SOFTWARE-LIKE INCREMENTAL REFINEMENT ON FPGA USING PARTIAL RECONFIGURATION

Dongjoon Park

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

2024

ACKNOWLEDGEMENT

I am deeply grateful to Professor DeHon for his mentorship throughout my PhD journey. Under his guidance, I have learned not only technical knowledge but also the skills of crafting academic papers and presenting research to diverse audiences. I have gained so much knowledge in this field and have gone through a one-of-a-kind PhD journey that I would have been able to experience elsewhere. For the rest of my career, whenever I face a research challenge, my first question will always be, "How would André approach this problem?" I am confident that this question will shed light on the problem, recalling the many enriching discussions we shared over the past few years.

I am also thankful to Professor Li for her guidance when I struggled when I returned after a three-year leave. Her encouragement helped me to rebuild my confidence and renew my focus on my research. I would like to thank Professor Mangharam for his incisive feedback during my thesis proposal, which strengthened my work and refined my final presentation.

I would like to thank IC group members, Rafi, Hans, Nick, Yuanlong, Syed, and Ezra. We shared ideas, faced challenges, and supported each other. I hope our paths cross again in the future.

Lastly, I would like to thank my family who have patiently supported me every step of the way. Their unwavering encouragement has helped me to persevere in the toughest moments. Thank you for everything.

ABSTRACT

SOFTWARE-LIKE INCREMENTAL REFINEMENT ON FPGA
USING PARTIAL RECONFIGURATION

Dongjoon Park

André DeHon

Jing Li

To improve FPGA design productivity, our goal is to create a development experience for FPGAs that aligns closely with widely accepted software design principles. Software programmers quickly test their minimally completed design, identify the bottleneck, and incrementally refine the design. In FPGA design, however, such incremental refinement is not supported. (1) FPGA compilation is long, (2) a minor refinement leads to another long compilation, and (3) FPGA developers cannot easily identify a bottleneck of the design. We introduce a divide-and-conquer strategy in FPGA compilation, proposing a fast separate FPGA compilation using a Network-on-Chip (NoC) and Partial Reconfiguration (PR). This approach enables parallel and incremental FPGA compilation but requires users to manually decompose designs into operators that fit fixed-sized pages. In this thesis, we take the next step to support variable-sized pages using Hierarchical PR to provide flexibility to the users. With variable-sized pages, users can decompose a design naturally, without careful planning, enabling rapid hardware testing in a similar manner to how software programmers start testing with a minimally functional prototype. In addition, we propose a bottleneck identification scheme based on FIFO counters to provide profiling capability in FPGA design. Finally, we introduce a fast incremental refinement strategy that integrates our fast compilation framework and bottleneck identification scheme. The idea is to quickly map the design on the FPGA using the fast compilation framework and incrementally refine the design based on our bottleneck identification. The fast compilation with the NoC and PR pages iterates many initial yet important design points quickly, and for the final, optimized design, our strategy migrates to the monolithic system that does not have the area and bandwidth overhead of the NoC. Throughout the design tuning, we always have a hardware-mapped design whose performance we can measure to provide feedback to the users or automation script to identify the next bottleneck. We evaluate our fast incremental strategy with design tuning for realistic High-Level Synthesis applications. Our framework, fully compatible with AMD Vitis, achieves 1.3–2.7× faster tuning time than a monolithic flow where the vendor tool monolithically compiles each design point.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1. Thesis

Through the fast and flexible FPGA compilation using Hierarchical Partial Reconfiguration, FPGA design-space exploration can be accelerated up to 2.7×, resulting in a final optimized design that is comparable to the optimized design when design-space exploration is done with the vendor tool's monolithic compilation.

1.2. Motivation

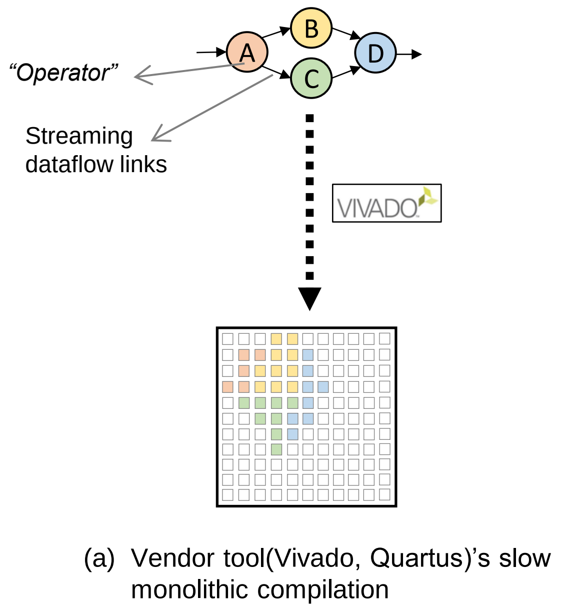
No programmer builds a system in a single compilation. Software programmers generally develop something that is barely functional and incrementally add functionality to complete their designs. *Incremental Refinement* is a natural practice when building even a simple system and inherently introduces many edit-compile-debug cycles. While Field-Programmable Gate Arrays (FPGA) are known to be flexible and power-efficient so that FPGAs can potentially possess the best of the both worlds of processors and Application-Specific Integrated Circuit (ASIC), a main limitation is FPGA's notoriously long compilation time. Compilation to CPU (Central Processing Unit) and GPU (Graphics Processing Unit) takes seconds or at most minutes while compilation to FPGA takes at least minutes and hours. What is worse is that even if a small function in an FPGA design is modified, today's FPGA vendor tools *monolitically* recompile the entire design; the tools try to optimize the whole design instead of actively reusing the previously compiled design. This monolithic FPGA compilation results in lengthy incremental compilation times, limiting the number of design points that can be explored. In software, on the other hand, programmers only need to recompile functions that have changed, and newly compiled object files are linked with the previously compiled objects. Therefore, even if the size of the design increases, incremental compilation time stays short.

Furthermore, when programmers incrementally refine their designs, they profile the design to identify the bottleneck function that limits the performance of the system, improve the bottleneck, and recompile the design to test whether the performance has improved. Not only the incremental compilation in FPGA is lengthy, but also FPGA developers do not have probing tools to identify the bottleneck. The lack of visibility in the hardware design results in a sub-optimal final design and makes hardware development more challenging.

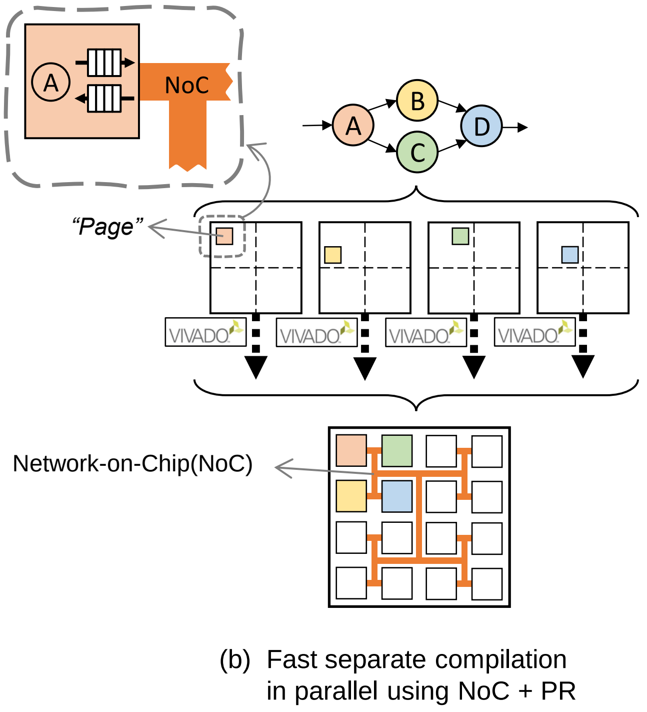
FPGA provides massive data-level parallelism and task-level parallelism. FPGA's reconfigurability provides flexibility that ASIC, which performs a single task in its entire lifespan, does not offer. Because many of today's applications require tremendous parallelism and power efficiency at the same time, users of traditional computing chips like CPUs and GPUs have come to pay more attention to FPGA. The fast-changing characteristic of these applications makes FPGA more attractive since ASIC cannot be changed after being manufactured, and manufacturing costs of ASIC are high. However, the aforementioned limitations in today's vendor tools prevent FPGA from becoming the mainstream hardware platform, and in many industries, FPGA still stays as a mere glue logic for different peripherals or ASIC emulator instead of being a computing device itself. Given that most engineers are accustomed to incremental refinement in software development, it raises the question of whether similar design principles can be applied to FPGA development.

1.3. Approach

To lower the entrance barrier for those who have never tried FPGA and to enhance the productivity of experienced FPGA engineers, this dissertation aims to support software-like FPGA development. A main hindrance in closing the gap between software development and FPGA development is the long compilation time of FPGA design. A reason for the notoriously long compilation is that AMD's Vivado and Altera's Quartus perform a monolithic compilation. To address the long monolithic compilation from FPGA vendor tools, we adopt a divide-and-conquer strategy in FPGA compilation. We split the FPGA device into multiple sub-regions and launch multiple FPGA compilation runs to compile each sub-region in parallel. Separately compiled sub-regions are linked with a pre-compiled packet-switched Network-on-Chip (NoC). The vendor tool compiles each sub-region monolithically, but because the problem size decreases, the compilation is faster. As today's vendor tool does not fully utilize abundant cores in modern desktop CPUs or compute servers, we deliberately decou-



(a) Vendor tool(Vivado, Quartus)'s slow monolithic compilation



(b) Fast separate compilation in parallel using NoC + PR

Figure 1.1: High-Level Idea of Separate Compilation using NoC and PR

ple sub-regions in a single FPGA device and perform separate compilations in parallel, explicitly utilizing more cores. Realizing our vision relies on two key components: a NoC and Partial Reconfiguration (PR) technology. A NoC enables full virtualization, allowing any operator to send and receive data from others. PR allows a partition of the design to be separately compiled. The high-level idea of our approach is described in Figure 1.1. We define an *operator* as a logical streaming computation block of the user design. In Figure 1.1's example, the user design consists of four operators, and these operators are connected with streaming links. The operators can be coded in either Register-Transfer level (RTL) or High-Level synthesis (HLS). We define each *page* as a physical partition of the FPGA device that operators will be mapped on, and these regions are PR regions. As can be seen in Figure 1.1 (b), NoC interfaces are compiled along with operators in designated pages. In Figure 1.1's case, our approach should provide fast compilation because each operator is compiled in parallel. In addition, with our system, only the changed operators can be recompiled because we decouple the compilation for each operator in contrast to monolithic compilation by the vendor tool where a minor fix in the design requires the recompilation of the entire design. Our FPGA compilation approach is inspired by software compilation where each function is compiled into a separate object file and linked together, allowing only the modified functions to be incrementally recompiled.

Although our separate compilation approach shows the potential of software-like FPGA compilation, the *fixed-sized* pages impose responsibility on the users to decompose a design into small operators that can fit into fixed-sized pages. If the separate compilation framework has large pages to reduce the user's burden, then the compilation speedup decreases compared to the system with many small pages. Therefore, we adopt Hierarchical Partial Reconfiguration technology to offer *variable-sized* pages. With variable-sized pages, small operators can be mapped on single-sized pages as done in the system with the fixed-sized pages. In addition, multiple single-sized pages can be recombined to offer large pages so that large operators can also be mapped. Hierarchical pages let the users start mapping their initial design quickly on hardware because the users can quickly start from the *natural decomposition* of the design without considering the expected resource usage of each operator. This flexibility enables the users to start from the barely functional design and incrementally add functionality just like the users build software designs.

Furthermore, to support bottleneck identification in our separate compilation framework, similar to software programming, we insert FIFO counters. Based on the number of full and stall counters after application execution, the users get to have more visibility on the design mapped on the hardware and identify the slowest operator in the design or whether the NoC bandwidth is a bottleneck at all. With all components together, we propose a fast incremental refinement strategy (Figure 1.2). We quickly map the design on the FPGA using our separate compilation framework, which consists of a NoC and PR pages. Then, we identify a bottleneck one at a time based on hardware execution results and refine the bottleneck operator to reduce the application runtime. The presence of NoC could introduce area or bandwidth overhead to the design. Thus, when the design reaches the point that it cannot improve in our separate compilation framework, we monolithically compile the design on FPGA where FIFOs directly connect operators, and we keep FIFO counters for profiling. The incremental refinement continues, identifying the bottleneck and improving the bottleneck

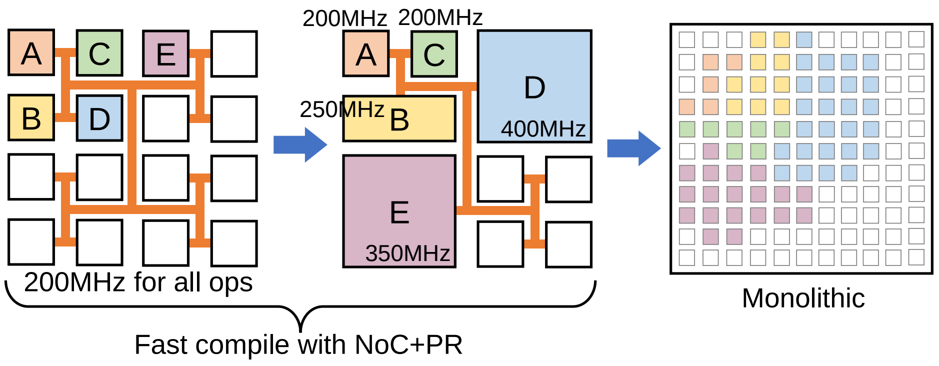


Figure 1.2: Fast Incremental Refinement Strategy on FPGA

operator. Our system is fully integrated with AMD Vitis acceleration flow [12]. Our solution resembles software programming practices, providing an intuitive and natural design practice for both software and hardware engineers. Our solution leverages FPGA's reconfigurability, mapping different design points incrementally and converging to a better design each step.

1.4. Contributions

- Using Hierarchical PR, we enhance the separate FPGA compilation framework with variable-sized pages to add flexibility. Unlike prior works, the users do not need to decompose a design into smaller operators to utilize the separate compilation framework. We show that variable-sized pages improve application performance by 1.4–4.9× while still compiling 2.2–5.3× faster than AMD Vitis.

- We further improve the separate compilation framework by supporting multiple NoC interfaces to a single operator, multiple clock frequencies (200MHz–400MHz) for different operators, and page assignment based on recursive graph bi-partitioning.

- We propose a fast, automatic runtime bottleneck identification scheme based on the FIFO counters. We demonstrate that our incremental refinement strategy reduces application tuning time by 1.3–2.7× compared to the monolithic flow. The fast FPGA compilation framework integrated with the bottleneck identification scheme is released as open-source.[1]

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

- To accommodate a variety of workloads in the fast compilation framework, we propose an asymmetric Butterfly Fat Tree (BFT) NoC architecture to selectively provide more bandwidth to specific sub-trees while using similar logic resources compared to traditional symmetric BFT NoC. We show that in realistic workloads and statistical traffic patterns, asymmetric BFTs achieve up to 32% and 76% more throughput than symmetric BFTs, respectively.

CHAPTER 2

BACKGROUND

In this chapter, we discuss the key elements of our fast FPGA compilation strategy: Partial Reconfiguration (PR) and Network-on-Chip (NoC). We also explain the compute model underlying our idea and the high-level overview of the divide-and-conquer strategy in FPGA compilation. Additionally, we review the related work on fast FPGA compilation using pre-compiled macros. This dissertation assumes that readers are familiar with FPGA architecture and FPGA compilation process. In this thesis, “synthesis” refers to logic synthesis, and “implementation” refers to the rest of the FPGA compilation including placement, routing, and bitstream generation unless explicitly stated otherwise.

2.1. Partial Reconfiguration

FPGA’s key differentiating feature from ASIC is that FPGA is reconfigurable. Users can configure Look-Up-Tables (LUTs), Flip-Flops (FFs), on-chip Block RAMs (BRAMs), Digital Signal Processing (DSP) blocks and programmable interconnect so that the circuit behaves like the hardware design they want. Unlike ASIC, which requires substantial Non-Recurring Engineering (NRE) costs, FPGA offers a fast time-to-market solution because FPGA can support arbitrary hardware designs. FPGA’s reconfigurability also provides flexibility that ASIC does not offer, enabling users to adapt designs to accommodate environmental or algorithmic changes. Partial Reconfiguration (PR) is a technology that reconfigures only parts of an FPGA device while other parts of the design are running.

PR is traditionally used to employ a system that is larger than the device because functions can be time-multiplexed with PR. Also, when functionality needs to dynamically adapt at runtime, PR can be used in loading appropriate pre-generated partial bitstreams while other parts of the design are operating. PR can support virtualization to enhance productivity. For example, AMD’s Alveo platform [6] consists of (1) static region where system peripherals and PCI Express (PCIe) IP are implemented and (2) dynamic region where user kernels can be mapped. In this way, users do not worry about software and hardware interfaces but can focus on designing hardware accelerators. The partial bitstream for the compute kernel can be loaded while the remaining infrastructure peripherals are still operating. As explained in Chapter 1, we use PR to separately compile each partition of the device but do not dynamically reconfigure the device. If we strictly distinguish *dynamic* reconfiguration and *partial* reconfiguration as done in [94], our usage belongs to partial reconfiguration but not dynamic reconfiguration.

FPGA vendors support PR with their software [14, 5]. PR design consists of a *static logic* that does not change and a *reconfigurable logic* that is partially reconfigured. In AMD Vivado's PR, also known as Dynamic Function eXchange (DFX), users first need to specify physical regions on the device to be partially reconfigured (*pblocks*), link modules to the physical regions, and compile the design. After place-and-route, users carve out modules that are mapped on PR regions. Then, the remaining logic and routing are locked, and the remaining placed and routed design becomes the static design. The interface points between the static logic and the reconfigurable logic are called *Partition Pins*. On top of the static design, reconfigurable modules can be placed and routed within the corresponding PR regions and are connected to Partition Pins. Finally, partial bitstreams are generated accordingly. It is important to note that static logic can use routing resources inside reconfigurable pblocks, but reconfigurable logic cannot use routing resources outside the reconfigurable pblocks.[2] If the static design uses too much of routing resources inside the PR regions, the rest of the reconfigurable modules may not be successfully placed and routed to generate partial bitstreams. Therefore, the rule of thumb is to use "the most challenging" module to create the static design. Academic tools like [56, 86, 15] use a blocker macro that occupies the routing resources of the designated area so that the vendor tool cannot use the routing resource in the region. Altera Quartus's PR consists of almost identical steps. In this thesis, we will mostly use "PR" to refer Partial Reconfiguration instead of "DFX". We will mostly use "PR region" to refer a collection of cells that are set to be partially reconfigurable instead of "reconfigurable pblock".

[2]To be more precise, AMD Vivado relaxes this constraint by introducing "expanded routing region", and Altera Quartus also has a routing region defined apart from a place region. But it is still true that reconfigurable logic should not use routing resources outside of designated regions.

