSA: TRIAL_NUM=1e5, T_0=100, T_MIN=1e-10, η=0.9997
MILP: T_MAX=86000s

Figure 6.18: Compile Time and Timing Slack Vs. BRAM Utilization

6.8.5 Quality of Results

BRAM Utilization vs. Compile Time – Figures 6.18(a) (c) (e) show how BRAM utilization affects the HiPR compiles time with different floorplan algorithms. For the

compile time, we use the average compile time from all 20 pages. We see the compile time is mainly driven by the clock frequencies since it takes more time to meet more strict timing constraints. The average compile time with different floorplanners is similar. Therefore, we believe XYSA can outperform MILP by an order of magnitude shorter execution time with a similar page FPGA compile time. XYSA and SQSA also have similar compile times, but XYSA can get a slightly smaller cost function with a faster execution time.

Table 6.4: Compile Time and Timing Slack with BRAM Utilization

BRAM Util	50%		60%		70%	
	Cost	Fmax / MHz	Cost	Fmax / MHz	Cost	Fmax / MHz
XYSA	0.01857	240	0.02124	240	0.03249	220
SQSA	0.01922	240	0.02144	150	0.03265	220
MILP	0.01842	240	0.01983	250	0.02426	220

BRAM Utilization vs. Timing Slack – Figures 6.18(b) (d) (f) show how BRAM utilization affects the HiPR timing slack. We see the timing slack is related more to frequency constraints than the BRAM utilization when the utilization is under 80%. When BRAM utilization is 80%, HiPR fails to generate overlays. We see XYSA, SQSA, and MILP have similar max clock frequencies in Table 6.4. Based on the facts above, the maximum frequency HiPR with XYSA can achieve is around 200 MHz to 250 MHz.

6.9 Experimental Evaluations

We evaluate the compile time acceleration of our framework by implementing the realistic Rosetta HLS benchmarks [146] on the Alveo U50 data center card [135] 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 (705,520 LUTs, 2,232 18Kb BRAMs and 4,920 DSPs). HiPR uses Xilinx Vitis 2022.1, including associated `Vivado` and `Vitis_HLS` and `XRT` as the backend. We perform the compilation on a cluster of 8 servers. Each server is equipped with two 2.7 GHz Intel E5-2680 CPUs and 128 GB of RAM (total of 8×2×8=128 cores).

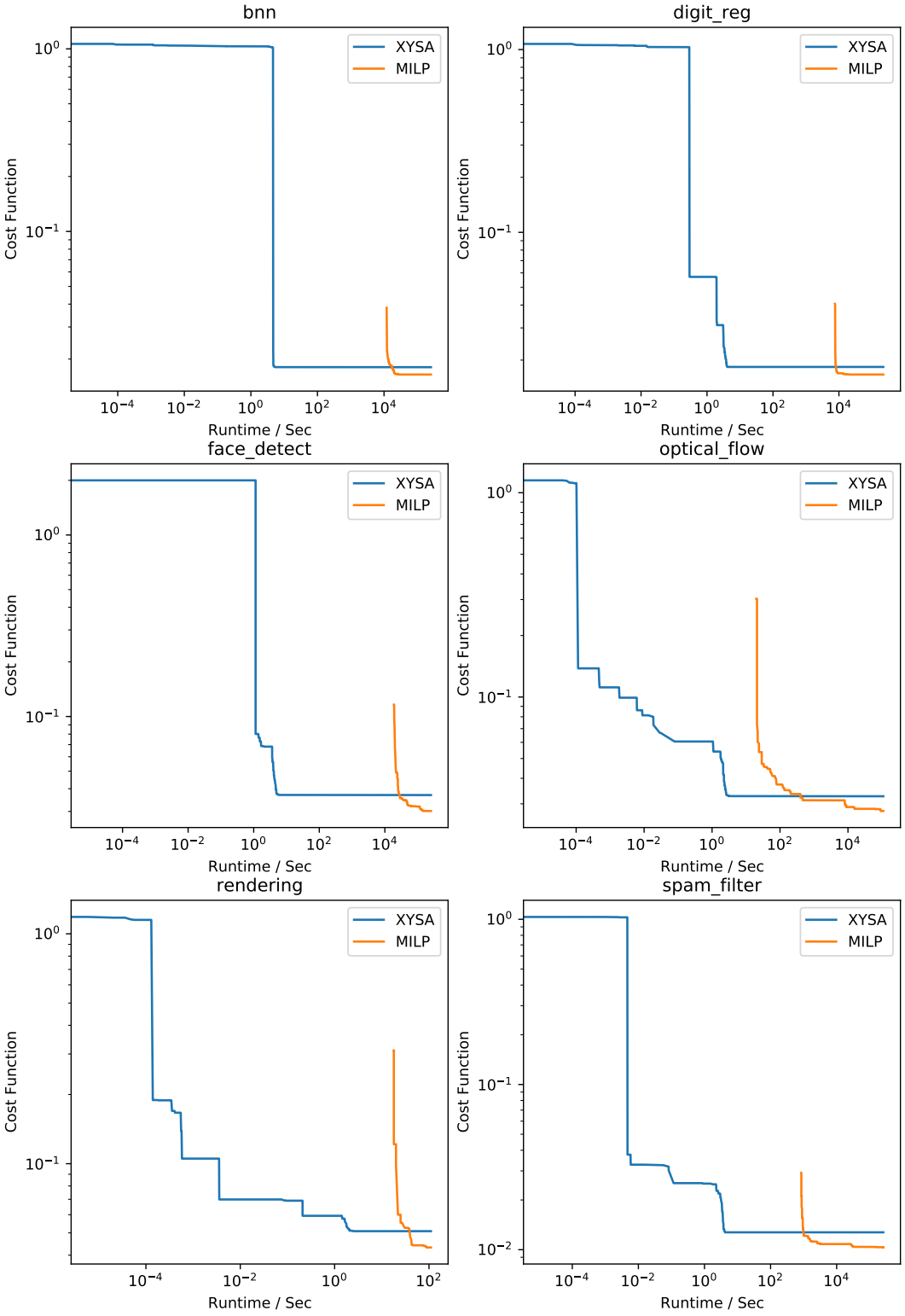


Figure 6.19: Floorplan Execution Time Comparison between XYSA and MILP

Table 6.5: Floorplan Runtime (in seconds)