Hierarchical PR is also supported from FPGA vendors [14, 5] and an academic tool [55]. Hierarchical PR, also known as Nested DFX in AMD's Vivado, provides multiple levels of reconfiguration. For a given parent reconfigurable block, users can reconfigure the entire parent partition. Using Hierarchical PR, this partition can be subdivided into multiple child reconfigurable partitions, depending on the design requirements. We can also create grandchildren reconfigurable partitions, and AMD PR user guide states that there is no formal limit on the depth of Nested DFX [14]. Alveo platform offers one large dynamic region for the compute kernel, but using Nested DFX, the users can subdivide the dynamic region into multiple segments for more granular reconfiguration. In Chapter 3, we take advantage of Hierarchical PR to support variable-sized pages for fast separate compilation framework.

As explained above, to generate different partial bitstreams for a reconfigurable partition, we first load a static design and implement the reconfigurable logic on top of the static design. In conventional PR uses, partial bitstreams are prepared beforehand, so bitstream generation time in PR is rarely addressed. In our case, we use PR to accelerate FPGA compilation and expect the bitstream generation time is short for a small reconfigurable logic regardless of the size of the static design. However, we observe that as the size of the static design increases, the compilation time increases even if the task of Vivado is to compile the same-sized reconfigurable block [79, 99]. To mitigate this issue, AMD introduces *Abstract Shell* [14], a minimal physical and logical database for each reconfigurable partition, to explicitly remove unrelated parts of the static design when compiling for a partial bitstream.

2.2. Soft NoC and Hard NoC

As the design size increases and routing becomes complicated, interconnect can introduce significant overhead in area and energy [32]. A packet-switched Network-on-Chip (NoC) can be a solution to the routing-dominated design, saving routing resources by sharing the NoC channels. Instead of directly connecting different operators, operators can be mapped as processing elements, and they can send packets to the NoC which delivers the packets to the correct destinations. This modular design methodology not only eases the application development, providing a higher level abstraction view, but also relaxes routing difficulty because complicated routing is replaced with a simple linking of the NoC. Although a NoC costs resources that could otherwise be used in computation, FPGA NoC often results in a design running at a superior clock frequency.

FPGA vendors have recently provided a "Hard" NoC, an embedded NoC on the FPGA [35, 89, 3, 44], and researchers have long studied "Soft" NoC, an overlay NoC built on top of the commercial FPGA [52, 76, 43, 50, 61]. Hard NoC usually provides higher performance in terms of bandwidth, latency, and area than soft NoC [107]. However, soft NoC provides more flexibility in customizing the NoC to the user designs. For example, users can select the NoC topology and the number of nodes when the NoC is mapped on top of the programmable logic.

2.3. Compute Model

As shown in Figure 1.1, we assume that the application can be decomposed into an array of operators connected with streaming dataflow links [47, 22, 33]. In dataflow architecture, operators can start computations immediately when the data are ready. The handshaking protocol in dataflow typically consists of *data*, *valid*, and *ready*. A valid signal indicates that the data is ready from the producer side, and the ready signal indicates the consumer is ready to receive the data. The streaming compute model reduces the complexity of the design because operators can be developed independently, reducing the problem size. As long as each operator follows the handshaking protocol of dataflow links, each operator does not need to know the details of communication channel or implementations of other operators. The streaming compute model also promotes reusability because in contrast to conventional cycle-based hardware designs where each computation should occur at specific cycle, the implementation of previously used operators can be reused with a proper computational pipeline in different applications. Additionally, the streaming dataflow approaches can improve the achieved operating frequency by decoupling the implementations of operators. The reduced design complexity, increased reusability, and higher operating frequencies together make the streaming compute model as a scalable solution as the device size grows.

2.4. Divide-and-Conquer in FPGA Compilation

The motivation behind adopting the divide-and-conquer strategy in FPGA compilation is that today's FPGA vendor tool does not exhibit enough parallelism to utilize abundant cores in a multicore workstation or a compute server when compiling the design. The divide-and-conquer strategy enables the tool to utilize more available cores effectively, and the smaller problem size leads to shorter compilation times. It is difficult to accurately characterize the factors contributing to FPGA compilation time and their impact, but we expect that FPGA compilation time scales at least linearly with the size of the design. Compilation time could be a superlinear function of the size of the design if the vendor tool's implementation algorithms handle small designs well but face increasing challenges as the design size grows. For example, the number of moves in the VPR's placer based on simulated annealing with the $O(N^{4/3})$ [74] when N is the number of the blocks to place. The Pathfinder algorithm [73] used in VPR's router initially allows multiple signals to use the same routing resources routing resources temporarily. Then, the algorithm resolves the congestion by making popular routing resources more costly so that in the next iteration, signals are discouraged to use these routing resources. Because Pathfinder algorithm iterates all the signals, it scales at least linear to the number of nets in the design, and because of repeated rip-ups and re-routing, routing runtime scales superlinearly for complicated designs.

In reality, with today's commercial FPGA tools, there should be multiple factors contributing to compilation time including the size of the device, the size of the design, the target frequency, the depth of the logic, resource compositions of the design, and routing complexity. In our divide-and-conquer strategy, we decompose a large problem into smaller problems that can be solved independently, and the compilation time is the maximum of all parallel compile runs.

2.5. Pre-compiled Macros

One methodology to accelerate FPGA compilation is to use pre-compiled "macros". Instead of mapping designs into small primitives like LUTs, FFs, BRAMs, and DSPs every time, researchers have explored preparing pre-routed macros and translating the FPGA compilation problem into assembling these macros. This approach simplifies the compilation process but is likely to result in suboptimal designs. HMFlow [64] adopts this approach, simplifying the FPGA mapping problem into stitching the pre-compiled macros. In stitching, authors use RapidSmith [65], an open-source CAD tool that provides APIs for Xilinx Design Language (XDL) to manipulate Xilinx FPGA designs. Benchmark designs used in HMFlow range from 150 Slices to more than 23K Slices (in Xilinx Virtex 4 devices, there are two 4-input LUTs in a Slice), and the number of macro instances per benchmark ranges from 21 to more than 1400. Compilation time for HMFlow is from 1–129 seconds, which is 9.2–50× faster than the commercial tool. Similar to HMFlow, in BPR [26], authors use pre-compiled macros to reduce FPGA compilation time. In contrast to HMFlow which saves relatively small primitives in the library, BPR stocks coarse-grained cores, thereby achieving higher speedup. BPR achieves less than 3 seconds of compilation for the designs, which the vendor tool takes 9–139 seconds to compile. Recently, DynaRapid supports a fast C-to-Routed design compilation with pre-compiled dataflow compute blocks [38]. Authors pre-route a library of HLS circuits generated by a dynamic HLS tool called Dynamatic [46]. For benchmarks, ranging from 500 to 9500 LUTs, DynaRapid reduces 500 seconds of compilation time by the vendor tool to 15–50 seconds (excluding bitstream generation time). In [70], authors provide an overlay that consists of 2×2 or 3×3 PR regions of which size is 9600 LUTs. The idea is to prepare a stock of partial bitstreams in the bitstream repository, and the interpreter dynamically assembles partial bitstreams according to programming patterns defined in their Domain Specific Languages. Because they replace compilation with assembling partial bitstreams, their approach takes less than 5 seconds to output valid designs.

A problem with the approach using pre-compiled macros is that the user design is limited to the combinations of the pre-defined macros. In this thesis, we take the next step to accelerate compilations of arbitrary user designs using divide-and-conquer strategy.

CHAPTER 3

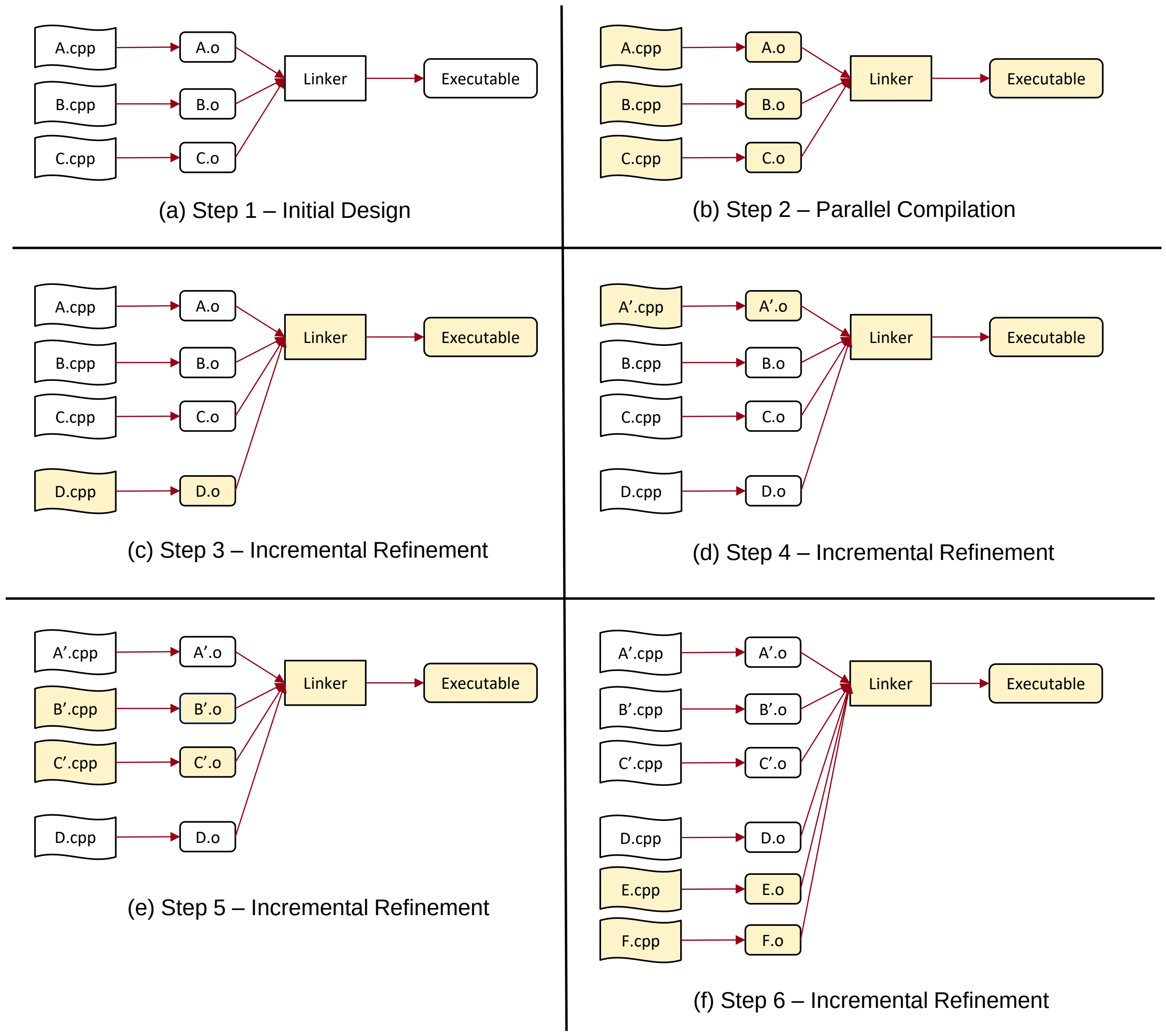
FAST AND FLEXIBLE FPGA DEVELOPMENT USING HIERARCHICAL PARTIAL RECONFIGURATION

In this chapter, we begin by reviewing parallel and incremental compilation in software with an example. We then review how today's FPGA compilation is different from software compilation. We discuss the prior efforts to support separate compilation on FPGA using fixed-sized pages and their limitations. Then, we explain how our approach using Hierarchical PR resolves the related issues with fixed-sized pages. This chapter was previously published in [Dongjoon Park, Yuanlong Xiao, and André DeHon. Fast and Flexible FPGA Development using Hierarchical Partial Reconfiguration. International Conference on Field-Programmable Technology. 2022.] [78]. I led the project and was in charge of the system implementation.

3.1. Motivation

Let us review software program development first and investigate whether the same practice can be applied in hardware development. Figure 3.1 illustrates a software development scenario. We, as software programmers, usually start with a simple design that is not fully functional but has appropriate placeholders (Figure 3.1 (a)). This is because having a simple yet stable foundation eases the debugging, reduces the risk, and lets the developers focus on the core functionality of the design. In software, these initial source codes can be compiled and assembled *in parallel* to object files as shown in Figure 3.1 (b). The object files are linked together to generate an executable file. Then, the rest of the design process is incremental. Programmers can add a new functionality as shown in Figure 3.1 (c),(f), or they can refine the existing function as shown in Figure 3.1 (d),(e). In software compilation, only the changed functions need to be recompiled because we already have object files for the unchanged functions. New or refined functions are compiled and simply linked together with the existing object files. Thus, even if the program becomes large, the incremental compilation stays short.

Let us take a look at how FPGA (hardware) development is different from software development.



(a) Step 1 – Initial Design

(b) Step 2 – Parallel Compilation

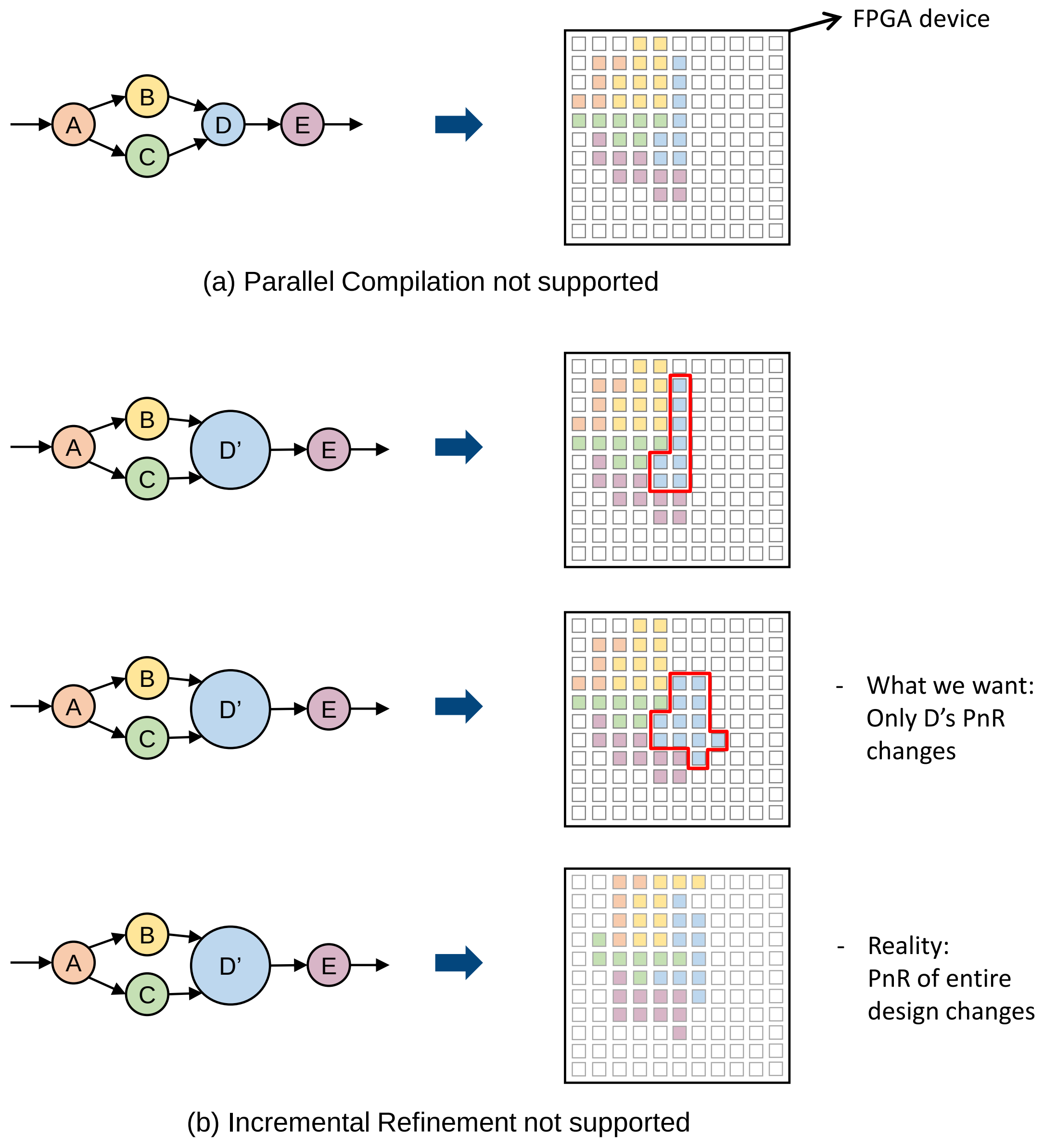
(c) Step 3 – Incremental Refinement

(d) Step 4 – Incremental Refinement

(e) Step 5 – Incremental Refinement

(f) Step 6 – Incremental Refinement

Figure 3.1: Software Development Example


FPGA device
A
B
C
D
E
D’
- What we want: Only D’s PnR changes
- Reality: PnR of entire design changes

(a) Parallel Compilation not supported

(b) Incremental Refinement not supported

Figure 3.2: FPGA (Hardware) Development Example

A, B, C, D, and E in Figure 3.2 are operators of a dataflow application written in synthesizable RTL or HLS. Small rectangles in the FPGA device shown in Figure 3.2 represent primitives like LUTs, FFs, BRAMs, or DSPs, and used elements are colored with the same color as the corresponding operators, A–E. The logic synthesis phase can be parallelized with a modern vendor tool like AMD Vivado; each operator (A–E) can be synthesized in parallel, and the stitching process is not time-consuming. Nonetheless, in place-and-route, FPGA vendor tools try to optimize the design as a whole (Figure 3.2 (a)). Unfortunately, parallel place-and-route is not supported by the commercial tool even if operators are independent of each other. In addition, users cannot selectively recompile one function, leaving the rest of the design to stay the same. For example, in Figure 3.2 (b), we refine operator D. We want to change the placement and routing of only the changed function (D), actively reusing the previously compiled results to reduce iterative compilation time, similar to software compilation. Nevertheless, with today's vendor tool, the entire design is monolithically recompiled, leading to a long compilation time for every modification to the design. The bottom case of Figure 3.2 (b) illustrates the monolithic compilation of the modern FPGA tools, and the placement of every function is slightly different from the first PnR result in Figure 3.2 (a) even if D is the only function that has changed.

3.2. Previous Work

In this section, we review the related work on split FPGA compilation including our works that utilize PR and others that utilize RapidWright [63].