Bench -mark	PR #	LUT	B18	DSP	Runtime (Seconds)			Cost Function			
					XYSA	MILP *	MILP Optimal	XYSA	MILP final	MILP BestBd	Impro-vement
3d render	6	5742	88	9	7	40	110	0.051	0.043	0.043	27%
Digit Recog	20	48875	359	1	10	8246	-	0.018	0.016	0.013	11%
Spam Filter	15	18703	75	256	10	1364	-	0.012	0.01	0.006	17%
Optical Flow	8	19432	143	286	7	397	109601	0.032	0.027	0.027	16%
Face Detect	20	137758	268	141	13	28646	-	0.036	0.03	0.006	17%
Binary NN	22	40546	1102	4	11	17110	-	0.018	0.016	0.009	11%

6.9.1 Floorplanner

Table 6.5 shows the comparison between `HiPlanner` (XYSA) and the state-of-the-art floorplanner (MILP). The proposed SA-based floorplanner is implemented in a C++ prototype (Chapter 6.7) and is compared to our implementation (Chapter 6.8.4) of the MILP floorplanner [105] that already showed better results than [100] and [99]. However, since [105] is only based on the Virtex-7 series and did not consider the hierarchical DFX features, we enhanced it to support these features mentioned in Chapter 6.7.1.

From Table 6.5, we can see it takes less than 15 seconds (column 6) for XYSA to converge to good results (column 9, XYSA Cost Function). For the MILP method, the runtime in column 7 (MILP *) means the MILP methods reach the same results as XYSA. We can see it takes more than 2 hours for MILP to generate similar results as XYSA when page numbers are 15-22. Column 8 lists the runtime when MILP reaches the optimum or the maximum runtime we set (48 hours). Only `3d-rendering` and `Optical flow` can reach optimality in 110 and 109,601 seconds. Column 10 lists the best results the MILP method can get within 48 hours. Figure 6.19 shows how the cost function changes with runtimes

between XYSA and MILP methods. Column 11 lists the predicted best bound MILP can possibly reach (not achieved unless optical). It takes much more time for MILP to generate similar results to XYSA when page numbers increase. Column 12 lists the improvements by MILP over XYSA (11-27%).

Table 6.6: Rosetta Benchmarks Incremental-Compile Times (seconds)

	Vitis Flow with 32 Threads					HiPR with 8 Threads					
	hls	syn	p&r	bit	total	hls	syn	p&r	bit	total	Speedup
digit_reg	1111	1640	3707	897	7355	42	94	867	98	969	7.6
optical_flow	219	1350	3358	837	5764	17	59	781	97	823	7.0
rendering	216	1020	2931	774	4941	77	79	802	112	940	5.3
spam_filter	147	898	3193	792	5030	17	58	737	97	781	6.4
bnn	1289	923	3866	984	7062	227	628	370	58	1230	5.7
face_detect	628	1637	5843	930	9038	67	241	2267	70	2590	3.5

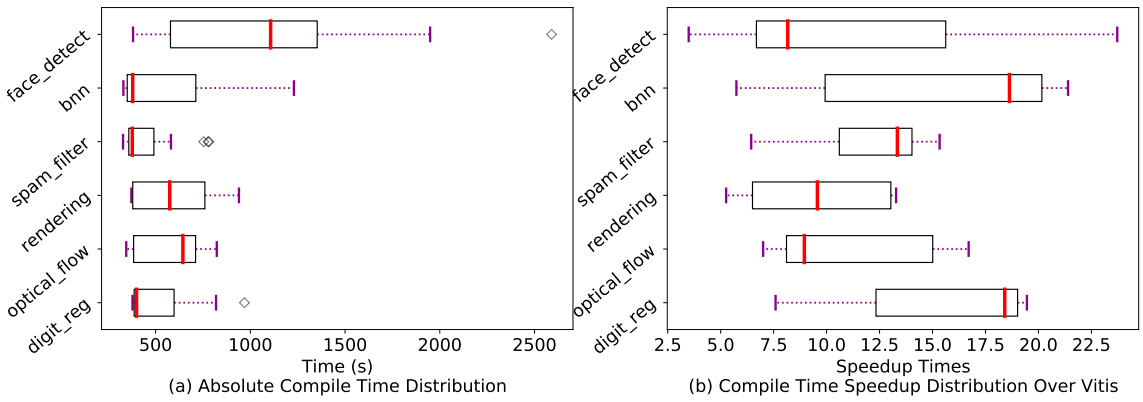


Figure 6.20: Operators Mapping Time Distribution

6.9.2 Compilation Time and Performance

Incremental-Compile: The main goal of HiPR is to accelerate the incremental compilation since only the modified functions need to be recompiled. Figure 6.20 shows the compilation distribution for different operators over the full benchmark sets. The operators can be incrementally recompiled in 5-43 minutes. For all the benchmarks, the median values are near 7–18 minutes. This means that in most cases, users can benefit from short

Table 6.7: Rosetta Benchmarks Overlay-Compile Times (seconds)

Benchmark	Vitis Flow with 32 Threads					HiPR with 32 Threads					
	hls	syn	p&r	bit	total	syn	p&r	abs	max_op	total	overhead
digit_reg	1091	3385	3823	911	9210	1107	6029	1378	833	9347	1.5%
optical_flow	216	3079	3389	853	7537	1092	9005	1229	747	12073	60.2%
rendering	248	2875	2941	754	6818	1238	4264	1279	784	7565	11.0%
spam_filter	130	3036	3319	818	7303	1101	16905	1270	706	19982	173.6%
bnn	1306	3346	3886	1008	9546	1166	24237	1394	375	27172	184.6%
face_detect	618	3422	5920	909	10869	1102	9389	1339	2282	14112	29.8%

† `Maximum compile time for all the operators minus the hls and syn time.`
§ `The overhead is calculated by dividing the total time difference between HiPR and Vitis over the Vitis time.`

incremental compilation to tune their target functions more efficiently. We can see that incremental compilation can be improved by a factor of 3–23 (Figure 6.20(b)). Figure 6.21 shows the compilation time breakdown for all the benchmarks. We can see the place-and-route time is accelerated most. Table 6.6 shows the detailed compilation time. For HiPR, we choose the maximum compile time from all functions for each benchmark as the final compilation time. Even with the worst case, HiPR can still outperform Vitis by 3.5–7.6×.