3.2.1. Using Partial Reconfiguration

Related Work

[79] is the pioneering work that first proposes separate compilation strategy using a pre-routed packet-switched NoC and PR. We specifically adopt deflection-routed Butterfly Fat Tree (BFT) for the NoC as it is shown to be lightweight and efficient on FPGA [48]. As stated in Section 1.3, the main idea is to divide a large problem into multiple smaller problems. An application is decomposed into multiple operators, and each operator is mapped on a PR page along with the NoC interface. In [79], an array of 31 MicroBlaze processors [102] is chosen as a case study. We have three different

configurations of MicroBlaze processors that range from around 1400 LUTs to 2700 LUTs, and the size of PR pages is around 3900 LUTs. Our framework enables 31 Vivado compilation runs in parallel (one compile run per each MicroBlaze processor), and the compilation time of our system is the maximum of the 31 parallel runs. As our approach explicitly uses more cores to compile the same design, it takes only 418 seconds to implement (placement, routing, and bitstream generation) the array of 31 MicroBlaze processors while the vendor tool's monolithic compile takes 1797 seconds to implement the design, which is more than 4× longer than our approach in the implementation phase. This work is followed by [99] with characterization of Vivado compilation times and more realistic applications from Rosetta benchmarks [109]. The size of benchmarks ranges from 11472 LUTs to 78381 LUTs, and the size of PR pages ranges from 5760 LUTs to 9120 LUTs. In [99], we can achieve up to 8.9× compilation speedup compared to the monolithic compilation of Xilinx Vivado. The reason why we do not achieve an order of magnitude compilation speedup is that compilation time is not perfectly linear to the size of the design. Also, reading a static design or a synthesized operator (`read_checkpoint`) in PR costs about 2 minutes of overhead in [99].

While a pre-routed NoC provides a complete virtualization for all the operators, it imposes limited bandwidth between operators. In [96], Xiao et al. improve bandwidth between operators by directly connecting operators through switch boxes. Extending [79, 99, 96], inspired by software compilation, Xiao et al. propose different compile options like `-O0`, `-O1`, and `-O3` for FPGA in PLD [98]. These options provide different FPGA design points that trade off between performance and compilation time. `-O0` is an option where users can quickly test their designs on RISC-V processors mapped on FPGA, and because the design runs on soft cores, `-O0` results in a design with low performance in exchange for seconds of compilation time. `-O1` is an option for a separate compilation approach with a pre-compiled NoC and PR presented in [79, 99]. This approach supports fast compilation, but the NoC could introduce a bandwidth bottleneck, potentially resulting in sub-optimal performance. `-O3` is equivalent to the vendor tool's monolithic compilation where operators are directly connected with each other. With `-O3`, there is no limited bandwidth from the NoC (`-O3`), but the compilation time is much longer. In a theoretical scenario, users may want to quickly (seconds) run their designs on the software cores with `-O0`. Then, they can use `-O1` to map the design on hardware (minutes, still much faster than monolithic compilation from vendor tools). For the final design, users can use `-O3` to remove the area and bandwidth overhead of the NoC for optimal performance.

[79, 99, 96, 98] are all top-down approach that uses PR. Users pay a cost upfront, the time to generate a static design in PR, or they can use the overlay generated by us. Once the users have an overlay that consists of a NoC and PR regions, a compilation for each operator is entirely decoupled from each other. When all the partial bitstreams are generated, they can be simply loaded along with the bitstream for the static design. This approach accelerates all the phases of FPGA compilation: HLS, logic synthesis, placement, routing, and bitstream generation.

NoC Interface

In [79, 99, 98], a NoC interface is compiled along with a user operator on a PR page. In a NoC interface, there is a single input port from the NoC and a single output port to the NoC, but an operator could have multiple input streams or output streams. A NoC interface includes registers to store information about where the input port is receiving data from and where the output port is sending data to. The input data to the NoC interface is parsed and stored in the appropriate reorder buffer to handle packets arriving out-of-order because of the deflection-routed scheme. The output packet is created with the source and destination information of the page, sequence bits for the reorder buffer, and the data for the computation. A NoC interface also has an arbiter unit that orchestrates the data from the multiple output ports of the operator to the limited output channel of the page.

The number of 36Kb BRAMs in a NoC interface scales with the input ports, I, and output ports, O. [3]

$$\text{36Kb BRAMs related to input ports} = 2 \times I + \sum_{i=1}^{I} \frac{INPUT\_WIDTH_i}{PAYLOAD\_WIDTH} \times 0.5 \quad (3.1)$$

$$\text{36Kb BRAMs related to output ports} = \sum_{i=1}^{O} \frac{OUTPUT\_WIDTH_i}{PAYLOAD\_WIDTH} \times 0.5 + 1 \quad (3.2)$$

[3]The implementation of the NoC interface has slightly changed over time. The Equation 3.1 and Equation 3.2 are based on [77], the most recently published work.

In Equation 3.1, $2 \times I$ is from two dual-port BRAMs for the reorder buffers. After the packets are reassembled in these buffers, the input data is transmitted in order through a FIFO, corresponding to 0.5 in Equation 3.1. $\frac{INPUT\_WIDTH_i}{PAYLOAD\_WIDTH}$ accounts for the differences in the payload size in the NoC packets and the width of the input data stream. In Equation 3.2, 0.5 is for the output FIFO, and $\frac{OUTPUT\_WIDTH_i}{PAYLOAD\_WIDTH}$ accounts for the differences in the payload size in the NoC packets and the width of the output data stream. The last +1 in Equation 3.2 is the arbiter FIFO that sends one output packet at a time when there are multiple output ports. The number of LUTs also scales with the number of input ports and the number of output ports. A NoC interface consumes a few hundred LUTs to a little over a thousand LUTs depending on the number of input ports and output ports [99]. Please refer to [99, 95] for more details of a NoC interface architecture. Major modifications in the NoC interface will be explained in this chapter and Chapter 4.

Out-of-Context Implementation Flow

Vivado's *Out-of-Context (OoC)* implementation flow is another candidate, other than PR, to support separate compilations in FPGA. In Hierarchical Design flow with OoC implementation [104], separate IPs are developed independently and later collected from the top level. Similar to PR, users define input and output ports (Partition Pins) for each module. With these ports defined, each module can be separately placed and routed. However, unlike PR, these routed designs are imported from the top level, and there is a final implementation that integrates them. Unfortunately, this seemingly simple top-level integration leads to a lengthy process, comparable to the runtime of a monolithic compilation runtime from scratch. This flow promotes a "team design" where separate IPs are developed by different teams and integrated later on. Hierarchical Design flow [104] may achieve a better operating frequency as Vivado can focus on optimizing a smaller problem independently. However, this flow is not suitable for reducing compilation time with a divide-and-conquer because of the time-consuming, final implementation.

3.2.2. Using RapidWright

Thomas et al. [91] challenge the long compilation time by copying and pasting a small identical Processing Unit (PU). They create *connectivity shells* and compile one PU for a slot. A connectivity shell is a pre-implemented, domain-specific infrastructure that provides empty slots for PUs to be replicated. Then, they use RapidWright [63, 62] to replicate routed PU to different slots. RapidWright is an open-source framework that enables design manipulations that are not supported or not trivial to achieve in AMD Vivado. In [91], authors reduce 80–120 minutes of vendor tool's compilation to about 10 minutes, including about a minute for RapidWright to perform a replication of PUs. The connectivity shell used in the evaluation has 180 PU slots, and the size of the PU slot is 960 LUTs.[4] The copy-and-paste approach is inherently limited to an application with identical PUs. Also, as authors mention in [91], for UltraScale+ device that has highly heterogeneous columnar resource distribution, their approach results in area wastage because the PU slots must be identical for one routed PU to be copied and pasted. [91] belongs to a top-down approach, as a connectivity shell is pre-generated, and then PUs can be inserted after being compiled. Bitstream generation is not accelerated because [91] generates a monolithic bitstream whereas bitstream generation is accelerated with the approaches using PR because they generate partial bitstreams.

Guo et al. propose RapidStream [41], a compilation framework using RapidWright. They also perform a divide-and-conquer, mapping processing elements to *islands* whose sizes are about 2×2 clock region and compiling them separately in parallel. *Anchor* registers are placed between islands to provide timing isolation. Authors stitch separately implemented islands together using RapidWright. They achieve 5–7× compilation speedup over the commercial tool and improve the max frequency up to 1.3×. Unlike previous works that use PR [79, 99, 96, 98] or [91], RapidStream undergoes a top-level stitching process as it employs a bottom-up approach. In evaluation, the stitching process using a fast open-source router [110], which alone takes about half an hour, implies that this step could be a potential bottleneck to the compilation speedup. The size of the island (more than 50K LUTs) is bigger than the size of the PR pages (about 4K LUTs in [79], about 8K LUTs in [99], about 20K LUTs in [98]) in [79, 99, 98]. Large islands or large pages diminish the benefit of separate compilations, and in a bottom-up approach like [41], small pages could lead to an even longer stitching phase. In [40], authors show their work-in-progress effort to adopt PR to

[4]Authors report each PU slot has 30×4 logic slices. Because AMD UltraScale+ device has 8 LUTs in a slice, each PU is expected to contain 960 LUTs available.

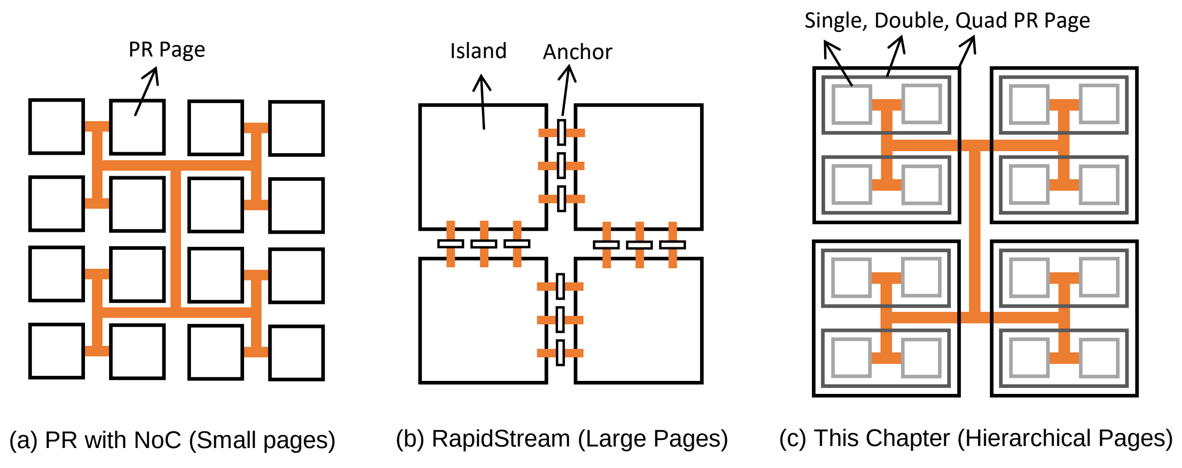

PR Page
Island
Anchor
Single, Double, Quad PR Page
(a) PR with NoC (Small pages)
(b) RapidStream (Large Pages)
(c) This Chapter (Hierarchical Pages)

Figure 3.3: Comparison between Related Work

reduce the time spent in the stitching phase. RapidStream does not accelerate bitstream generation because it generates a monolithic bitstream.

3.2.3. Problems with Previous Work

A common challenge from previous work on separate FPGA compilation in Section 3.2.1 and Section 3.2.2 is that the operators are mapped in the *fixed-sized* pages. [79, 99, 98] belong to Figure 3.3 (a). Authors create small PR pages to achieve better speedup in the separate compilation strategy. However, it is the user's responsibility to manually divide an application into small operators. To use the fast compilation framework with the fixed-sized pages, the users need to have some idea of how operators are synthesized and implemented because if operators require more resources than those available in the PR pages, operators cannot be mapped. This careful decomposition prevents the users from rapidly testing their designs on hardware. This limitation also makes it difficult for HLS developers without a hardware background to use our fast compilation. Some application needs to be *unnaturally* decomposed to be mapped on small pages. Unnaturally decomposed operators refer to operators that logically belong to a single function but are decomposed to fit small PR pages. Such decomposition could lead to excessive communication over the NoC.

RapidStream [41] belongs to Figure 3.3 (b). RapidStream uses larger islands, relieving the user's responsibility to decompose a design into fine-grained operators. Nonetheless, large islands limit

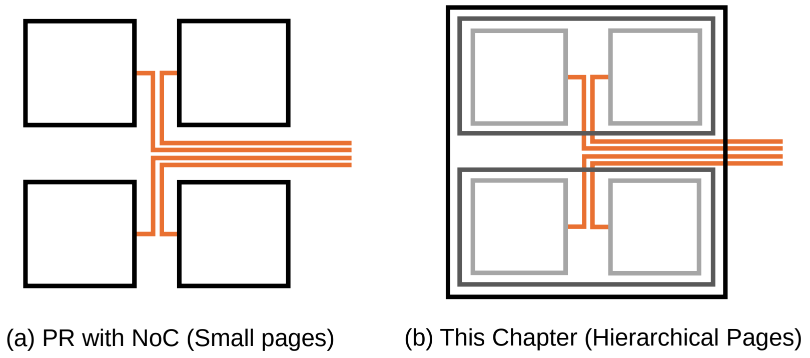

(a) PR with NoC (Small pages)
(b) This Chapter (Hierarchical Pages)

Figure 3.4: Hierarchical Pages, zoomed in
Note: Four single-sized pages are shown in both cases.

the compilation speedup in parallel compilation strategy. Furthermore, the global stitching time is already non-negligible and is expected to grow if the size of the islands decreases.

[79, 99, 98] use PR and does not have the global stitching process, but fixed-sized PR pages make it difficult for users to use the fast compilation framework. An approach using RapidWright to stitch separately compiled islands could provide more flexibility in the granularity of separate compilation, but small islands will likely to result in a long stitching phase. Therefore, we want to use PR with *variable-sized* pages. If a design can be regularly decomposed into small operators, we will use small PR pages for all operators. If it is natural to have a large operator, we will use a large PR page for the operator.

3.3. Variable-sized Pages using Hierarchical PR

To support variable-sized PR pages, we use Hierarchical PR [14], which was first introduced in AMD Vivado version of 2020.1. Previous works using PR to support separate compilation [79, 99, 96, 98] use a single level of PR, but in this work, we create multi-level PR pages to provide more flexibility. Using Hierarchical PR, we create a separate compilation framework whose small PR pages can be recombined to form larger PR pages. Based on post-synthesis resource utilization estimates, our framework assigns the appropriately sized PR page for each operator, significantly reducing the user's burden of designing operators to fit in fixed-sized pages. While users can enjoy the benefit of fine-grained separate compilation with single-sized pages so that they can achieve high compilation speedup, users are not forced to decompose a design into small operators because single-sized pages can be recombined to offer double-sized pages or quad-sized pages. The new separate compilation system using Hierarchical PR is shown in Figure 3.3 (c). Figure 3.4 compares fixed-sized pages and variable-sized pages. We first create a larger, upper-level PR page and *subdivide* the page into smaller pages. Both Figure 3.4 (a) and Figure 3.4 (b) have four single-sized pages in the subtree, but in Figure 3.4 (b), these single-sized pages can form two double-sized pages or one quad-sized page.

Figure 3.5 illustrates the flow of the new separate compilation framework with variable-sized PR pages. We build upon [98], and our framework is fully compatible with AMD Vitis acceleration flow [12]; the interfaces between the host and the kernels are abstracted out, and with only HLS source codes and the host code, an application can run on the hardware. Each operator should have its own HLS source file, and as shown in Listing 3.1, an operator should have inputs and outputs defined in Vitis HLS Streams, `hls::stream` [13]. Then, our framework launches separate HLS runs with Vitis HLS and separate logic synthesis runs with Vivado as done in [79, 99, 98]. The difference is that once all logic synthesis runs are finished, our *page assignment* algorithm assigns an appropriate page to each operator based on each operator's post-synthesis resource estimates, including expected utilization of LUTs, BRAMs, and DSPs. Then, a synthesized netlist for each operator can be placed and routed on the assigned PR page, and partial bitstreams are generated in the `xclbin` file format [12] accordingly.

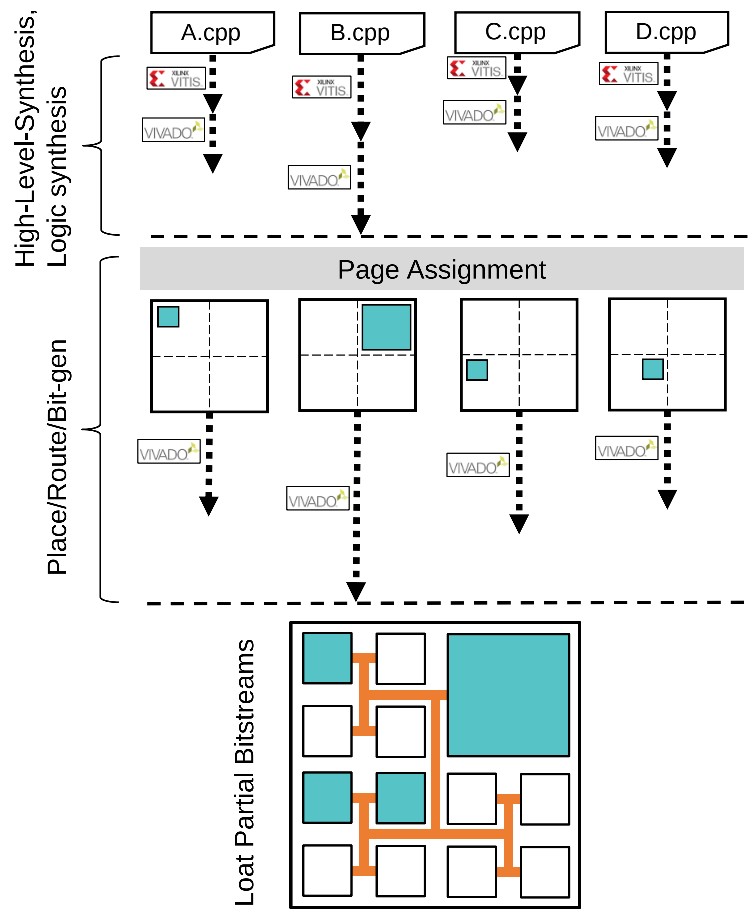


Figure 3.5: Separate Compilation with Hierarchical PR Pages

```
void sample (
    hls::stream<ap_uint<64>> & Input_1,
    hls::stream<ap_uint<32>> & Input_2,
    hls::stream<ap_uint<128>> & Output_1,
    )
{
#pragma HLS INTERFACE axis register port=Input_1
#pragma HLS INTERFACE axis register port=Input_2
#pragma HLS INTERFACE axis register port=Output_1

    // your code

}
```

Listing 3.1: Example of an HLS Source Code for an Operator, `sample`, with Streaming Interfaces

The page assignment algorithm in this chapter ([78]) is as simple as (1) sort operators in the descending order of their size and (2) pick the smallest page that accommodates each operator. We define the size of an operator as $LUT_{op}/LUT_{total} + BRAM_{op}/BRAM_{total} + DSP_{op}/DSP_{total}$, when LUT_{total}, $BRAM_{total}$, and DSP_{total} refer to the total each resource available in the device. This simple mechanism can reduce both external fragmentation and internal fragmentation. By prioritizing the mapping of a larger operator, we can prevent two small operators from being assigned to separate single-sized pages with different double-sized parent pages, which would otherwise block the larger operator from obtaining a valid double-sized page mapping (external fragmentation). By assigning to the *tightest* page to an operator, we minimize the unused space within the page, leaving as much space as possible to other operators that still need to be mapped (internal fragmentation). This *capacity-based* page assignment, however, does not consider the locality of operators. Later in Section 4.5, we introduce a page assignment algorithm based on graph bi-partitioning. Also, in Section 4.6, instead of using hard constraints to determine whether a netlist could be successfully mapped or not, we train a classifier to make a decision, hoping to reduce internal fragmentation.