Table 6.8: Performance Comparison: Vitis vs. HiPR

Benchmark	Vitis Flow		HiPR Flow	
	Freq (MHz)	Runtime	Freq (MHz)	Runtime
Digit Recognition	250	2.3 us	250	2.6 us
Optical Flow	200	2.4 ms	200	2.2 ms
Rendering	200	1.5 ms	200	1.4 ms
Spam Filter	200	16.8 ms	200	19.1 ms
BNN	150	5.1 ms	150	4.8 ms
Face Detect	200	19.1 ms	200	23.0 ms

Overlay-Compile: When a benchmark is compiled 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 Figure 6.22, we can see HiPR takes more time to generate the overlay for all the benchmarks,

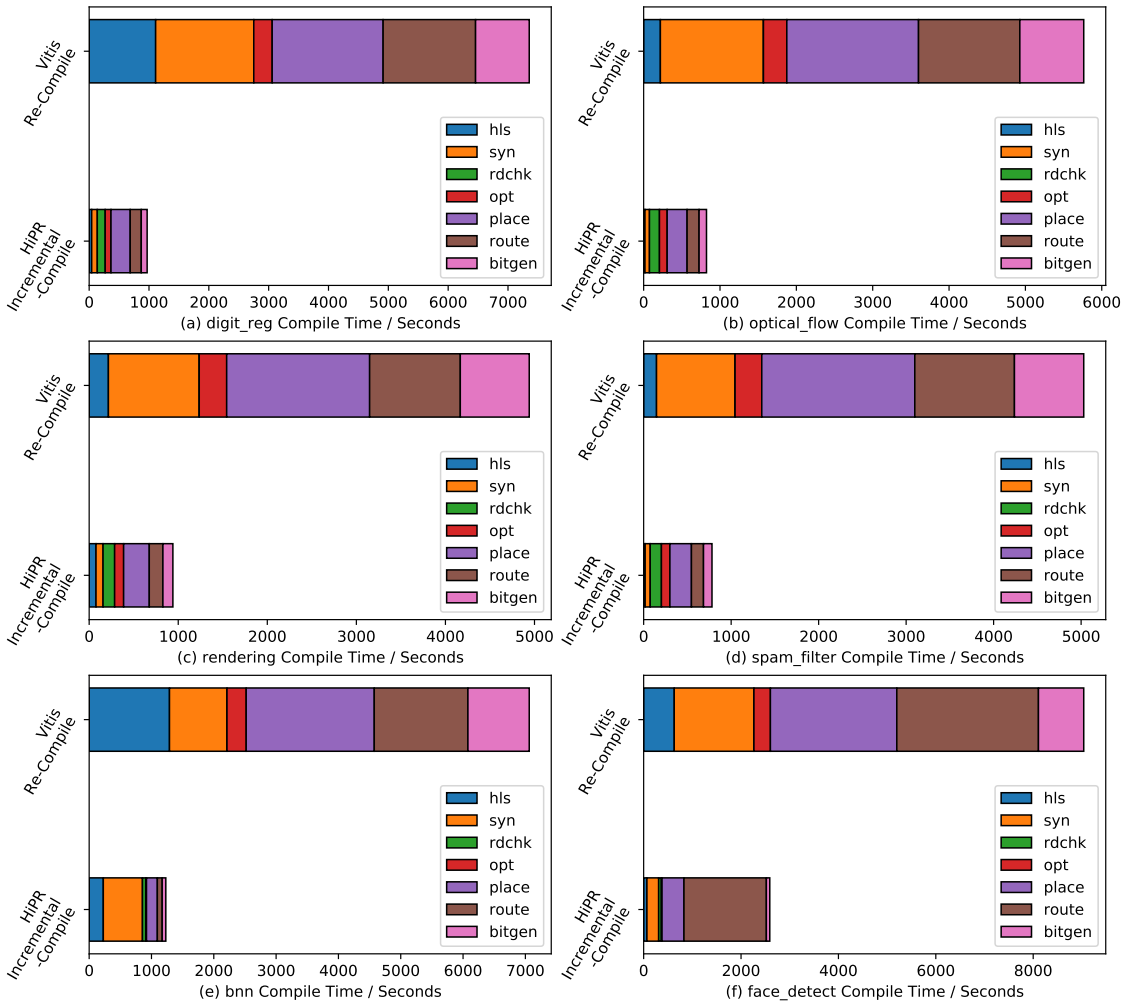


Figure 6.21: Incremental-Compile Breakdown

as the operators have to be placed and routed along with overlay generation. In Table 6.7 column 9, we choose the maximum value between overlay bitstream generation and abstract shell generation, as they can be conducted simultaneously. It takes 1.5–184.6% overhead in compile time to set the overlays up. However, this process is usually performed once, and users can benefit from incremental compilations afterward.

Performance Comparison: Table 6.8 summarizes the performance between Vitis and HiPR. As we rewrite the original code in latency-insensitive style (Chapter 6.3), the throughput is slightly different from Vitis implementation performance. However, HiPR achieves