Similar to [99, 98], the user inputs include the graph file for the application in addition to a source code per operator. The graph file is a simple wrapper function that users use to instantiate all the operators; users would need this file for their design regardless of using our framework or not. Our script parses the wrapper file, and it uses the interconnection information in the page assignment step and when configuring destination page and source pages in the NoC interfaces.

We evaluate our new fast and flexible FPGA compilation framework with AMD Vitis Embedded platform [12], equipped with ARM Processing System. The host code running on the ARM core is responsible for sending data to and receiving data from the programmable logic. In our framework, an application starts with the NoC configuration packets sent from the host to each page. These packets configure registers in the NoC interfaces so that all input and output ports of all the pages know where to send the data to and receive the data from. Then, the host sends the data for the computation to a DMA operator, one of the operators that is responsible for receiving data from the host and sending the computation results back to the host. When the host receives all the data back after computation, the program ends, and we report the application latency, defined as the runtime from the start to the end of the application for a given set of input data.

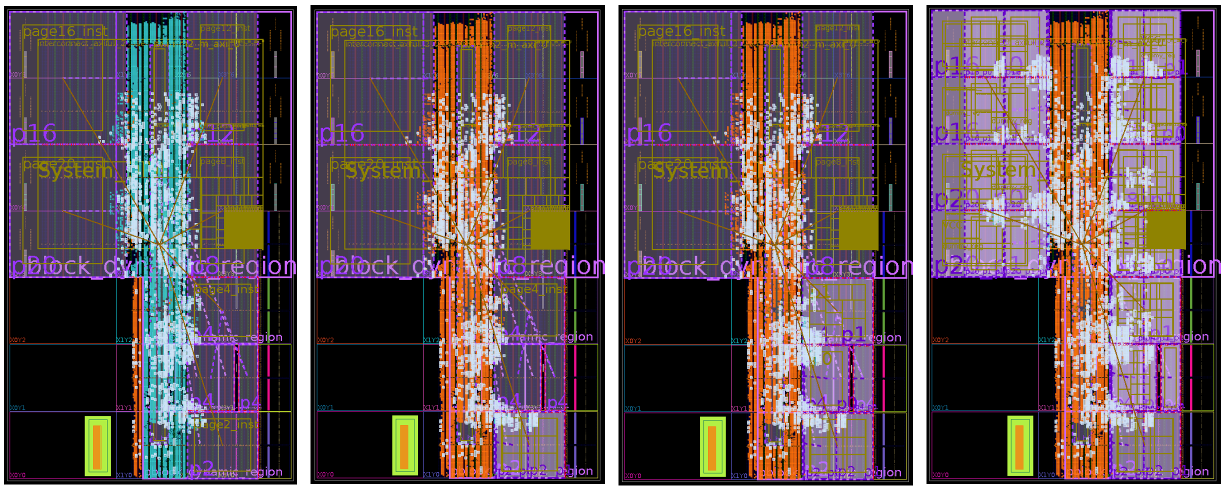
In this work [78], we support source codes of HLS. However, because our framework uses HLS-generated RTL sources for the inputs for Vivado's logic synthesis, RTL sources can be easily supported with minor modifications in the tool flow.

3.4. Engineering Details

As application engineers, users who want to enjoy the fast compilation with our framework do not need to know how to generate the static overlay. They only need to provide an HLS source code per operator, and our framework automatically compiles each operator separately in parallel. Nevertheless, for those who are interested in generating a custom overlay for separate compilation and our experience with Nested DFX [14], we share engineering details in this section.

3.4.1. Overlay Generation and Nested DFX

AMD Embedded DFX Platform and AMD Alveo Data Center Platform [103, 6] use PR to abstract out implementation details on data transfer from the host to the programmable logic. These platforms consist of a dynamic region for the user kernel logic and a static region where a pre-routed interconnect or peripherals are mapped. Creating a PR page on these platforms already requires the use of Hierarchical PR (Nested DFX) because we would be creating a second-level PR region (level



(a) First subdivision (b) Second subdivision (c) Third subdivision (d) 17th subdivision

Figure 3.6: Static Design Generation with a Sequential Subdivision in Nested DFX Technology

2 PR regions) inside a dynamic region (level 1 PR region). For example, PLD [98] uses Nested DFX technology to create PR pages in the dynamic region of AMD Alveo U50 Data Center platform [6]. In this chapter ([78]), we take a step further to create deeper-level PR regions and recombine them when necessary to provide flexibility in the sizes of PR pages.

In AMD's Nested DFX technology, a PR region is subdivided into smaller PR regions, to which new modules are linked and implemented. It is important to note that when there are multiple PR regions to be subdivided, one PR region needs to be subdivided at a time [14]. For example, Figure 3.6 shows the screenshots of the routed design after each subdivision in our experiments. Figure 3.6 (a) shows the routed design after the first subdivision. The orange-highlighted cells, narrowly placed at the bottom, represent the static logic of the ZCU102 DFX platform. We modified the dynamic region of the officially released platform [103] to allocate more area for a user application. The cyan-highlighted cells represent our soft NoC and Vitis-generated logic that supports the host and kernel data transfer. We subdivide the large dynamic region into five quad-sized pages and one double-sized page, and after the first subdivision, newly created PR pages are 2nd-level DFX regions. Figure 3.6 (b) shows that one double-sized page is subdivided into two single-sized pages, and these single-sized pages are 3rd-level DFX regions. As stated earlier, we need to subdivide one PR region at a time. Even if we have 5 independent quad-sized pages to be subdivided, we subdivide one PR region and then move on to the next PR page. It is important to ensure that in Vivado, parent-children relationships for Nested DFX regions are set as desired. Figure 3.6 (d) shows the static design after 17 sequential subdivisions.

The numerous sequential top-level implementations required to generate the static design in the Nested DFX flow make it more challenging to create a static design compared to the single-level DFX flow. The implementation becomes more difficult as the depth of the DFX grows. Additionally, there is a risk that at least one of the sequential implementations may fail. In such a case, we should try manually routing the failed nets. We may need to try different constraints to guide the range of Partition Pins or different implementation directives. We may have to try different floorplanning for PR regions, starting over the sequential top-level implementations.

After we generate Figure 3.6 (d), we *recombine* subdivided PR pages, removing the lower-level reconfigurable partition definitions and restoring the parent reconfigurable partition definition. Then, we generate the partial bitstreams for the parent PR regions, the context partial bitstreams needed in Nested DFX flow. In Nested DFX flow, we need to load the partial bitstreams in the upper levels first to set up the context and then load the partial bitstreams in the lower levels. Figure 3.7 shows the example of loading multi-level bitstreams. Purple-ish blocks are recombined partial bitstreams necessary to set up the proper context. Orange and yellow blocks are partial bitstreams for user operators mapped on PR pages. A blue block indicates a NoC. Figure 3.8 shows the example of an application mapped with our new variable-sized PR pages. 9 operators are mapped on single-sized pages, and one large operator is mapped on a quad-sized page.

3.4.2. Abstract Shell

As discussed in Section 2.1, a larger size of static design leads to a longer compilation time in AMD Vivado's PR technology. Previously in [99], we characterize compilation time for the identical reconfigurable module on the same PR region, changing only the size of the static design. The size of the design mapped on the PR page is the same among all experiments, so the task size of Vivado implementation runs should be the same, placing and mapping the same sample design into the

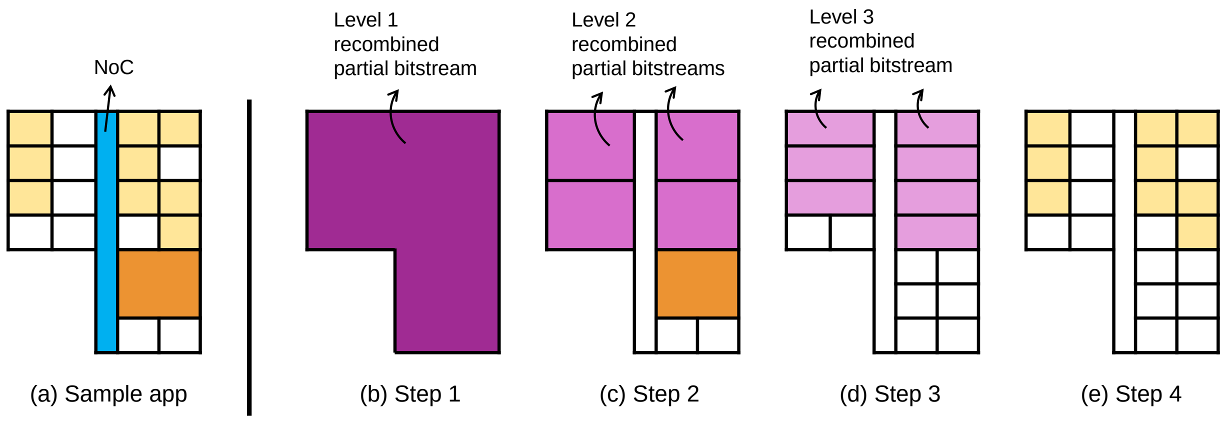

NoC
Level 1 recombined partial bitstream
Level 2 recombined partial bitstreams
Level 3 recombined partial bitstream
(a) Sample app
(b) Step 1
(c) Step 2
(d) Step 3
(e) Step 4

Figure 3.7: Loading Multi-level Partial Bitstreams

same size of the PR region. Therefore, we expect the compilation times for different experiments to be similar. Nevertheless, we notice that a larger static design leads to longer PR mapping time [99]. The workaround used in [79, 99] is to make the NoC reconfigurable and remove it from the static design to reduce the size of the static design.

Starting the 2020.2 version, Vivado supports Abstract Shell, which removes unnecessary static logic and static routing for each PR region. Figure 3.8 shows that each operator is mapped with a corresponding Abstract Shell, when the orange-highlighted cells indicate the static logic remaining in each Abstract Shell. In [78], however, we observe that the sizes of Abstract Shells (the sizes of static design remained in Abstract Shells) can be unbalanced even for the same quad-sized pages. Because Abstract Shells with large static logic lead to longer compile time as shown in [95], to balance the sizes of Abstract Shells, we, again, have a reconfigurable pblock for a NoC in [78]. Since the static logic cannot be placed in the NoC's pblock, the static logic is pushed out, balancing the sizes of Abstract Shells. Later in Section 4.4.1 ([77]), we observe that the problem with unbalanced sizes of Abstract Shells is mitigated if the PR page is not aligned with the clock region boundary as suggested in AMD PR user guide [14].

3.5. Evaluation

We evaluate our new separate compilation framework that supports variable-sized PR pages on AMD ZCU102 which uses UltraScale+ ZU9EG FPGA. For evaluation, in Section 3.5.2, we show

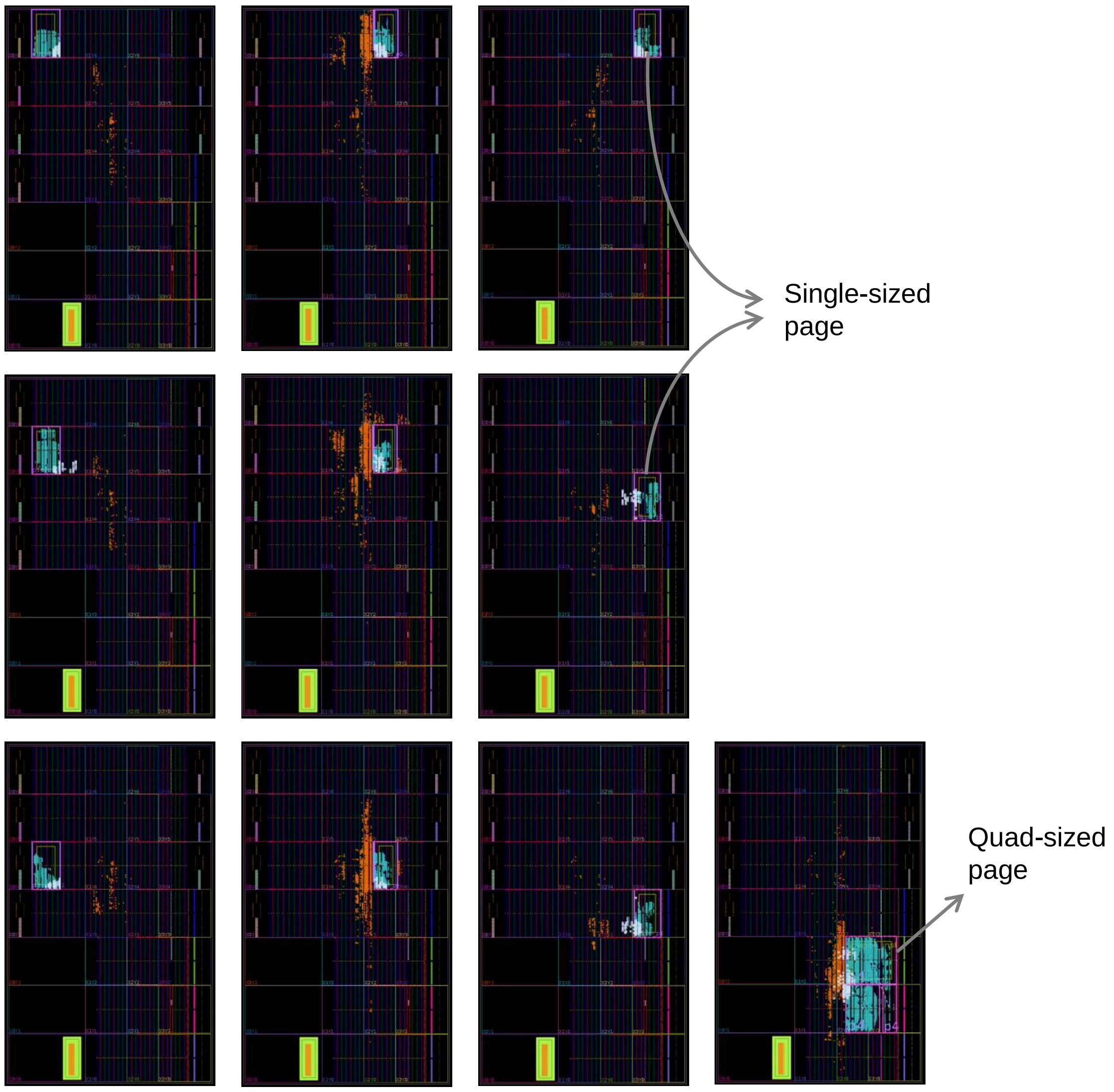


Figure 3.8: Optical Flow Application, Separately Compiled with Variable-sized Pages

how the variable-sized pages can be useful in incremental development scenario. In Section 3.5.3, we show how the variable-sized pages improve the application performance compared to the fixed-sized pages while compiling faster than the vendor tool's monolithic compilation. We use an officially released ZUC102 DFX Platform from [103] and modify the pblock for the dynamic region to reserve more area for the user application. The dynamic region available to the users consists of 264,464 LUTs, 1,752 18Kb BRAMs, and 2,448 DSPs. We use AMD Vitis 2021.1 including Vitis HLS and Vivado. We evaluate our framework on a workstation equipped with the 3.7GHz AMD Ryzen 9 5900X 12 Core CPU with 24 processing threads and 128 GB of RAM.

3.5.1. New Overlay with Hierarchical PR pages

Figure 3.6 (d) is the static design used in the experiment, and Table 3.1 shows the available resources for each PR page in our overlay used for the evaluation. The available resources represent the total resources available in the PR region minus the blocked resources due to static routing over the PR regions. % LUT, % RAMB18, and % DSP columns indicate the proportion of resources available in each PR region. The overlay consists of 22 single pages, (6933–8409 LUTs, 38–72 18Kb BRAMs, 47–72 DSPs). 22 single pages can be recombined to create 11 double pages, (14632–16001 LUTs, 102–116 18Kb BRAMs, 116–143 DSPs). 10 double pages can also be recombined to create 5 quad pages, (29899–31750 LUTs, 214–228 18Kb BRAMs, 233–282 DSPs). The reason why the pages even with the same size have different resources available is that AMD UltraScale+ FPGA device [105] has an irregular columnar resource distribution. In Table 3.1, some single-sized pages with three BRAM columns have more than 70 RAMB18s while some with two BRAM columns have less than 50 RAMB18s. Another factor for the heterogeneity in PR pages is the different amount of static routing over the PR regions, resulting in different amount of blocked logic resources and different amount of routing resources available. For instance, one single-sized PR page has only 77% BRAMs left because static routing over the PR region blocks 23% of them. The total available resources on PR pages are (64% LUTs, 70% BRAMs, 57% DSPs) of the device's dynamic region.

We use a deflection-routed BFT network [48] with Rent exponent [59] of p=0.5 as our NoC. The size of the packet is 49 bits, consisting of 1 bit of valid, 5 bits of address bits, 4 bits of input or output port bits, 7 bits of sequence bits for reassembly buffer, and 32 bits of data. The NoC consumes 11,799 LUTs. The number of PEs in the NoC is 32, but only 24 of them are used in the system, including 2 operators related to DMA operations and 22 PR pages for the user operators. About 36% = 100% - 64% of the dynamic region (about 95K LUTs) is left for the soft NoC and Vitis-generated auxiliary logic (35K LUTs together). The clock frequency for the NoC and the user operators is 200MHz in this chapter ([78]), but in Chapter 4 ([77]), we support 400MHz for the NoC and multiple clock frequencies (200–400MHz) for user operators. As explained in Section 3.4, the framework used in the experiment has 17 parent partial bitstreams (1 dynamic region, 5 quad-sized pages, 11 double-sized pages) in `xclbin` file formats. The total number of Abstract Shells is 38; 5 quad-sized pages, 11 double-sized pages, and 22 single-sized pages. In Section 7.7, challenges in creating a static design in the commercial PR technology will be discussed in detail.

Our overlay used in the evaluation has up to quad-sized pages. AMD DFX user guide [14] states that realistic designs should not exceed three levels of reconfiguration, but we achieve four levels of PR regions: a dynamic region, a quad page, a double page, and a single page. AMD DFX user guide also states that there is no formal limit in the number of levels in the Nested DFX technology, so we can even create octuple pages that could provide more flexibility to the users at the cost of more difficulty in generating the static design. This means that instead of subdividing the dynamic region into five quad pages and one double page in Figure 3.6 (a), we can subdivide the dynamic region into two octuple pages and one sextuple page. This adds another level of granularity, and the number of sequential top-level implementations required to generate the static design would increase from 17 to 20. The added level of granularity increases implementation complexity and raises the likelihood of failure in at least one of the sequential top-level implementations.

The size of single pages is about 7K–8K LUTs in the experiment. One reason why we do not provide finer-grained single pages is that we want single pages to have a little bit of everything: some LUTs, some BRAMs, and some DSPs. Because of the heterogeneous columnar distribution of the AMD UltraScale+ device, if we reduce the size of single pages to the range of 4K LUTs, some single pages would not have BRAMs or DSPs at all. Another reason is that AMD's PR technology does not

Table 3.1: Resources Available in Different Hierarchical PR Pages

Page Size	blocked LUT	blocked RAMB18	blocked DSP	available LUT	available RAMB18	available DSP	% LUT	% RAMB18	% DSP
Single	46	2	0	7898	46	72	99.42%	95.83%	100.00%
	11	2	0	7541	70	48	99.85%	97.22%	100.00%
	121	6	3	7623	42	69	98.44%	87.50%	95.83%
	27	4	0	6933	68	48	99.61%	94.44%	100.00%
	103	10	4	7721	38	68	98.68%	79.17%	94.44%
	2	0	0	7422	72	48	99.97%	100.00%	100.00%
	43	4	0	7781	44	72	99.45%	91.67%	100.00%
	12	2	0	7412	70	48	99.84%	97.22%	100.00%
	41	4	2	7703	44	70	99.47%	91.67%	97.22%
	8	2	0	7416	70	48	99.89%	97.22%	100.00%
	45	6	2	7779	42	70	99.42%	87.50%	97.22%
	4	2	0	7420	70	48	99.95%	97.22%	100.00%
	63	8	3	7817	40	69	99.20%	83.33%	95.83%
	23	10	1	7529	62	47	99.70%	86.11%	97.92%
	56	16	3	8408	54	69	99.34%	77.14%	95.83%
	33	0	2	7047	48	70	99.53%	100.00%	97.22%
	9	8	1	8271	60	71	99.89%	88.24%	98.61%
	8	0	0	6952	48	72	99.89%	100.00%	100.00%
	45	10	2	8275	60	70	99.46%	85.71%	97.22%
	18	2	2	6942	46	70	99.74%	95.83%	97.22%
	7	4	5	8409	64	67	99.92%	94.12%	93.06%
	0	0	0	7080	48	72	100.00%	100.00%	100.00%
Double	57	4	0	15519	116	120	99.63%	96.67%	100.00%
	152	10	3	14632	110	117	98.97%	91.67%	97.50%
	109	10	4	15203	110	116	99.29%	91.67%	96.67%
	55	6	0	15257	114	120	99.64%	95.00%	100.00%
	51	6	2	15197	114	118	99.67%	95.00%	98.33%
	51	8	2	15261	112	118	99.67%	93.33%	98.33%
	89	18	4	15423	102	116	99.43%	85.00%	96.67%
	89	16	5	15959	104	139	99.45%	86.67%	96.53%
	17	8	1	15727	110	143	99.89%	93.22%	99.31%
	67	12	4	15709	108	140	99.58%	90.00%	97.22%
	7	4	5	16001	114	139	99.96%	96.61%	96.53%
Quad	261	20	7	29899	220	233	99.13%	91.67%	97.08%
	108	12	2	30532	228	238	99.65%	95.00%	99.17%
	141	26	6	30763	214	234	99.54%	89.17%	97.50%
	107	24	6	31717	216	282	99.66%	90.00%	97.92%
	74	16	9	31750	224	279	99.77%	93.33%	96.88%

Target device: AMD ZCU102 with UltraScale+ ZU9EG FPGA allow stacking reconfigurable regions within a single clock region. This means that if we want to create small PR pages, the shape is likely to be narrow and vertically high. While this shape does not violate any design rule check, the aspect ratio is not desirable, and we decided to create single pages that have 7K–8K LUT.

3.5.2. Incremental Development Scenario

Our vision is to quickly try the application on the hardware and incrementally refine the design. Previous works on separate compilation using PR [79, 99, 96, 98] show the feasibility of recompiling only the changed operators. However, challenges remain. First, if the naturally decomposed operators do not fit into the fixed-sized pages, we cannot use our framework to support parallel compilations. Second, unnaturally decomposed operators could exhibit excessive communication over the NoC, and we cannot simply merge them into a single operator if the size of the merged operator is beyond the capacity of the fixed-sized page. Lastly, the fixed-sized page limits the degree of optimization of an operator because the increased size of the operator may not fit the fixed-sized page. The variable-sized PR pages can solve all the problems. In this section, to illustrate the benefit of our new framework, we present an incremental refinement scenario for Optical Flow application from the Rosetta Benchmark Suite [109]. We will demonstrate how flexibility provided by variable-sized PR pages can support natural decomposition of an application and enable both inter-operator optimization and inter-operator optimization.

Natural Decomposition

Optical Flow is naturally decomposed into a set of operators connected with dataflow streams. However, *Compute flow* operator consumes more than 16K LUTs, which is about 60% of the LUTs of the design, and it cannot be mapped on a fixed single-sized PR page, which has 6–8K LUTs in [99, 96]. Therefore, the authors in [99, 96] refine the code with a mix of fixed-point computation and floating-point computation to reduce the LUT consumption. Also, because of the limited DSPs available in the single-sized pages, *Compute flow* is divided into two operators in [99, 96].

In this work, however, we can quickly decompose an application into the most natural form without substantial code refactoring. The natural decomposition of Optical Flow is shown in Figure 3.9 (a), and the decomposition is now almost identical to the one in the original diagram from [109]. The post-synthesis LUT utilization of *Compute flow* operator is about 16K LUTs, but now *Compute flow* can be mapped on a quad-sized page that consists of four single-sized pages. The other 8 operators can be mapped on single-sized pages. Step (1) from Table 3.2 corresponds to this natural decomposition of the design. Since Step (1) is the initial compilation of the design, there are 9 parallel compilations, and 261 seconds is the compile time for *Compute flow*, the one that takes the longest time to compile.

Inter-operator Optimization

One issue with natural decomposition mapped with our framework is the 192 bits of datawidth between *Outer product*, *Tensor_y*, *Tensor_x*, and *Compute flow*. Because the NoC used in the evaluation currently supports only 32 bits of datawidth, the application may suffer from the IO bottleneck. A simple resolution for the limited NoC bandwidth is to merge these operators into a single operator so that they do not communicate over the NoC. The application diagram after this inter-operator optimization is shown in Figure 3.9 (b), and Steps (2)–(4) correspond to this optimization. The quad-sized page is large enough to accommodate the merged *Compute flow* operator. As shown in Table 3.2, the number of incremental compile jobs in Steps (2)–(4) is 1 because we only need to recompile the merged *Compute flow*. The size of the largest operator in Steps (2)–(4) increases as more operators are merged to *Compute flow*. The application latency improves from 18.7ms to 14.7ms because 192 bits of data do not need to travel over the NoC but communicate within the page.

We can also split *Weight_y* and *Weight_x* into three operators (Steps (5), (6) in Table 3.2). In these steps, the incremental compilation takes only 1–2 minutes since only single-sized pages are recompiled. Figure 3.9 (c) is the dataflow graph after Step (6).

Intra-operator Optimization

Intra-operator optimization refers to an optimization within an operator. For example, users may want to unroll some loops in one function or change the data type of some variables. This optimization could result in more resource consumption, and fixed-sized pages may not accommodate the

Table 3.2: Optical Flow Incremental Development with Hierarchical PR Pages

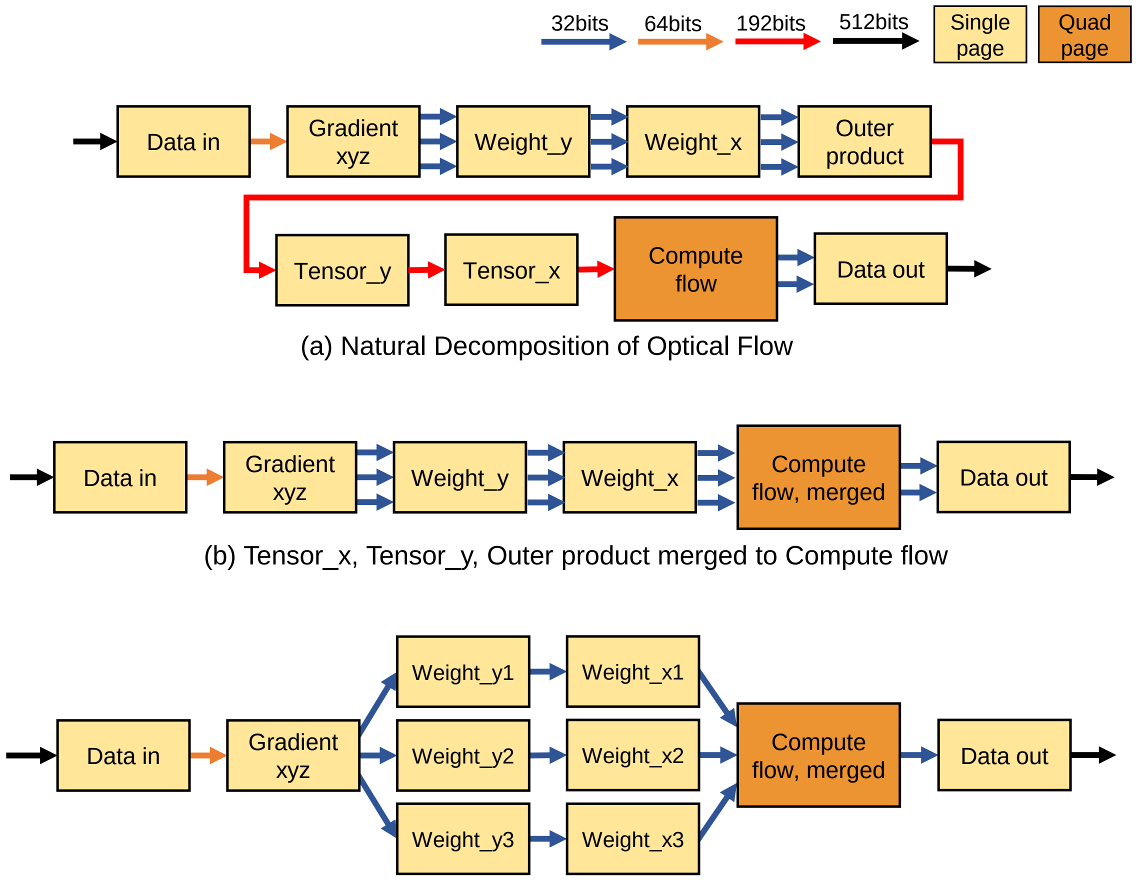
Steps	Largest Operator	Page Size	Compile Jobs	Largest Op Resource Usage			Incr Compile T	App Latency
				LUTs	B18s[†]	DSPs		
(1) Natural	Compt flow	Quad	9	16829	7	24	**261s**	18.7ms
(2) Merge Tensor_x	Compt flow	Quad	1	17560	7	54	**245s**	18.1ms
(3) Merge Tensor_y	Compt flow	Quad	1	18473	33	68	**274s**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

[†] B18s: BRAM18s "App Latency" – application execution time per input

new design. Nevertheless, the variable-sized pages using Hierarchical PR can provide appropriate sizes of PR pages by recombining multiple small pages. Table 3.2's Step (7) is one example of intra-operator optimization that we increase the parallelization factor in *Tensor_y* and *Tensor_x* within *Compute flow* operator. In Step (8), we change the datawidth of `calc_pixel_t` from 64 to 96 to reduce the error rate of the application. In doing so, we partially use a floating-point computation because using solely the fixed-point results in 62K LUTs of *Compute flow*, and even our quad-sized pages in the current overlay cannot accommodate it. We refer to the 64 bits of `calc_pixel_t` with only fixed-point computations as *Optical, (64,fixed pt)*. We refer to the 96 bits of `calc_pixel_t` with a mix of fixed-point and floating-point computations as *Optical, (96,mix)*. Figure 3.9 (c) is still the dataflow graph after optimizations from Step (7) and (8) because Step (7) and (8) are intra-operator optimizations within *Compute flow*.

3.5.3. Experiment Results for Rosetta Benchmarks

To evaluate the compilation speedup and performance, we run the monolithic compilation for Rosetta Benchmarks with AMD Vitis. Resource usage, compile time, and application latency for monolithic compilations are shown in Table 3.3. The resource usage does not include the peripherals or interconnect between the host and programmable logic which are automatically generated by Vitis Acceleration flow. The resource usage includes only the logic used for the application. The compile time in Table 3.3 does not include the packaging step in the Vitis flow, which is typically around 20 seconds in our experiment environment. In Table 3.4, experiment results for the fixed-sized PR pages (those marked with *, note that these use only single-sized pages) and the



(a) Natural Decomposition of Optical Flow

(b) Tensor_x, Tensor_y, Outer product merged to Compute flow

(c) Split Weight_y and Weight_x and optimize Compute flow

Figure 3.9: Incremental Refinements for Optical Flow

Table 3.3: Rosetta Benchmarks with Monolithic Vitis Flow (200MHz)

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

† Par: Parallelization Factor

variable-sized PR pages are presented.

For the monolithic Vitis flow in Table 3.3, we use the exact same application code and host code as the originally released Rosetta Benchmark Suite [109]. For the fast compilation framework, we rewrite the code in a way that each application consists of operators with streaming interfaces as mentioned in Section 3.4. Thus, code refactoring could create some discrepancy between the original Rosetta Benchmark Suite (Table 3.3) and the application mapped on our fast, separate compilation framework (Table 3.4). This is why even the Step (6) in Table 3.2, the step before increasing the parallelization factor, has a slightly different application latency compared to the application latency of Vitis flow in Table 3.3. However, the results from Table 3.3 set the meaningful baseline to evaluate the compile time benefit of our framework's fast and incremental compilation.

Resource usage in Table 3.4 could be higher than resource usage in Table 3.3 because in the separate compilation flow, operators are compiled with NoC interfaces in the PR region. Figure 3.10 shows Rosetta Benchmarks mappings on the FPGA device with Hierarchical PR pages. In the remaining sections, we explain how Hierarchical PR pages not only provide flexibility in the operator mapping but also improve application latency compared to the fixed-sized pages and still compile faster than Vitis monolithic flow.

Optical Flow with Fixed-Sized PR Pages

The first two rows in Table 3.4 are the experiment results of the decompositions for the fixed-sized PR pages. As stated in Section 3.5.2, the users need to use a mix of fixed-point computations and floating-point computations to shrink the sizes of operators to fit in the single-sized pages. We refer to these decompositions as *Optical, (64,mix), Par=2* and *Optical, (96,mix), Par=2* when 64 and 96 are the datawidths of `calc_pixel_t`. Both *Optical, (64,mix), Par=2* and *Optical, (96,mix), Par=2* have a parallelization factor of two, optimizing *Tensor_y* and *Tensor_x* as done in Step (7) in Table 3.2 All 17 operators are mapped on single pages. However, even after increasing the parallelization factor, the application latency is worse than Vitis monolithic flow (Table 3.3) because of the limited NoC bandwidth.

Optical Flow with Hierarchical PR Pages

The third and fourth rows in Table 3.4 represent Optical Flow decompositions (*Optical, (64,fixed pt), Par=2* and *Optical, (96,mix), Par=2*) with the Hierarchical PR pages. They are the incrementally developed versions from Section 3.5.2. The application latencies of 8.8ms and 8.8ms are slightly different from 8.7ms and 8.7ms in Table 3.2. This is because the page assignment for the incremental development and the page assignment from scratch are different. As we mitigate the limited NoC bandwidth bottleneck issue by merging the problematic operators, we achieve 3.6× and 4.94× reduction in the application latency compared to the fixed-sized page system. Compared to the monolithic Vitis flow, *Optical, (64,fixed pt), Par=2* and *Optical, (96,mix), Par=2* achieve 2.2× better application latency thanks to optimizations like increasing unroll factor. The compilations are still 2.2× and 2.3× faster than the monolithic Vitis flow.

3-D Rendering

We increase the data parallelism by splitting computationally heavy operators into data parallel operators, and one operator in *Rendering, PAR=2* is mapped to a double page. Our framework achieves 1.4× reduction in the application latency compared to the framework with only single pages. Compared to the monolithic Vitis flow, *Rendering, PAR=2* improves application latency by 1.4× by increasing the parallelization factor while compiling 2.5× faster than the monolithic Vitis flow.

Digit Recognition

We have 10 operators that compute K-Nearest-Neighbors (KNN) in parallel. From *Digit Rec, PAR=40* to *Digit Rec, PAR=80*, we increase the unroll factor (from 40 to 80) in these operators.

All the operators that were previously mapped in single pages are then mapped in double pages because of the increased BRAM usage. This optimization leads to 1.5× reduction in the application latency compared to the framework with only single pages. Our framework compiles 5.3× faster than the monolithic Vitis flow.

While *Digit Rec, PAR=80* (from PAR=40 to PAR=80) achieves 1.5× the performance improvement with the Hierarchical PR pages (Table 3.4), the performance improvement in the monolithic Vitis flow is only 1.1× (Table 3.3). KNN algorithm consists of the Hamming distance computation and KNN vote computation. While increasing the unroll factor accelerates the Hamming distance computation, the workload for KNN vote computation increases too, becoming the new bottleneck for the monolithic flow. In the PR page decompositions, KNN vote computation is distributed in 10 operators along with the Hamming distance computation, and KNN vote computation does not become a bottleneck.

Spam Filter

We increase the unroll factor from 32 to 64 in *Spam Filter* application, and 7 operators are mapped to double pages because of the increased DSP usage. However, we do not improve the application latency, and we believe that this is because of the DMA bottleneck in our experiment. We transfer data from the host to the DMA operator first, and the DMA operator sends data to another operator through the NoC, which has only 32 bits of datawidth. We can resolve the limited NoC bandwidth between operators by merging those operators as done in Optical Flow. Nevertheless, because the DMA operator is in the framework's static design, we cannot merge it with other operators. This is *not* an inherent problem with our approach but a problem specific to our implementation of the current framework.