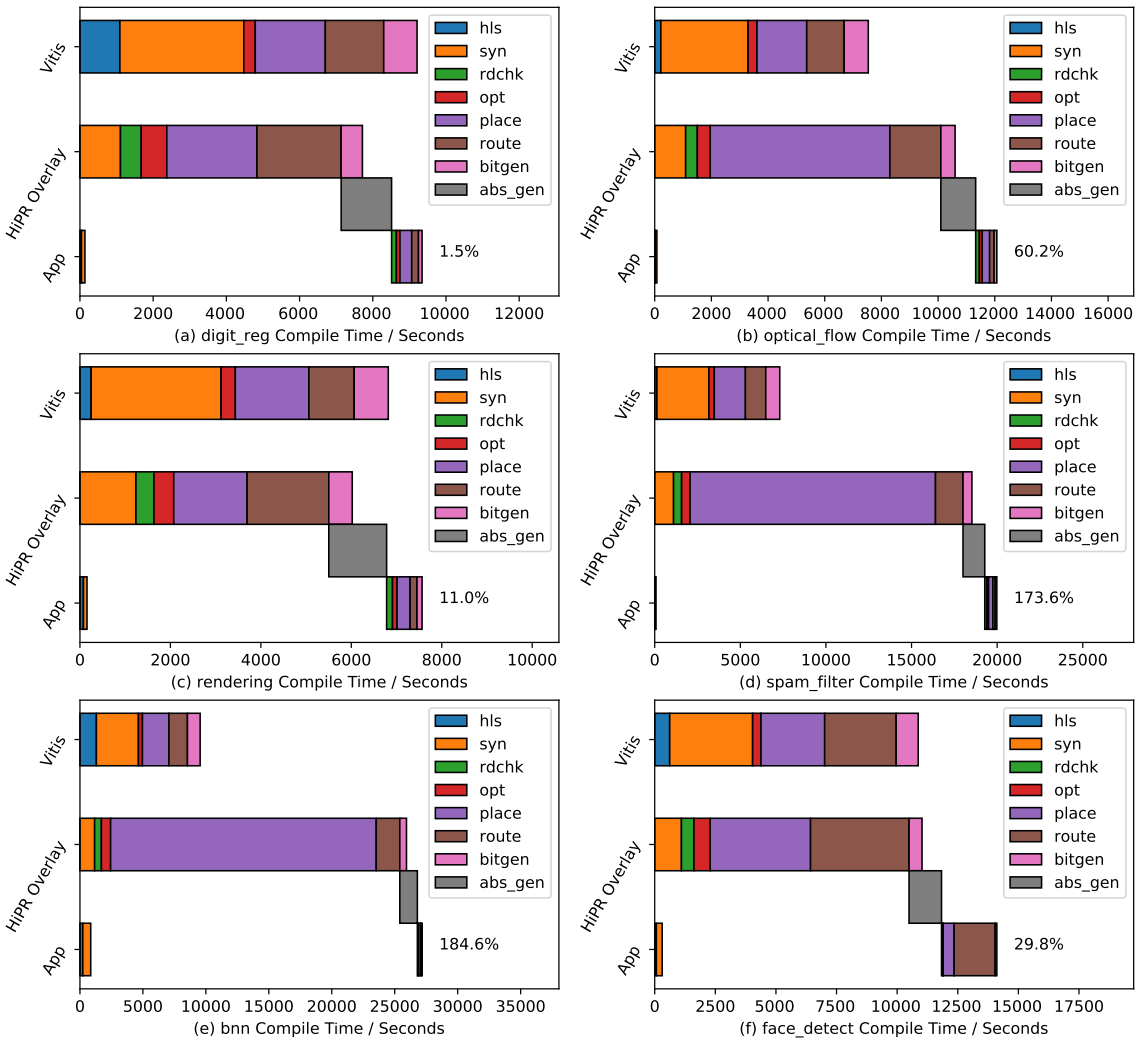


Figure 6.22: Overlay-Compile Breakdown

the same frequency and performance as the original Vitis Flow. We also implement floorplan generated by MILP and get a similar compile time and timing slack as XYSA.

Compared with PRflow in Chapter 3, HiPR can customize the overlay according to applications. This can address two significant limitations – the fixed-page size issue and the inter-page bandwidth issue. As HiPR can assign various PR blocks to PR operators/functions according to resource requirements, there is no need for the developers to decompose their design to fit the fixed page size in PRflow. Also, HiPR allocates dedicated links between operators and uses relay stations [1, 19] to meet the inter-page bandwidth

instead of potentially being limited by the uniform low NoC bandwidth. Therefore, HiPR does not degrade communication bandwidth, while delivering a similar short incremental compile time as PRflow. Since it takes more time to map larger operators, developers have the freedom to decompose their designs to accelerate compilation.

6.10 Conclusions

In this paper, we propose HiPR, a framework that allows users to define partially 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 a long time for full recompilation. The experiments from Rosetta Benchmark implementation show that HiPR can decrease the incremental-compilation time by a factor of 3.5–7.6× without performance loss or need to target fixed PR region sizes.

Chapter 7

Prior Related Work and Our Future Work

7.1 Prior Closely-Related Work

For the FPGA compilation time, many methods and frameworks have been proposed. The recent Xilinx-released open-source work RapidWright [73] is an excellent framework that provides APIs to manipulate the low-level implementation (P&R). Except for bitstream generation, RapidWright can also support divide-and-conquer compilation. Based on RapidWright, some compilation time acceleration methods are proposed, such as RapidStream [48], which can both accelerate the compilation time and boost the design frequency. Our work differs from [73, 48] as we aim at separate compilation from HLS to bitstreams, while [73, 48] need to stitch separate post-place&route netlists together and generate one bitstream for a full device, which may limit the speedup, especially for larger devices.

To improve the FPGA coding experience, Cascade [104] and SYNERGY [70] are proposed. They allow the application to run immediately in the form of simulations and hide the compilation behind the execution. When hardware compilation is done, the application can be executed by FPGA fabrics. Users only see the application is running faster without knowing the details behind the framework. This method is similar to the transition from `-O0` (chapter 5 to `-O1` (Chapter 3) by our work. However, our work support C/C++ inputs, while Cascade and SYNERGY only support Verilog/VHDL inputs. We believe extending Cascade and SYNERGY to support HLS C/C++ inputs would be a great complementary part of our work to improve the FPGA programming experience.

To bridge High-Level-Synthesis (HLS) to low-level implementation and bitstreams, we see AutoBridge [47], AutoSA [114], RapidStream [48], etc. AutoBridge couples coarse-grain floorplan with pipelining during the HLS stage. More layout-level information is provided for HLS tools to generate RTL code with better timing performance. HiPR (Chapter 6) also proposes to relate HLS code with low-level implementation. But HiPR aims to allow users to define partially reconfigurable functions at the C/C++ level and automate the process from HLS, PR modules floorplan, and partial bitstream generations, which can accelerate the incremental compilation time for PR functions. Even though AutoBridge does not consider PR, we believe some coarse-grain floorplan strategies from AutoBridge may be borrowed to improve HiPR timing, which we leave as one of our future works. AutoSA can relieve users from manual interference to automatically transmit HLS code to systolic array architectures for FPGA implementation. Although not all applications are suitable for systolic array implementation, AutoSA still provides a partial solution to one of our future works (chapter 7.5) for auto-decomposition. The generated systolic arrays can be mapped as pages in our PRflow to accelerate the compilation time.