Table 3.4: Rosetta Benchmarks with PR Pages

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

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

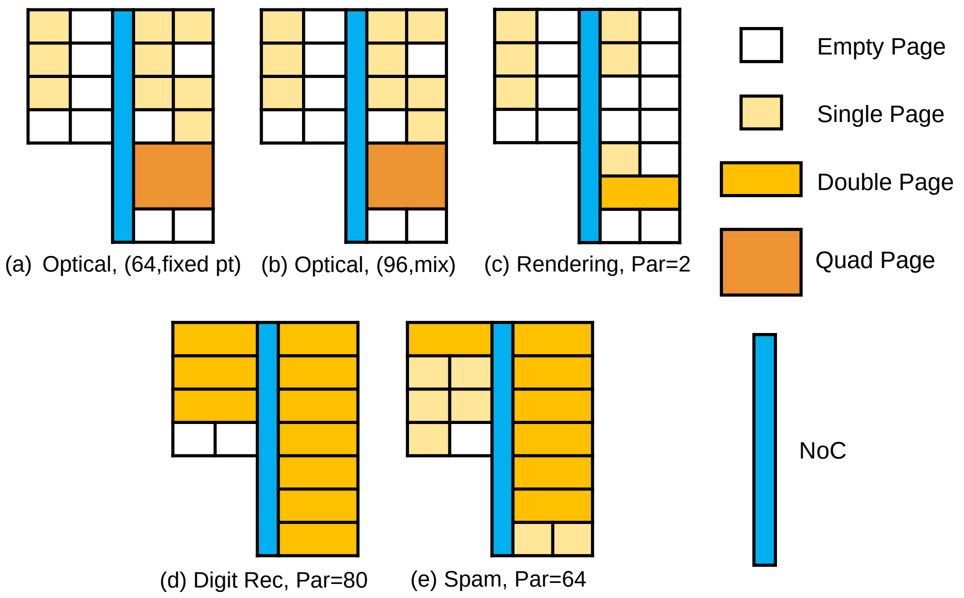


Figure 3.10: Rosetta Benchmarks with Hierarchical PR Pages Mappings

3.5.4. Bitstream Loading Time Overhead

One advantage of PR in many traditional uses is the short loading time of the partial bitstream compared to that of the full bitstream. But in our framework, the bitstream loading time is often larger than the monolithic bitstream loading time. As explained in Section 3.3, to load a lower-level bitstream, we need to load the upper-level context bitstreams first. Table 3.5 shows the bitstream loading time for both the monolithic Vitis flow and our framework using Hierarchical PR. “Dynamic Region” column refers to the first level partial bitstream that contains some peripheral logic for Vitis flow, the NoC, and the context information. “Dynamic Region” corresponds to Figure 3.7 (b)’s level 1 partial bitstream. “Total” column refers to the loading time for the first level partial bitstream,

Table 3.5: Bitstream Loading Times for Full Applications

Benchmarks	Monolithic Vitis	Hierarchical PR pages	
		Dynamic Region (1st level)	Total
Optical, (64,fixed pt), Par=2	147.6ms	165.9ms	496.7ms
Optical, (96,mix), Par=2	147.6ms	166.1ms	334.8ms
Rendering, Par=1	153.1ms	167.1ms	319.2ms
Digit Rec, Par=40	176.9ms	166.2ms	463.2ms
Digit Rec, Par=80	211.6ms	166.4ms	522.5ms
Spam, Par=32	300.1ms	166.8ms	612.6ms
Spam, Par=64	466.0ms	165.5ms	619.6ms

remaining recombined partial bitstreams, and partial bitstreams for the operators.

We observe that there exists a significant overhead in bitstream loading time due to the upper-level context bitstream, but we can accept the hundreds of milliseconds of overhead to save minutes or hours of compilation time. In an incremental development scenario, it is possible to only load the changed partial bitstreams and related upper-level bitstreams, and the overhead is not as large as that of the full application in Table 3.5.

As stated in Section 2.1, we do not dynamically reconfigure a PR region on the fly. We use PR as a means to achieve separate compilation on FPGA. If we can create a full bitstream from a collection of partial bitstreams, in our case, it is acceptable to assemble generated partial bitstreams and load the full bitstream even if it requires reconfiguring the entire device.

3.6. Discussion

3.6.1. Overlay Generation

To provide different sizes of PR pages, in this work, we use Hierarchical PR so that smaller pages can be recombined to offer larger pages. An alternative is to stock a variety of overlays that have different combinations of different sizes of pages. When a refined operator cannot find a valid mapping in the current overlay, the page assignment algorithm can look for other overlays in the repository. Nevertheless, in this case, when one operator needs more resources and is mapped on a different overlay's larger PR page, all other operators need to be recompiled as well. We believe that a single static design with variable-sized pages using Hierarchical PR is more elegant. Another approach to preparing different overlays in the library is to have overlays with different NoC bandwidths. The NoC in the current overlay supports 32 bits of datawidth, but it is possible to have overlays with larger NoC bandwidths at the expense of using more resources for the NoC.

More enhancements on the separate compilation framework including supporting multiple clock frequencies and multiple NoC interfaces are introduced in Chapter 4.

3.6.2. Incremental Refinement

In Section 3.5.2, we have seen that a single page takes less than 2 minutes to compile. The benefit of separate compilation is that the tool only needs to compile for the portion that is changed, not the entire design.

In HiPR [97, 100], we also show the potential of fast incremental compilations using PR. However, because we do not employ a NoC, which supports communication between arbitrary sources and destinations, if the interconnection of operators changes as done in Steps (2)–(6) in Table 3.2, users need to generate a new static design with the new operator connectivity, leading to hours of compilation time. HiPR [97, 100] is more suitable to incremental compilations when the application development is mature, and it is certain that the interconnections of operators stay the same.

So far, we assume that users have insights into the inner state of their design and can identify the bottleneck. For instance, in Table 3.2's each step, we assume that we know what optimization we need to perform next. In Section 5.4, we explain how FIFO counters in the NoC interfaces can be used to systematically determine whether the limited NoC bandwidth causes the performance bottleneck or a slow operator causes the bottleneck.

3.7. Conclusions

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

CHAPTER 4

ENHANCEMENTS TO FAST COMPILATION FRAMEWORK

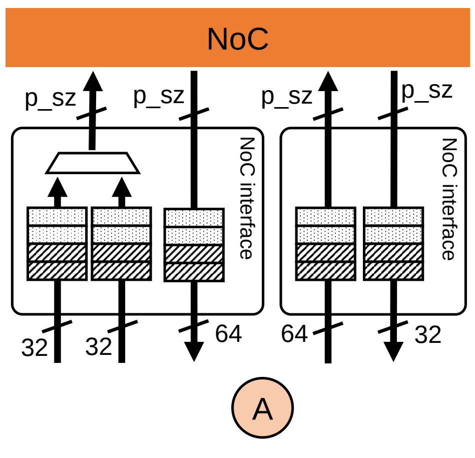
In this chapter, we explain the enhancements to the fast compilation framework with variable-sized pages from Chapter 3. The enhanced framework will be used in Chapter 5's incremental refinement. This chapter was previously published in [Dongjoon Park, and André DeHon. REFINE: Runtime Execution Feedback for INcremental Evolution on FPGA Designs. International Symposium on Field-Programmable Gate Arrays. 2024.] [77]. I led the project and was in charge of the system implementation.

4.1. NoC Bandwidth

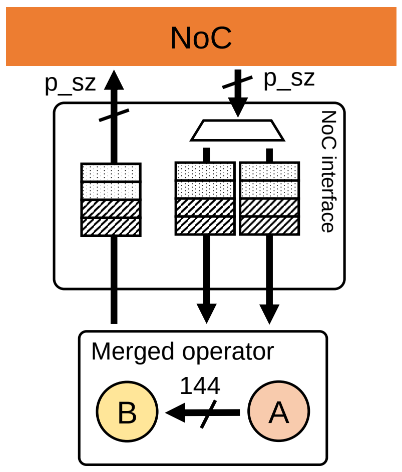
A main element in fast and flexible FPGA compilation from Chapter 3 is a NoC. Because a NoC supports communication from any source to any destination, our framework can support any application decomposed into a set of operators. While our *NoC-based* system supports fast and flexible FPGA compilation, the NoC provides only a limited bandwidth to operators. In Chapter 5, we will explain how we can identify whether the limited NoC bandwidth is the performance bottleneck in the current design point. Before that, in this section, we discuss two approaches to mitigate the NoC bandwidth bottleneck. These two approaches are the "knobs" that our tuner in Chapter 5 will explore when the NoC bandwidth bottleneck is detected.

4.1.1. Multiple NoC interfaces

In our fast compilation framework with variable-sized PR pages (Chapter 3, [78]), when a synthesized netlist is large such that the operator does not fit within a single-sized page, multiple single-sized pages can be recombined to create a larger double-sized or quad-sized page. Yet, in Chapter 3, the operator in the recombined PR page still uses a single NoC interface even after multiple PR pages are recombined. This limitation could lead to a NoC bandwidth bottleneck when the communication is heavy, so in this section, we add support for multiple NoC interfaces for recombined pages. Figure 4.1 (a) is the example that the user Operator A can now use two NoC interfaces when A has two input streams (32 bits, 64 bits) and three output streams (32 bits, 32 bits, 64 bits). The



(a) Multiple NoC interfaces



(b) Merging

Figure 4.1: Two Ways to Mitigate the Limited NoC Bandwidth Issue

p_sz: packet size

distribution of the streams to the multiple NoC interfaces is statically determined. We create all the possible combinations of the distribution and select the one that has the minimal standard deviation of sums of the stream widths. In simple words, in Figure 4.1 (a) case, we distribute two 32-bit output streams to one NoC interface and one 64-bit output stream to another NoC interface because the sums of the output streams per a NoC interface are equally 64. Larger datawidth does not always mean more data out or data in, so for accurate load balancing, we need to monitor runtime read and write rates for each stream. However, for simplicity, we statically assign input and output streams to proper NoC interfaces and do not change the assignment from then on.

4.1.2. Merging

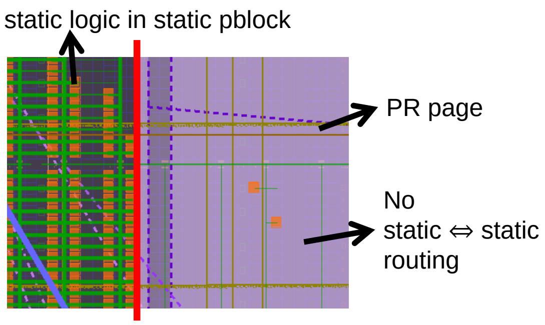
If a larger number of NoC interfaces does not help, we can simply merge two operators whose connection seems to suffer from the limited NoC bandwidth. In Section 3.5.2, we have already shown that merging operators with the problematic links improves application performance. Merging removes the NoC bandwidth limitation, at the cost of a larger page that will be slower to compile. In Figure 4.1 (b), two operators A and B are merged, and 144 bits of data packets then do not need to be routed over the NoC.

```
module leaf(
    input wire clk_200,
    input wire clk_250,
    input wire clk_300,
    input wire clk_350,
    input wire clk_400,
    input wire [48 : 0] din_leaf_bft2interface,
    output reg  [48 : 0] dout_leaf_interface2bft,
    input wire resend,
    input wire reset_400,
    input wire ap_start
    );

    // NoC interface is instantiated here

    // HLS-generated user operator module is instantiated here

endmodule
```

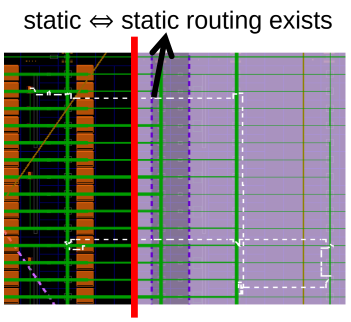
Listing 4.1: Automatically Generated RTL for a Single-sized PR Page

The NoC-based fast compilation framework from Chapter 3 has a single clock frequency for an application and the NoC. With a single clock frequency, if one operator's implementation can run only 200MHz not above, the rest of the operators should run at the same 200MHz even if they can run at higher clock rates. With the multiple clock frequencies, the operating clock is not limited by the lowest operating clock frequency among all the operators, and operators can run at different rates. Furthermore, in an incremental refinement scenario, when we want to increase the clock frequency for only one operator, we can recompile the operator that should run at the higher clock frequency instead of recompiling all the operators for the overlay with a higher clock frequency.

While in Chapter 3 ([78]), only 200MHz is supported for both the NoC and operators, we improve the system by running the NoC at 400MHz and supporting 200MHz, 250MHz, 300MHz, 350MHz, and 400MHz for operators. Therefore, now, there are 5 input clocks (`clk_200`, `clk_250`, `clk_300`, `clk_350`, and `clk_400`) in a page module as shown in Listing 4.1 while there is single input clock in [78]. The static design for the NoC-based system has 5 different clocks available for all the PR pages. When the user indicates that an operator runs at a specific clock frequency, our framework



(a) `CONTAIN_ROUTING true` for static pblock



(b) No static pblock

Figure 4.2: Snapshots of NoC-based System's PR Page

automatically generates a corresponding RTL like the one in Listing 4.1 that hooks the specific clock input (one of `clk_200`, `clk_250`, `clk_300`, `clk_350`, and `clk_400`) to the input clock signal of the operator module's instantiation. We use Xilinx Parameterized Macros [9] for Asynchronous FIFOs and other Clock Domain Crossing (CDC) circuits. `set_max_delay -datapath_only` constraint is automatically added in the implementation script to limit the distance of the clock domain crossing FFs.

4.3. Page Heterogeneity and Static Pblock

4.3.1. Page Heterogeneity

As discussed in Section 3.5.1, PR pages in the NoC-based system are heterogeneous, which means that even if they are the same single-sized pages, available logic and routing resources are different. Two reasons for the heterogenous PR pages are (1) irregular columnar resource distribution on modern FPGAs and (2) different amounts of static routing over PR regions. Heterogeneous PR pages mean that a netlist that can be successfully placed and routed in one single-sized page may fail in another single-sized page, and it complicates the page assignment algorithm which will be explained in Section 4.5.

4.3.2. Static Routing over PR Pages

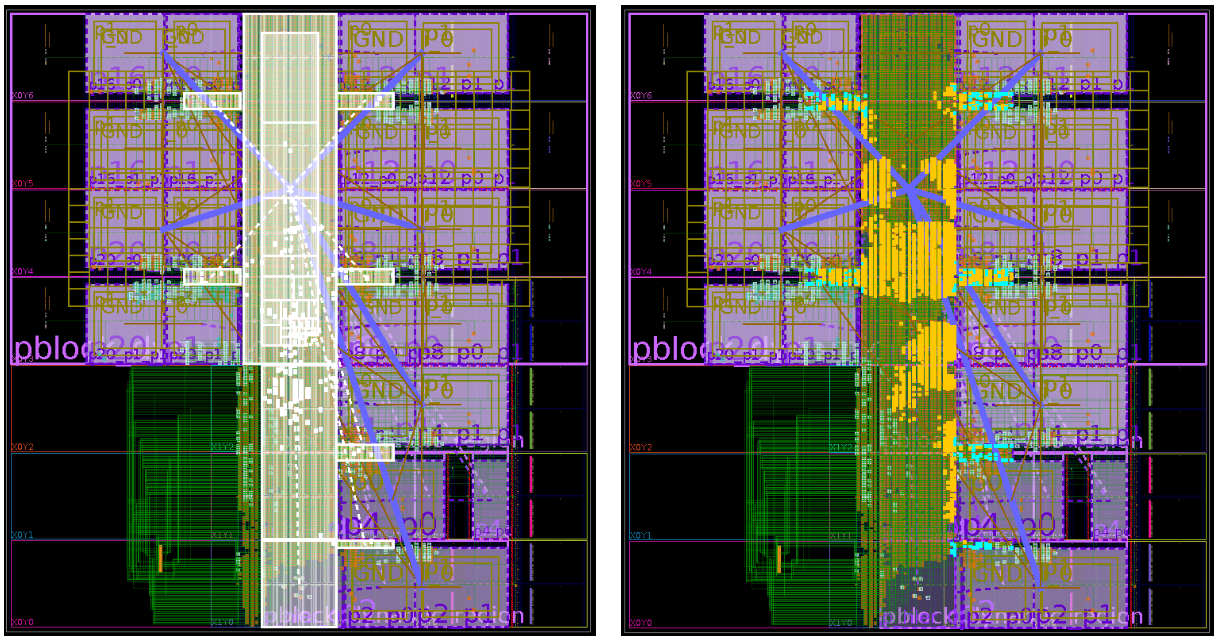
We reduce the effects of static routing over PR pages by creating a pblock for non-pages elements (the NoC, AXI interconnect, peripherals) and having `CONTAIN_ROUTING true` for the pblock. This requires Vivado to create a hierarchy in the block diagram for non-pages logic and assign the newly created cell to the pblock. With this setting, routing whose source is in static design and destination is in static design (selected in white in Figure 4.2 (b)) will be prevented from the reconfigurable regions. Figure 4.2 (a) shows the snapshot of a "clean" PR page when the static logic is added to a pblock that has `CONTAIN_ROUTING` property on. The remaining green routing on the PR page shown is static⇔reconfigurable routing or global clock signals. A similar technique is introduced in the AMD DFX user guide [14] with the section title of "Reduce Bleed Over of Static Nets to the Reconfigurable Pblocks". Since static routing over PR regions blocks resources in the PR regions, enabling `CONTAIN_ROUTING` property for the static pblock reduces the number of blocked resources. Table 4.1 presents the number of blocked resources and available resources in the PR pages of the new NoC-based system with the static pblock. The target device is AMD ZCU102 with ZU9EG UltraScale+ FPGA, the same device as Chapter 3. The same overlay will be used in the next chapter for incremental refinement. While a direct apples-to-apples comparison with Table 3.1 is inappropriate due to different floorplannings, Table 4.1 shows a significant reduction in blocked resources when `CONTAIN_ROUTING` property is enabled.

Although `CONTAIN_ROUTING true` constraint reduces the number of blocked resources, whether having the constraint for the static pblock leads to more resources available on the PR pages is questionable because `CONTAIN_ROUTING` definitely makes the generation of static design more challenging at the first place. One option is to create large PR pages and accept static routing over PR pages (`CONTAIN_ROUTING false`). Another option is to create slightly smaller PR pages and prevent static routing from PR pages (`CONTAIN_ROUTING true`). Since the first option allocates more resources to PR pages, even with some resources blocked, there is a chance that the first option leads to more resources in the PR pages. Nevertheless, having `CONTAIN_ROUTING true` constraint for static pblock leads to more regular PR pages, at least PR pages in the same columns that have the same columnar resource distributions.