7.2 What are solved and unsolved by this work?

This dissertation mainly presents different frameworks to implement divide-and-conquer compilation strategy for FPGAs. PRflow (Chapter 3), DW (Chapter 4), and HiPR (Chapter 6) can map pre-separated C/C++ applications to FPGAs in minutes, providing short debug-edit-compile loops for users to get quick returns for design development. Softcore (Chapter 5) allows users to map applications to a cluster of pre-compiled on-chip RISC-V cores and enables on-chip debugging by supporting hardware `prinf` feature. PRflow and DW support 200 MHz constraints and fixed-size page mapping, while HiPR supports various frequencies and page sizes according to applications. All these demonstrate the first steps toward full FPGA support for software developers, including separate compilation strategies, quick linkage, and versatile performance-and-compile-time trade-offs.

Nonetheless, we note that compiling FPGA applications in a software manner is chal-

lenging due to intrinsic spatial features. The major limitations and potential future work include: 1) PR overhead of resource wastage and compile time; 2) soft cores performance and memory shortage; 3) C/C++ auto-decomposition; 4) leverage the hard NoC on new devices 5) scalability for larger devices.

7.3 PR Overhead of Resource Wastage and Compilation Time

In order to perform separate compilations, we leverage Partial Reconfiguration (PR) by vendors, which is a mature off-the-shelf technique. However, PR itself has several limitations. First, the implementations for partial reconfigurable regions must be performed in-context, since static logic can use the routing resource in PR regions. It is hard for users to precisely control how much routing resource can be utilized by static logic. If the routing resource for certain scarce IPs (DSPs or BRAMs) is occupied by the static logic, those IPs are blocked and unusable for later PR mapping. To fully resolve this problem, it is beneficial to have a method to prevent static logic from using resources in PR regions. Previous works OpenPR [107] propose to use a route blocker that uses all the routing resources in the PR region such that static logic cannot use any wires from PR regions. Unfortunately, this method can only work with ISE vendor tool, which has been completely replaced by Vivado. Similar works are [69, 29].

Second, the scalability of PR compilation cannot easily be controlled by the users. Since PR implementation has to take the wires occupied by static logic into account, the context logic is related to both static logic and the number of PR regions in addition to the target PR region itself. The abstract-shell technique can partially address this issue by only reserving wires related to the target PR region [134]. However, the abstract shell design checkpoint can inevitably be affected by the locations and the nearby logic. Symbiflow [93] and RapidWright [73] are two excellent open-source tools that can both manipulate the FPGA database to performance implementations on a portion of the entire FPGAs. Since Symbiflow does not support all devices by vendors, and RapidWright is limited in generating partial bitstreams, this dissertation does not leverage these tools. Nevertheless, we believe

these tools can potentially improve the scalability of PR compile time when they become more mature.

7.4 Faster Soft Cores and More Memory Options

This dissertation shows how to use RISC-V cores to map operators to realize software first for incremental development. However, the performance of the RISC-V cores is still poor compared with X86 or FPGA accelerators. For the optical flow benchmark, we see the total runtime is even longer than the FPGA compilation time. This is because our RISC-V is a simple lightweight version. We can speed up the execution by pipelining the RISC-V cores as general CPUs. Additionally, some common hardware cores (e.g., floating point multipliers, vector units [141], etc.) are also missing in our RISC-V CPUs. We believe the performance of pure `-O0` in this dissertation can be greatly improved by supporting these common acceleration methods in CPU.

In addition, great efforts have been put into reducing the size of the ELF file to fit the limited size of the on-chip BRAMs in PR regions. However, we cannot completely address the issue when an array with a big length is utilized in an operator. One possible solution is to use the on-chip BRAMs only as cache and use the PSNoC to access the big off-chip DDR memory or HBM memory, which are usually several Gigabytes.

7.5 More Directives for Decomposition

Given an application written in the latency-insensitive model, this dissertation can handle the back-end compilations from HLS to bitstreams generations. We leave the users to prepare their code in a dataflow graph manner. However, this might add some burden on the developers and impact the coding efficiency. We believe adding more directives/pragmas to guide the tools to decompose the code into several small operators may be more efficient for design space auto-exploration in the future. Some existing works may be potential solutions, such as HeteroRefactor [72] and AutoSA [114]. It might be more interesting to integrate more front-end compilers to automate compilation flow more efficiently.

7.6 Hard NoC

The hardware embedded NoC is proposed in [2], which can increase the frequency by 10–80%, and reduces utilization of long wires by 40%. Our work also used Packet-Switched-Network-on-a-Chip (PSNoC) for spatial linkage to connect separately-compiled PR blocks together in [95, 123], before Versal ACAP system [137] was released by Xilinx. The limited bandwidth between the NoC and pages (0.8GB/s) may bring up some IO bottleneck issues, which can partially be solved by using directed wires in Chapter 4 and dedicated links in Chapter 6. The hard NoC system in Versal chips provides new opportunities for our PRflow implementation. The NoC system includes NoC master units (NMUs), NoC slave units (NSUs), and NoC packet switches (NPSs). Stacked Silicon Interconnect Technology (SSIT) is employed for multi-SLR NoC inter-die bridges (NIDBs). The hard NoC can be leveraged by PRflow to increase the inter-page bandwidth and save more resources for users.

7.7 Scalability for Larger Devices

Scalability for larger devices is an important feature of hardware compilation tools. Different compilation frameworks proposed in this dissertation have different characteristics for scalability.

For our `-O0` (RISC-V) and `-O1` (PR) options, the page compilation time should not scale up with larger devices if we define the same page size (around 20K-LUTs). For the inter-page connections, we use Packet-Switched NoC. There is no need for a good placement algorithm; the only constraint is that operators should fit the sizes of the mapped pages. The spatial linkage can be implemented by configuring the PSNoC. The configuration time is determined by the number of configuration packets, which are proportional to the number of streaming links to connect the operators. As it only takes one FPGA clock cycle (2–5 ns) to send a configuration packet, the spatial linkage is unlikely to dominate compile time. The overlay generation time can scale up since more pages can be pre-implemented on larger devices. But this is a one-time overhead during the framework preparation for the new devices and does not affect the compilation time for different applications.