Table 4.1: Resources Available in Different Hierarchical PR Pages (`CONTAIN_ROUTING true`)

Page Size	blocked LUT	blocked RAMB18	blocked DSP	available LUT	available RAMB18	available DSP	% LUT	% RAMB18	% DSP
Single	2	0	1	7438	64	87	99.97%	100.00%	98.86%
	1	0	0	7919	66	44	99.99%	100.00%	100.00%
	2	0	0	7302	64	88	99.97%	100.00%	100.00%
	0	0	0	7776	66	44	100.00%	100.00%	100.00%
	0	0	0	7264	62	88	100.00%	100.00%	100.00%
	0	0	0	7776	66	44	100.00%	100.00%	100.00%
	0	0	0	7304	64	88	100.00%	100.00%	100.00%
	2	0	0	7774	66	44	99.97%	100.00%	100.00%
	0	0	0	7264	62	88	100.00%	100.00%	100.00%
	0	0	0	7776	66	44	100.00%	100.00%	100.00%
	0	0	0	7440	64	88	100.00%	100.00%	100.00%
	2	0	0	7918	66	44	99.97%	100.00%	100.00%
	1	0	0	7839	44	88	99.99%	100.00%	100.00%
	1	0	0	7919	66	66	99.99%	100.00%	100.00%
	1	0	0	7615	44	88	99.99%	100.00%	100.00%
	1	0	0	7775	66	66	99.99%	100.00%	100.00%
	1	0	0	7695	44	88	99.99%	100.00%	100.00%
	0	0	0	7776	66	66	100.00%	100.00%	100.00%
	2	0	4	7742	44	84	99.97%	100.00%	95.45%
	2	0	0	7918	66	66	99.97%	100.00%	100.00%
Double	0	0	0	15400	132	132	100.00%	100.00%	100.00%
	1	4	1	14647	126	131	99.99%	96.92%	99.24%
	1	0	0	15119	132	132	99.99%	100.00%	100.00%
	0	0	0	15080	130	132	100.00%	100.00%	100.00%
	0	2	0	15120	130	132	100.00%	98.48%	100.00%
	0	0	0	15080	130	132	100.00%	100.00%	100.00%
	0	0	0	15400	132	132	100.00%	100.00%	100.00%
	0	0	0	15840	110	154	100.00%	100.00%	100.00%
	0	0	0	15472	110	154	100.00%	100.00%	100.00%
	0	0	0	15552	110	154	100.00%	100.00%	100.00%
	2	0	4	15742	110	150	99.99%	100.00%	97.40%
Quad	2	4	0	29806	260	264	99.99%	98.48%	100.00%
	0	0	0	30240	264	264	100.00%	100.00%	100.00%
	0	0	0	30520	264	264	100.00%	100.00%	100.00%
	0	0	0	31392	220	308	100.00%	100.00%	100.00%
	2	0	4	31374	220	304	99.99%	100.00%	98.70%

Target device: AMD ZCU102 with UltraScale+ ZU9EG FPGA



(a) Static pblock (`CONTAIN_ROUTING true`) selected

(b) Orange: NoC + FIFOs w/ almost-full, Cyan: pipeline registers

Figure 4.3: Screenshot of the New Overlay

4.4. Engineering Details in Overlay Generation

4.4.1. Floorplanning

In Chapter 3's NoC-based system, the NoC runs at 200MHz, so it was relatively easier to create a static design that meets timing. In this chapter, however, we run the NoC at 400MHz and feed in 5 different clock frequencies to PR pages (Section 4.2). We also have a static pblock that has `CONTAIN_ROUTING true` (Section 4.3), adding another burden in the routing phase. To successfully generate a static design that runs at 400MHz, we insert small FIFOs with an almost-full signal and pipeline registers [2, 96, 39] between the NoC and PR pages to isolate the timing closure of the NoC and PR pages. In this Latency Insensitive system [21], an almost-full is asserted a couple of cycles before the small FIFO is really full, and this delay corresponds to the number of pipeline stages for the registers. The placement of pipeline registers is constrained near the PR pages to provide guidance to Vivado. Figure 4.3 (a) shows the static pblock with `CONTAIN_ROUTING true` which contains the NoC and Vitis-generated peripherals. The orange-highlighted cells in Figure 4.3 (b) are the NoC and small FIFOs (about 13K LUTs together). The cyan-highlighted cells in Figure 4.3 (b) are pipeline registers that are placed near the PR pages. The remaining logic in the static pblock are AXI interconnect and peripherals (about 25K LUTs).

Another difference from the previous overlay in Figure 3.6 is that in Figure 4.3, we deliberately leave some space between the clock region boundary and PR pages as suggested in [14]. In the previous chapter ([78]), we report the unbalanced sizes of static logic for Abstract Shells. The workaround in [78] is to create a pblock for the NoC and indirectly adjust sizes of static logic for different Abstract Shells. When we leave some space between the clock region boundary and PR pages, this issue is mitigated, resulting in more regular Abstract Shells in terms of the size of static logic.

4.4.2. Placeholder Modules in Partial Reconfiguration

In both [78] and [77], we use almost empty placeholder reconfigurable modules when we generate a static design. Using empty modules is the opposite of the convention to use the most challenging modules. The reason why we use almost empty modules is that Vivado struggles to route the design with non-empty modules in the first place. Furthermore, when we use non-empty modules to generate static design, it turns out that more resources are blocked in [78]. We believe that one possible reason for more blocked resources is that empty modules result in more flexible locations of Partition Pins whereas non-empty modules have more constraints on where Partition Pins should be placed.

4.5. Page Assignment Based on Recursive Bi-partitioning

In the previous chapter ([78]), our framework synchronizes parallel compile runs after HLS and logic synthesis, performs a capacity-based page assignment, and launches parallel placement and routing. The algorithm in Section 3.3 first sorts the operators in descending order in their sizes. Then, it creates `possible_pblock_list` for the largest operator, which is the list of page candidates that are expected to safely accommodate this operator, and it selects the “tightest” operator to leave as much space as possible for other operators. This capacity-based page assignment could potentially lead to NoC congestion if logically adjacent operators are placed far apart from each other. For example, if two small operators, A and B, are directly connected to each other, it is intuitive to map Operator A to one single-sized page and map Operator B to another single-sized page both sharing the same parent double-sized page. Then, the data packets between the two need to hop a single switch to be transferred. However, the capacity-based page assignment prioritizes to reduce the internal fragmentation, possibly mapping two operators far apart from each other. As a result, the packets between the two operators may need to hop multiple switches, potentially causing NoC congestion with other packets.

In this section, we improve the page assignment algorithm so that it performs locality-aware page assignment and still finishes in less than a second. BFT NoC [66, 29, 48] is used in our framework [79, 99, 78, 77] because it is lightweight on FPGA. A simple heuristic-based approach to place processing elements on BFT nodes is to bi-partition the dataflow graph to *minimize the traffic between two partitions* and assign two partitions in different subtrees. We can recursively bi-partition and assign partitions in different subtrees until there is only one operator left in a partition. Figure 4.4 describes recursive bi-partitioning. The numbers inside the operators indicate "weights", the expected sizes of PR pages based on the operators' post-synthesis resource estimates.

We use `metis` software [54] to perform graph bi-partitioning. We first perform the capacity-based page assignment to assign the weights for each operator. At this point, we have a valid mapping, and weights are the sizes of the tightest PR pages for each operator from the capacity-based page assignment. After every bi-partition, we make sure that each partition is mappable to an assigned subtree. Because of heterogeneity in PR pages, a partition that is mappable to one subtree may not be mappable to another subtree with the same size. For example, one operator that has a weight of one cannot find a large enough single page in a specific subtree and needs a double page. If a partition is not mappable to a subtree, then we swap the mapping of partitions and subtrees. After recursive bi-partitioning, when there is a single operator left in a partition, instead of mapping the operator to the tightest page, we map to a larger page. For instance, if a single operator is left and is assigned to a subtree that has four single pages, then we simply assign this operator to a quad page. In the previous work [99], we observe that the size of the PR page does not affect the compilation runtime much, so it will not lengthen the compilation. When there is no valid mapping,

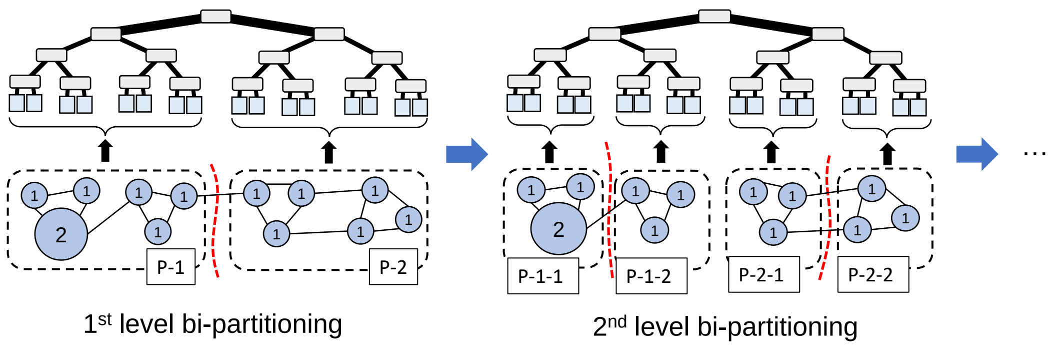


Figure 4.4: Page Assignment Based on Recursive Bi-partitioning

we return the capacity-based page assignment which was used to generate the weights. If there is no valid capacity-based page assignment, then we exit the page assignment, notifying the users or the automation script that there is no valid mapping.

In an incremental refinement scenario, the sizes of operators change. If the refined operators can still be mapped in the previous page assignment, then we use the previous page assignment. In this way, we can recompile only the changed operators, and this is one reason that we assign to a larger PR page instead of the tightest page when a single operator is left to be mapped to a subtree. If the previous page mapping is not expected to work, we perform a new page assignment from scratch.

The current page assignment algorithm statically assigns operators to pages. The current algorithm captures interconnection between operators so that neighboring operators can be placed next to each other, but it does not take dynamic read and write rates into account. For example, even if two directly connected operators rarely send and receive packets to each other, they are placed close to each other because they are directly connected in a dataflow graph. A smarter page assignment should analyze the communication among operators at runtime and assign operators accordingly. In the current NoC-based overlay, however, the number of singe-sized pages is only 22 and has a Rent exponent of 0.67 (note that the Rent exponent has changed from the previous chapter). In such a small BFT, page assignment has little effect on application performance. For a larger device that has 1M LUTs, our overlay can have more pages, and if the number of pages increases to 128 or 256, the impact of page assignment on application performance is expected to increase, making a better page assignment more desirable.

4.6. isFit classifiers

4.6.1. Motivation

A sub-task in page assignment problem is how to determine whether a synthesized netlist fits a specific PR page or not. A simple solution is to use capacity-based hard constraints. For instance, we can assume that the operator can be successfully placed and routed if the estimated LUT utilization is below X% of the available resources on a specific page. [34, 39] use such capacity-based hard constraints when floorplanning on FPGA. AMD also warns that routing could be challenging for the designs with LUT utilization over 80% of the entire device [101]. The potential problem with the approach based on hard constraints is that too conservative constraints lead to large internal fragmentation on the page and too aggressive constraints lead to an implementation (place/route) failure. Designs with different logic utilization and routing complexity would need different hard constraints. For example, one design may be successfully mapped on a PR page even if the post-synthesis LUT estimate is over 90% of LUTs available on the PR page, but another design with higher routing complexity may need more conservative constraints like 65%. Also, the two single-sized pages could have different resource distribution. They may have different resources because of heterogeneous columnar resource distribution or may have different routing resources because of different expanded routing regions. For these reasons, we train a classifier per PR page and let the classifier make a decision. If all LUT, BRAM, and DSP resource estimates are lower than 60%[5] of the resources available in the PR page, we assume that the netlist can be successfully placed and routed on the page. Otherwise, we use our *isFit* classifiers to make a decision.

4.6.2. Training and Testing

To generate training and test datasets for *isFit* classifiers, we generate a range of designs that have different resource ratios and routing complexity, run placement and routing on each PR page, and record whether the implementation has failed or not (Figure 4.5). We categorize the implementation

[5]For double-sized and quad-sized pages, the numbers are 60%, 60%, 60% for LUT, BRAM, DSP. For single-sized pages, the number are 60%, 50%, 50%. We also check the resource estimates of LUTRAM and FF are below the available resources (100%) in the PR region.

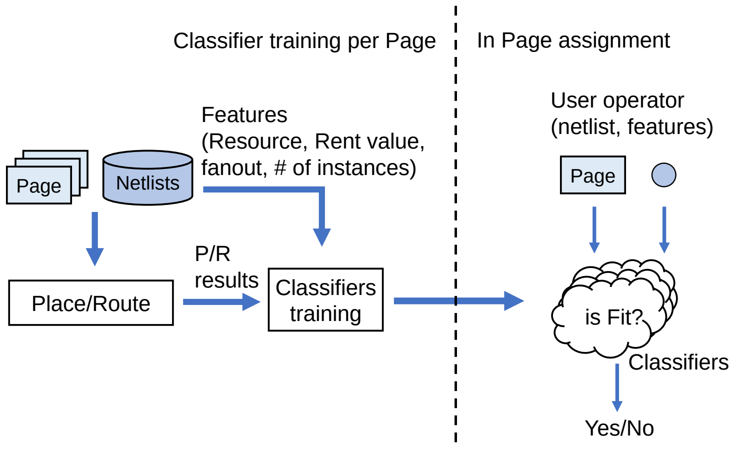


Figure 4.5: isFit Overview

results into three, “success”, “timing violation”, and “failure”. When labeling the datasets, we label only “failure” as positive and label “success” and “timing violation” as negatives, assuming “failure” is from lack of resources available in the PR page and “timing violation” can be improved with different implementation directives [7]. We use Rosetta benchmarks [109] to generate datasets. We traverse different parameter values for different benchmarks and use integrated top functions of multiple sub-modules to create diverse designs as done in [108]. Parameters include datatype, constant value, parallelization factor and storage type (LUTRAM or BRAM). Because our PR pages are the processing elements of the NoC, we include the NoC interface in the design when creating datasets. Features used in our classifiers are post-synthesis resource estimates (LUT, BRAM, and DSP), Rent value, average fanout, and `Total Instances`. After HLS and logic synthesis, for each design, we extract these features with Vivado’s `report_utilization` command and `report_design_analysis` command. We use Rent value, average fanout, and `Total Instances` from the complexity characteristics report because these are the metrics that are together related to the routing congestion according to AMD user guide [8]. For each PR page’s training and test data, we select netlists with over 60% of the PR page in LUT utilization. Since we train a classifier per a PR page, the number of classifiers is $\# \ of \ frequencies \times \# \ of \ total \ PR \ pages$.

The new NoC-based system in this chapter (Table 4.1) has 5 different frequencies (200MHz–400MHz) and has total 36 PR pages (single + double + quad), so the total number of trained classifiers is

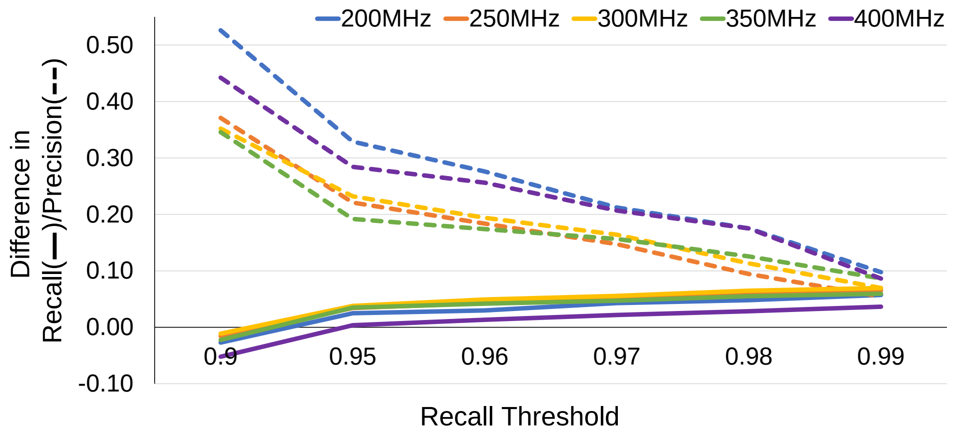


Figure 4.6: Difference in Recall and Precision between Our Trained Classifiers and Classifications Based on Hard Constraints

180. The average number of training and test datasets for each page in the experiment is 2573 (200MHz), 2730 (250MHz), 2660 (250MHz), 2532 (300MHz), 2364 (400MHz). We use Vitis HLS 22.1 and Vivado 22.1 to generate datasets for the classifiers. We use the scikit-learn library [81] to train and test the classifiers. We randomly select 80% of stratified data for training and 20% for testing. We use a Random Forest model for all the classifiers and perform a grid search on `n_estimators` and `max_features` to find the best hyper-parameter for each classifier.

In our case, False Positives (the classifier predicts that the netlist would fail in implementation, but it succeeds) are relatively acceptable. However, False Negatives (our classifier predicts that the netlist would succeed in implementation, but it fails) are not acceptable as they could require recompilation of the page, slowing down our fast compilation strategy. Therefore, we adjust our classifiers to at least match a target value of *recall* $(\frac{True\ Positives}{True\ Positives+False\ Negatives})$ and evaluate whether the classifiers still perform better than hard constraints in *precision* $(\frac{True\ Positives}{True\ Positives+False\ Positives})$. Figure 4.6 shows the difference in recall (solid line) and precision (dashed line) achieved by *isFit* over the hard constraint classifier. So, the difference in recall indicates *isFit*'s recall minus the hard constraint classifier's recall. The hard constraint classifier predicts implementation failure if one of the resources among LUT, BRAM, and DSP is over 70% of the PR page. The difference values are the average of all 36 PR pages. The X-axis shows the recall threshold to limit the number of False Negatives. As we set the recall threshold of 0.95 or higher, classifiers perform better than hard constraints in recall (difference in recall is positive), while still performing better (difference in precision is 0.19–0.33 when the recall threshold is 0.95) at classifications of False Positives than classifiers based on conservative hard constraints. Our page assignment algorithm based on graph bi-partitioning and *isFit* classifiers finishes in less than a second.

4.6.3. Limitations

A clear downside of *isFit* classifiers is a long training time. Our overlay in this chapter supports 5 different clock frequencies and has a total of 36 PR pages (single + double + quad). The average number of training and test datasets for each PR page is about 2,500. In our experiments, we have more than 20,000 synthesized netlists, and we run 450,000 = 2,500×180 placement and routing to create datasets, which contain features and labels of implementation results. Assuming we have a large compute server that can run 50 Vivado compile runs, each compute node is in charge of 9,000 = 450,000/50 Vivado implementation runs. Even if implementations for PR regions are shorter than an implementation for the entire chip, it should take 90,000 minutes = 62.5 days, assuming each implementation takes 10 minutes. Because 2500 training datasets are quite small, if we want to double the number of training datasets, it takes 135 days just to train classifiers. If we make a minor fix on the overlay and the floorplanning changes, then we probably need to re-train these classifiers. If the programmable logic is regular [35, 42, 25] and we completely remove static routing over reconfigurable regions, we can train one classifier per size. Then, we need only 3 classifiers (single, double, quad) per clock frequency, so we need only 15 classifiers. This leads to only 5 days of training with the same compute resources.

4.7. Conclusions

The support for multiple NoC interfaces or merging operators that suffer from limited bandwidth can close the performance gap between the design from our NoC-based separate compilation framework and the design from the monolithic compilation without the NoC. With the support for multiple clock frequencies, clean PR regions, and a smarter page assignment algorithm, we improve the quality of the mapped design with our fast separate compilation framework. In the next chapter, we will use this enhanced NoC-based system in a fast incremental refinement strategy.