For our `-O2` (Direct Wires), the compilation time for pages does not scale up if we use the same page size (around 20K-LUTs) for larger devices. However, the page assignment for all the operators can scale up super-linearly since we use the Simulated-Annealing algorithm for operator placement and the greedy routing algorithm to find pipelined direct wires across pages (chapter 4.4). With 24 pages defined on the Alveo U50 data-center FPGA, operator assignment and pipeline wire routing can be completed within seconds. Currently, the page assignment and operator routing are far from dominating the compilation time. Still, it is worth employing proper algorithms to improve scalability in dealing with more pages on larger devices in the future. For example, the bi-section [16] or mincut [38] have nearly linear time related to the number of operators for the placement problem.

Since HiPR `-O3` generates application-specific overlays, the compile time includes overlay-compile time and incremental-compile time. Incremental-compile time for pages does not scale up with devices. It is related to the initial size of the operators and the elastic resource pragma (Figure 6.3(b) Line 3). For the overlay-compile time, the overlay placement&routing has similar scalability as the vendor tools because it is implemented more like standard Vitis flow. For the floorplan, we currently use a lightweight Simulated Annealing algorithm. The floorplan time is around seconds for operator number around 20. When mapping larger designs to larger devices, Simulated Annealing may not scale well. Using a proper floorplan algorithm to map larger designs can be another future work to make HiPR work well on larger devices.

Chapter 8

Conclusion

A divide-and-conquer compilation framework (PRflow) is developed for accelerating FPGA compilation, which splits a complete FPGA compilation into several independent partial reconfiguration compilations. Specifically, we pre-define separate partial reconfiguration blocks at the FPGA layout level and perform a one-time implementation to get an overlay for later use. A packet-switched network-on-a-chip (PSNoC) is adopted to connect these separate PR regions together with a uniform interface. For the applications, we encourage developers to use the latency-insensitive model to prepare the source code. An application is decomposed into separate operators, which are agnostic to each other and only communicate with each other by streaming interfaces. Each operator only processes valid in-comping data and flushes data out when its consumer is ready. Otherwise, the operator stalls. Subsequently, the operators at C-level can be mapped in parallel to the separate no-overlapping locations on an FPGA chip. By configuring the local registers on each page, virtual connections can be set up by PSNoC without routing physical wires.

By mapping Rosetta HLS benchmarks [146], PRflow can accelerate the compilation 6.4–10.9× than the state-of-the-art vendor tools. In terms of performance, we note that the throughput of some benchmarks is degraded due to the limited fixed bandwidth of the NoC (6.4 Gbps), which we call the IO bottleneck. Additionally, applications may not run unless all the decomposed operators can fit the fixed-page size. This size-fit issue may limit the flexibility for development since developers usually start with an executable application and perform refinement incrementally to optimize it.

To address the IO bottleneck issue, we propose Direct Wires framework. Instead of using PSNoC to connect separate blocks, the FPGA chip is divided into a grid of rectangle blocks. Two adjacent rectangle blocks are connected by re-routed fixed wires. The key design principle is to route a maximum number of wires under specific constraints for later PR recompilation. A route-aware placement tool is developed to assign the separate operators at C-level to the proper blocks without over-using the pre-routed fixed wires. By mapping the same benchmarks, the performance can be improved by at most 10×, with a similar compilation-time speedup as PRflow.

To address the size-fit issue, we extend the PRflow with soft cores support by pre-compiling RISC-V cores for quick mapping. This can ensure that developers can start their development with a runnable application without dedicated code decomposition. Both RISC-V hardware and software libraries are customized mainly to constrain the ELF file to be small enough to be mapped by the Block RAMs (BRAMs) within a page. The softcore can also act as a debugging module interposed into the dataflow to redirect debugging information from the hardware without re-implementing the applications. This hardware print function is meaningful for on-chip debugging.

While Direct Wires and Softcore can partially address the IO-bottleneck and size-fit issues, both have their limitations. For example, Softcores cannot provide hardware-comparable performance, and Direct Wires still suffer from fixed-size page issues. To overcome the above limitations, HiPR is proposed to help developers to generate customized overlays for different applications. By using relay stations with arbitrary datawidth to connect separate PR regions, performance will not be degraded by IO-bottleneck. With a lightweight floorplanner, operators can be mapped to PR regions on demand. Users can define a ratio to deliberately assign larger PR regions for later refinement. With an acceptable compilation time overhead to customize the overlay, both size and bandwidth requirements can be met with similar incremental compilation as PRflow.

In conclusion, PRflow can accelerate the FPGA compilation from 2–3 hours (state-of-the-art Vitis) to 10-24 minutes. We believe the divide-and-conquer compilation strategy

can reduce the compilation time gap between FPGAs and general computing platforms, making more developers embrace FPGAs for their applications.

BIBLIOGRAPHY

[1] M. Abbas and V. Betz. Latency insensitive design styles for fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–3607. IEEE, 2018.

[2] M. S. Abdelfattah, A. Bitar, and V. Betz. Take the highway: Design for embedded nocs on fpgas. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 98–107, 2015.

[3] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 89–108, 2010.

[4] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):609–626, 1997.

[5] P. Banerjee, M. Sangtani, and S. Sur-Kolay. Floorplanning for partial reconfiguration in fpgas. In *2009 22nd International Conference on VLSI Design*, pages 125–130. IEEE, 2009.

[6] P. Banerjee, M. Sangtani, and S. Sur-Kolay. Floorplanning for partially reconfigurable fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(1):8–17, 2010.

[7] K. Beck and C. Andres. *Extreme Programming Explained.* Addison-Wesley, 2004.

[8] 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*, pages 532–538, 2011.

[9] G. Borriello, C. Ebeling, S. A. Hauck, and S. Burns. The triptych fpga architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):491–501, 1995.

[10] J. Boucaron, A. Coadou, and R. De Simone. Latency-insensitive design: retry relay-station and fusion shell. *Electronic Notes in Theoretical Computer Science*, 245:23–33, 2009.

[11] A. Brant and G. G. Lemieux. ZUMA: An open FPGA overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*, pages 93–96. IEEE, 2012.

[12] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 77–86, 1997.

[13] 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*, pages 218–231, 2020.

[14] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*, chapter 19. Addison Wesley Logman, Inc., 25th anniversary edition, 1995.

[15] M. Butts, A. M. Jones, and P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 55–64. IEEE, 2007.

[16] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Can recursive bisection alone produce routable placements? In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, page 477–482, New York, NY, USA, 2000. Association for Computing Machinery.

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

[18] 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*, pages 1073–1086, 2019.

[19] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). In *International Workshop on Field Programmable Logic and Applications*, pages 605–614. Springer, 2000.

[20] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[21] 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*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery.

[22] J. Cheng, L. Josipovic, G. A. Constantinides, P. Ienne, and J. Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 288–298, 2020.

[23] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. Dass: Combining dynamic & static scheduling in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(3):628–641, 2021.

[24] 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*, pages 327–338, 2016.

[25] 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*, 38(2):8–20, 2018.

[26] D. D. Clark. Window and acknowledgement strategy in tcp. Technical report, Online, 1982.

[27] J. Cong and J. Wang. Polysa: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[28] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, 2010.

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

[30] N. Deak, O. Cret, and H. Hedesiu. Efficient fpga floorplanning for partial reconfiguration-based applications. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 309–309. IEEE, 2019.

[31] 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. *Microprocessors and Microsystems*, 30(6):334–354, 2006.

[32] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *ISFPCCM*, pages 143–151. IEEE, 2006.

[33] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[34] S. R. Dickey and R. Kenner. A combining switch for the NYU ultracomputer. *Ultracom-puter Note*, 1(178), 1992.

[35] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser. Generic software framework for adaptive applications on FPGAs. In *ISFPCCM*, pages 55–62, 2009.

[36] C. Fallin, C. Craik, and O. Mutlu. CHIPPER: A low-complexity bufferless deflection router. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 144–155, 2011.

[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)*, pages 6 pp.–, 2006.

[38] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[39] B. A. Forouzan. *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[40] 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 *International Symposium on Computer Architecture*, pages 1—-14. IEEE Press, 2018.

[41] K. Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[42] GNU. *GNU make*, 2021 (Accessed: 2021-09-10).

[43] Google. *Deploying a Slurm cluster on Compute Engine*, 2021 (Accessed: 2021-08-10).

[44] J. Gray. GRVI phalanx: A massively parallel RISC-V FPGA accelerator accelerator. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20, 2016.

[45] J. Gray. Grvi phalanx joins the kilocore club. `http://fpga.org/2017/01/12/grvi-phalanx-joins-the-kilocore-club/`, January 2017.

[46] N. B. Grigore, C. Kritikakis, and D. Koch. Hls enabled partially reconfigurable module implementation. In *International Conference on Architecture of Computing Systems*, pages 269–282, Braunschweig, Germany, 2018. Springer, Springer.

[47] 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 *ISFPGA*, pages 81—-92, New York, NY, USA, 2021. ACM.

[48] 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*, FPGA '22, pages 1–12, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Gurobi. *Version 9.5.1.* GUROBI OPTIMIZATION, LLC., 2022 (Accessed: 2022-12-31).

[50] J. D. Hadley and B. Hutchings. Design methodologies for partially reconfigured systems. In *FCCM*, pages 78–84, April 1995.

[51] Intel. *AN 797: Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board*, 2018.

[52] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *ISFPCCM*, pages 1–8, 2016.

[53] A.-S. Jamal, J. Goeders, and S. J. Wilton. An FPGA overlay architecture supporting rapid implementation of functional changes during on-chip debug. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 403–4037. IEEE, 2018.

[54] G. Jo, H. Kim, J. Lee, and J. Lee. SOFF: an OpenCL high-level synthesis framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 295–308. IEEE, 2020.

[55] L. Josipovic, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[56] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136, 2018.

[57] L. Josipovic, A. Guerrieri, and P. Ienne. Speculative dataflow circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 162–171, 2019.

[58] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2142–2155, 2021.

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

[60] 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)*, pages 9–16. IEEE, 2015.

[61] N. Kapre. Deflection-routed butterfly fat trees on fpgas. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

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

[63] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 107–127, 2018.

[64] R. Kirchgessner, A. D. George, and G. Stitt. Low-overhead FPGA middleware for application portability and productivity. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(4):1–22, 2015.

[65] R. Kirchgessner, G. Stitt, A. George, and H. Lam. Virtualrc: a virtual FPGA platform for applications and tools portability. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 205–208, 2012.

[66] D. Koch, C. Beckhoff, and G. G. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *2013 23rd international conference on field programmable logic and applications*, pages 1–8. IEEE, 2013.

[67] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. *Toronto*, 2009.

[68] D. T. Kwadjo, E. Nghonda Tchinda, J. M. Mbongue, and C. Bobda. Accelerating hybrid quantized neural networks on multi-tenant cloud fpga. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 491–498, 2022.

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

[70] 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*, pages 818–831, 2021.

[71] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on computers*, 100(12):1469–1479, 1971.

[72] J. Lau, A. Sivaraman, Q. Zhang, M. A. Gulzar, J. Cong, and M. Kim. Heterorefactor: Refactoring for heterogeneous computing with fpga. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 493–505, New York, NY, USA, 2020. Association for Computing Machinery.

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

[74] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using hard macros to reduce fpga compilation time. In *2010 International Conference on Field Programmable Logic and Applications*, pages 438–441, 2010.

[75] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating fpga compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, 2011.

[76] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. RapidSmith: Do-it-yourself CAD tools for xilinx FPGAs. In *Proceedings of International Workshop on Field-Programmable Logic and Applications (FPL)*, September 2011.

[77] C. E. Leiserson. Vlsi theory and parallel supercomputing. *Defense Technical Information Center*, 1989.

[78] D. Lim and M. Peattie. *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, May 2002.

[79] M. Lin. The amorphous fpga architecture. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 191–200, 2008.

[80] C. Liu, H. C. Ng, and H. K. H. So. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *ICFPT*, pages 56–63, 2015.

[81] S. Ma, Z. Aklah, and D. Andrews. Just in time assembly of accelerators. In *ISFPGA*, pages 173–178, 2016.

[82] P. Maidee, C. Neely, A. Kaviani, and C. Lavin. An open-source lightweight timing model for rapidwright. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 171–178, 2019.

[83] 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*, 47(1):15–31, 2007.

[84] J. Mandebi Mbongue, D. Tchuinkou Kwadjo, and C. Bobda. Automatic generation of application-specific fpga overlays with rapidwright. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 303–306, 2019.

[85] 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*, 38(1):107–130, 2004.

[86] E. Micallef, Y. Xiao, and A. DeHon. Hls-compatible, embedded-processor stream links. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 214–218, 2021.

[87] 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)*, pages 1–8. IEEE, 2010.

[88] 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)*, 3(4):1–34, 2010.

[89] 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*, 15(12):1518–1524, 1996.

[90] M. Nguyen and J. C. Hoe. Amorphous dynamic partial reconfiguration with flexible boundaries to remove fragmentation. *arXiv preprint arXiv:1710.08270*, 2017.

[91] 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*, pages 149–158, 2016.

[92] Online. RISC-V. `riscv.org`.

[93] Online. SymbiFlow: fpga-tool-perf. `https://github.com/SymbiFlow`, 2021.

[94] Oracle. *Oracle Grid Engine*. Online, 2021 (Accessed: 2021-08-10).

[95] D. Park, Y. Xiao, N. Magnezi, and A. DeHon. Case for fast fpga compilation using partial reconfiguration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 235–2353. IEEE, 2018.

[96] T. M. Parks. *Bounded Scheduling of Process Networks*. UCB/ERL95-105, University of California at Berkeley, 1995.

[97] 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 paterns. *ACM SIGARCH Computer Architecture News*, 45(2):389–402, 2017.

[98] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, et al. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 02CH37315)*, volume 1, pages 334–472. IEEE, 2002.

[99] 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*, 25(1):151–164, 2016.

[100] 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*, pages 186–193. IEEE, 2014.

[101] 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*, pages 252–255, 2015.

[102] A. Sadeghi, M. Z. Lighvan, and P. Prinetto. Automatic and simultaneous floorplanning and placement in field-programmable gate arrays with dynamic partial reconfiguration based on genetic algorithm. *Canadian Journal of Electrical and Computer Engineering*, 43(4):224–234, 2020.

[103] M. Saldana, L. Shannon, J. S. Yue, S. Bian, J. Craig, and P. Chow. Routability of network topologies in fpgas. *IEEE Transactions on very large scale integration (VLSI) Systems*, 15(8):948–951, 2007.

[104] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–286, 2019.

[105] B. B. Seyoum, A. Biondi, and G. C. Buttazzo. Flora: Floorplan optimizer for reconfigurable areas in fpgas. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20, 2019.

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

[107] A. A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood. OpenPR: An open-source partial-reconfiguration toolkit for xilinx fpgas. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 228–235. IEEE, 2011.

[108] G. Stitt, F. Vahid, and W. Najjar. A code refinement methodology for performance-improved synthesis from C. In *ICCAD*, pages 716–723, 2006.

[109] J. Varghese, M. Butts, and J. Batcheller. An efficient logic emulation system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):171–174, 1993.

[110] R. Venkatakrishnan, A. Misra, and V. Kindratenko. High-level synthesis-based approach for accelerating scientific codes on FPGAs. *Computing in Science & Engineering*, 22(4):104–109, 2020.

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

[112] K. Vipin and S. A. Fahmy. Automated partitioning for partial reconfiguration design of adaptive systems. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 172–181. IEEE, 2013.

[113] K. Vipin and S. A. Fahmy. Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In *FCCM*, pages 202–205, May 2014.

[114] J. Wang, L. Guo, and J. Cong. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *ISFPGA*, pages 93–104, New York, NY, USA, 2021. ACM.

[115] T. Wiersema, A. Bockhorn, and M. Platzner. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6. IEEE, 2014.

[116] D. Wilson and G. Stitt. Seiba: An FPGA overlay-based approach to rapid application development. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2019.

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

[118] C. Wolf. PicoRV32—a size-optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`, 2020.

[119] Y.-L. Wu and D. Chang. On the np-completeness of regular 2-d fpga routing architectures and a novel solution. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 362–366, 1994.

[120] Y. Xiao, S. T. Ahmed, and A. DeHon. Fast linking of separately-compiled FPGA blocks without a NoC. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 196–205, 2020.

[121] Y. Xiao, A. Hota, D. Park, and A. DeHon. HiPR: High-level partial reconfiguration for fast incremental FPGA compilation. In *2022 32nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2022.

[122] 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*, ASPLOS 2022, pages 933–945, New York, NY, USA, 2022. Association for Computing Machinery.

[123] 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 *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 153–161, 2019.

[124] Xilinx. *xilinx-u50-gen3x16-xdma-dev-201920.3-2784799_all.deb*, 2021 (Accessed: 2022-06-18).

[125] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC6200 FPGA Advanced Product Specification*, version 1.0 edition, June 1996.

[126] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *WP477 (v1.0): UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices*, June 2016.

[127] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG909 (v2018.1): Vivado Design Suite User Guide Dynamic Function eXchange*, February 2018.

[128] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG984: MicroBlaze Processor Reference Guide*, June 2018.

[129] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG1027: SDSoC Environment User Guide*, May 2019.

[130] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG908: Vivado Design Suite User Guide: Programming and Debugging*, January 2019.

[131] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG1393: Xilinx Vitis Unified Software Platform User Guide*, June 2020.

[132] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG897: Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator*, June 2020.

[133] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG902: Vivado Design Suite User Guide: High-Level Synthesis*, January 2020.

[134] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG909 (v2020.2): Vivado Design Suite User Guide Dynamic Function eXchange*, February 2020.

[135] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG1120: Alveo Data Center Accelerator Card Platforms*, April 2021.

[136] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG904: Vivado Design Suite User Guide Implementation*, August 2021.

[137] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG1388: Versal ACAP System Integration and Validation Methodology Guide*, May 2022.

[138] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG947 (v2022.2): Vivado Design Suite Tutorial: Dynamic Function eXchange*, February 2022.

[139] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *UG949 (v2022.2): UltraFast DesignMethodology Guide for FPGAs and SoCs*, November 2022.

[140] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Runtime (XRT) Architecture*, June 2022.

[141] P. Yiannacouras, J. G. Steffan, and J. Rose. Portable, flexible, and scalable soft vector processors. *TRVLSISYS*, 20(8):1429–1442, 2012.

[142] Y. Zha and J. Li. Reconfigurable in-memory computing with resistive memory crossbar. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2016.

[143] 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*, ASPLOS '20, page 845–858, New York, NY, USA, 2020. Association for Computing Machinery.

[144] Y. Zha and J. Li. 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*, pages 123–134, 2021.

[145] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24, 2017.

[146] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, et al. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 269–278, 2018.

[147] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt. Rwroute: An open-source timing-driven router for commercial fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 15(1), nov 2021.