CHAPTER 5

INCREMENTAL REFINEMENT AND BOTTLENECK IDENTIFICATION

In Chapter 3, we take a step forward to software-like incremental refinement by providing more flexibility to the users through the variable-sized pages. In this chapter, we propose a fast incremental refinement strategy utilizing our fast compilation framework. We also address a missing component from software-like incremental refinement: bottleneck identification. We introduce a systematic profiling capability based on FIFO counters. For evaluation, we compare the design tuning time of our fast incremental refinement strategy with that of the monolithic flow, both equipped with the bottleneck identification scheme. This chapter was previously published in [Dongjoon Park, and André DeHon. REFINE: Runtime Execution Feedback for INcremental Evolution on FPGA Designs. International Symposium on Field-Programmable Gate Arrays. 2024.] [77]. I led the project and was in charge of the system implementation.

5.1. Motivation

A separate compilation framework using PR [79, 99, 98, 78] supports fast parallel compilation and incremental compilation. One key element in this line of work is a soft NoC on top of the FPGA programmable logic. NoC provides virtualization between different operators with simple linking. Furthermore, a single static NoC overlay can support arbitrary designs unlike an application-specific overlay like HiPR [97, 100] where static design needs to be regenerated if interconnections change. However, NoC comes with a cost. First, because we use soft NoC, NoC costs FPGA resources that could otherwise be allocated to operators. Second, limited NoC bandwidth could be the performance bottleneck of the application. For designs that do not utilize a lot of resources or do not suffer from limited NoC bandwidth, our NoC-based system is enough for the final, optimized design. However, for designs that need to utilize more resources and suffer from limited NoC bandwidth, we should remove the NoC. Since we do not want the final design point of the incremental refinement to be sub-optimal, we need to remove the NoC at some points of the iterative process even if it means a longer, monolithic compilation.

To make FPGA development more productive, our ultimate goal is to support “software-like” FPGA development. In [79, 99, 78], we attack the long FPGA compilation problem so that users can have short edit-compile-debug cycles as they compile software programs. Another difference in FPGA development from software programming apart from long FPGA compilation is the lack of visibility on the inner state of the hardware design. Software engineers are used to rich profiling tools to identify where the application spends the most time. Then, they incrementally refine the bottleneck function to improve the application performance. However, in hardware design, such runtime performance profiling tools are not readily available. Models for application performance or resource utilization can provide estimated results in the early stages of the hardware compilation, but they are potentially inaccurate. As we have a fast and flexible FPGA compilation framework from the previous chapter, we raise a question, *can we identify the bottleneck of the application along with the fast compilation to iterate through the initial design points quickly on the path to an optimized monolithic FPGA design?*

5.2. Previous Work

5.2.1. HLS Design-Space Exploration on FPGA

HLS DSE is generally categorized into two approaches: model-based and synthesis-based, with some methods being a mixture of the two [83]. Model-based techniques build predictive models for performance and resource consumption, so they can quickly search the design-space [68, 69]. However, model-based techniques are inaccurate compared to synthesis-based methods that invoke HLS tools to evaluate the design point, and inaccurate models could lead to a sub-optimal design, under-utilizing or over-utilizing resources for the given platform [83]. Synthesis-based methods use results in the early stages of the hardware mapping like post-HLS estimates [87]. They are more accurate than model-based methods at the expense of runtime, but there is still a huge gap between post-HLS estimates and the actual placed and routed designs. Authors in [27] report that the post-HLS resource estimation errors by the vendor tool can reach up to 141% for LUT utilization and 95% for FF Utilization. The problem with post-HLS estimates is exacerbated for the data-dependent applications. For example, when there are variable loop bounds, AMD Vitis HLS does

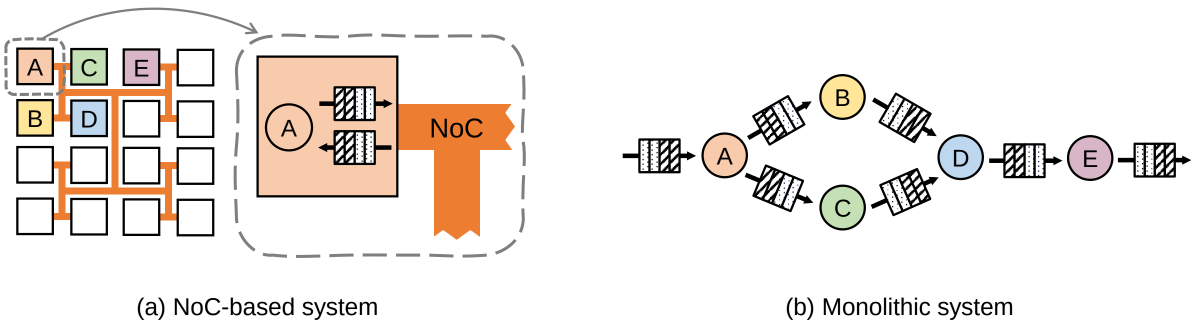


Figure 5.1: NoC-based System and Monolithic System

not report the trip counts as the values are unknown at compile time [24, 12].

We aim to place, route, and run the design on the FPGA. Based on the runtime execution feedback, we incrementally refine the design, so our approach is possibly an extreme version of the synthesis-based DSE. Our strategy is different from the aforementioned approaches that use models or predictive results of the early stages. As FPGA compilation is expensive, instead of exploring many design points to identify the performance bottleneck, we perform bottleneck-guided optimization, similar to the strategy used in [87]. Instead of relying on post-HLS estimates as done in [87], we use the feedback from the placed and routed design that runs on the hardware.

5.3. Incremental Refinement Strategy

We have two approaches to compiling a user design as shown in Figure 5.1. The NoC-based system is our fast compilation framework that supports parallel and incremental compilation using Hierarchical PR (Chapter 3, Chapter 4). However, the NoC and the NoC interfaces add area overhead. A soft NoC costs 11K LUTs in Chapter 4, and a single NoC interface per operator consumes a few hundred to slightly over a thousand LUTs depending on the number of input and output ports [99]. The NoC-based system also results in fragmentation, not fully utilizing available programmable logic. For instance, if 16 operators whose sizes are 5K LUTs are mapped to 16 single-sized PR pages whose sizes are 8K LUTS, 48K LUTs = 3K LUTs×16 in the PR pages are unused. The NoC introduces a limited bandwidth too. If 128 bits of data are transferred over the NoC which has a payload size of 32 bits, it takes four cycles to transmit a single data. The monolithic system is a naive implementation of a dataflow application that directly connects operators with FIFOs. This approach does not have area overhead from the NoC interfaces and fragmentation incurred by the divide-and-conquer strategy. The monolithic system does not suffer from the limited bandwidth of the NoC either. Nevertheless, the compilation is long.

Our incremental refinement strategy quickly maps the user design on hardware with the NoC-based system. We build upon our system from the previous chapter which provides fast and flexible compilation. To identify the bottleneck operator or determine whether the limited NoC bandwidth is the bottleneck, our framework increments FIFO counters in the NoC interface. The support for bottleneck identification in this chapter ([77]) is the key component in incremental refinement that previous split FPGA compilation works are missing.

When bottlenecks are identified, we can refine the bottleneck operators or the NoC communication mapping for the streams between operators. To resolve the NoC bandwidth bottleneck, we can use multiple NoC interfaces for an operator or operator merging to directly connect high bandwidth operators avoiding the need for the NoC on those links, as explained in the previous chapter.

The user or automation selects a new design point by refining either the bottleneck operator or the NoC bandwidth bottleneck. In this process, the size of operators or the operating frequencies of operators should change as previously illustrated in Figure 1.2. When a new design point is selected, our NoC-based system recompiles only the necessary operators. This gives us a new design which can then be profiled with the FIFO counters to identify the next bottleneck.

This iteration continues until the design-space is all explored for the bottleneck operator or the design needs more resources than available in the PR pages. Finally, a design is compiled with the monolithic system that also integrates FIFO counters to identify the bottleneck, and the iteration continues. This strategy can help the users to quickly iterate the important, initial design points, or the flow can be automated to output an optimized design in the given design-space for the final, optimized design. Because our strategy migrates to the monolithic system at the expense of a longer compilation, our final design does not suffer from area overhead or limited bandwidth from NoC infrastructure. We expect to achieve a faster tuning time than the design tuning done only by monolithic compilation, iterating initial yet important design points faster with our fast NoC-based system.

5.4. Bottleneck Identification

As shown in Figure 5.1, operators in a dataflow application are connected with FIFOs. In the NoC-based system, FIFOs reside in the NoC interfaces (Figure 5.1 (a)), and in the monolithic system, FIFOs directly connect operators (Figure 5.1 (b)). Building upon [85, 24], we retrieve information from these FIFOs to identify the bottleneck operator or whether the NoC bandwidth is the bottleneck. The high-level intuition of our bottleneck identification is illustrated in Figure 5.2. In this example, Operator 3 may be slower than Operator 2 because Operator 3 does not consume the data at the rate that Operator 2 produces. Similarly, Operator 3 may be slower than Operator 4 because Operator 3 does not produce the data at the rate that Operator 4 consumes. As a result, Operator 1, Operator 2, and Operator 4 stall while Operator 3 does not stall, busy processing data. In this example, it is likely that Operator 3 is the bottleneck of this application. The idea is similar to `-pg` option in the software compiler that causes each function to call `mcount` routine whose results are later profiled by a program like `gprof` [37].

[85] uses FIFO status (full, empty) to identify the bottleneck in a cluster of "Computing Elements". In evaluation, [85] uses MicroBlaze cores as Computing Elements, and for two different application architectures, they manually improve the application performance based on the monitoring information extracted from the FIFOs. However, [85] does not demonstrate iterative application refinement, and case studies are limited to an array of soft cores. HLScope+ [24] modifies HLS source codes to monitor the module status and FIFO status. HLScope+ introduces a stall analysis network (SAN) that analyzes the root cause of the stall. HLScope+ focuses on generating fast and accurate cycle estimation instead of demonstrating bottleneck identification in design-space exploration. We do not modify HLS source codes; we identify bottlenecks in software based on the raw counter data collected. Unlike previous works, we demonstrate the incremental refinement on realistic applications using bottleneck identification (Section 5.7), and our work is embedded with the fast separate

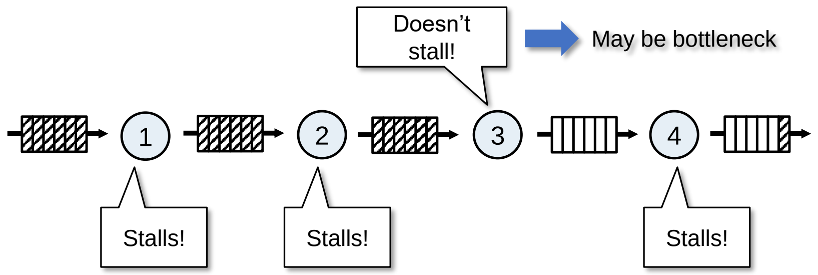


Figure 5.2: High-level Intuition of Bottleneck Identification using FIFO Counters

compile so that the runtime hardware execution feedback is obtained quickly.

5.4.1. Stall Counters

Both the initial NoC-based system and the final monolithic system utilize *stall counters* to identify the bottleneck operator as shown in Figure 5.3 (a). The input stall condition is defined as the state when *the input FIFO is empty and the user operator asserts a ready signal.* At a high level, it means that the operator wants to process the data, but the data is not available, so the operator stalls. Similarly, the output stall condition is defined as the state when *the output FIFO is full and the user operator asserts a valid signal for the data.* At a high level, it means that the operator wants to output the data, but the successor is still busy processing the previous data. The stall condition for the operator is asserted when *at least one input FIFO has a stall condition or at least one output FIFO has a stall condition.* Finally, the stall counter increments when the stall condition is asserted. We know that as the number of stall counters for an operator is low, the operator is likely to be the bottleneck because this operator is busy processing some data while other operators are waiting for the input data or waiting to output the data.

5.4.2. Full Counters

In the NoC-based system, each operator has a single, limited-bandwidth input channel into the NoC and a single limited-bandwidth output channel; these can be narrower than the total input and output width needed by the application, and multiple input and output ports share one input channel and one output channel. The limited NoC bandwidth could limit the application performance [96]. A NoC bottleneck can be detected similarly with full counters on the FIFOs associated with the

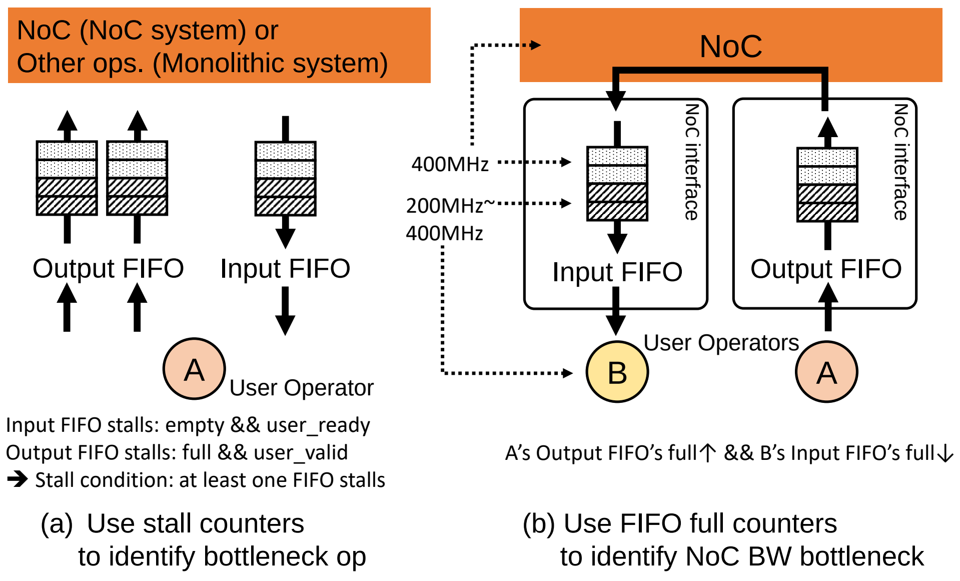

NoC (NoC system) or
Other ops. (Monolithic system)
Output FIFO
Input FIFO
A
User Operator
Input FIFO stalls: empty && user_ready
Output FIFO stalls: full && user_valid
➔ Stall condition: at least one FIFO stalls
(a) Use stall counters to identify bottleneck op
NoC
400MHz
200MHz~400MHz
NoC interface
Input FIFO
NoC interface
Output FIFO
User Operators
B
A
A's Output FIFO's full↑ && B's Input FIFO's full↓
(b) Use FIFO full counters to identify NoC BW bottleneck

Figure 5.3: Bottleneck Identification with FIFOs

stream links into and out of each operator. Full counters increment when the FIFO is full. In Figure 5.3 (b), Operator A sends data to Operator B through the NoC. If Operator A's output FIFO has large full counters and Operator B's input FIFO has small full counters, we can assume that the NoC bandwidth could be a bottleneck. This means that Operator A tries to send out the data often, but Operator B does not receive the data at a similar rate. In our system, we consider there exists NoC bandwidth bottleneck *if the difference in the full counters (output FIFO's full counters of A - input FIFO's full counters of B) is large enough.*

5.4.3. Resource Usage

Logic related to FIFO counters is implemented in RTL along with the NoC interface (NoC-based system) or the top-level wrapper function (monolithic system). Other than stall counters and full counters, we keep empty counters and read counters for debugging purposes. In the NoC-based system, a NoC interface and a user operator are mapped in a PR page. When the counters are set to 28-bit registers, a NoC interface with a single user input stream (32-bit) and a single user output stream (32-bit) costs about 700 LUTs, 1000 FFs, and 4 36Kb BRAMs including counter logic. Logic related to counters alone uses about 200 LUTs and 400 FFs. The size of single-sized pages in the previous chapter (Table 4.1) is about 7,000–8,000 LUTs, so the counter logic is about

2–3% of the LUTs in a single-sized PR page. In the monolithic system, logic related to counters uses about 60 LUTs and 140 FFs per operator. One reason for the discrepancy in resource usage is that the NoC-based system uses about double the number of FIFOs than the number of FIFOs used in the monolithic system. For instance, for operators A and B in Figure 5.3 (b), in the NoC-based system, they need one output FIFO for A and one input FIFO for B. In the monolithic system, on the other hand, one FIFO between A and B is enough. Among the benchmarks in our experiments (Section 5.7), Rendering has the largest number of streams (30 streams) thereby consuming the most resources for counter logic. In the final monolithic design of Rendering, counter logic is only 3.4% (1900 LUTs) of the final design's total LUT utilization, and only 5.3% (3500 FFs) of the FF utilization.

5.4.4. Limitations

It is possible to have a stall counter per input and output stream to identify the problematic stream for finer-grained analysis. For simplicity, we keep a single stall counter per operator and identify the bottleneck operator.

For the same run time, an operator running at 400MHz can have up to twice as large stall/full counters as an operator running at 200MHz. Therefore, we normalize counters by dividing them by the operating frequencies, but this simple approach may not be sufficient to reflect the differences in operating frequencies. Furthermore, an operator with different rates of input and output may be harder to classify correctly as the low rate side is less likely to fill a FIFO than the high rate side. Although our bottleneck identification based on FIFO counters is an approximation, in Section 5.7, we show how our approach based on FIFO counters identifies bottleneck operators and resolves the NoC bandwidth bottleneck in incremental development for realistic HLS designs, improving the application performance by 2.2–12.7×. A more detailed analysis using a simple experimental setup and thought experiments is presented in Section 5.8.4 and Section 5.8.5.

5.5. NoC-based System and Monolithic System

In this chapter, we use the enhanced fast separate compilation framework from Chapter 4 as the NoC-based system. Therefore, in addition to Hierarchical PR pages from Chapter 3, the NoC-based system supports multiple NoC interfaces to a single operator, multiple clock frequencies (200MHz–400MHz), clean PR regions to remove static routing over the PR regions, and page assignment based on recursive bi-partitioning. The assumption in our fast incremental refinement strategy is that the NoC-based system can quickly explore the design points that the monolithic system would have explored, so using multiple NoC interfaces or merging operators that suffer from limited bandwidth would help to make a smooth continuum. FIFO counter logic explained in Section 5.4 is embedded in the NoC interface to support profiling capability.

The monolithic system directly connects operators with FIFOs as previously shown in Figure 5.1 (b). The source codes for the NoC-based system and monolithic system are identical, and the monolithic system has a top wrapper module that instantiates all the operators, FIFOs, and FIFO counter logic. Similar to the NoC-based system, operators can run at different clock frequencies (200MHz–400MHz). HLS for each operator is done in parallel. After all HLS runs, our framework automatically generates a top wrapper file (Verilog) and performs logic synthesis monolithically. Then, the generated netlist is linked with (Vivado's `link_design` command) the monolithic overlay which contains similar AXI Interconnect (runs 300MHz) and peripheral infrastructure to the NoC-based system. The rest of the placement, routing, and bitstream generation are also done monolithically.

5.6. Automated DSE Case Study

Stall counters and full counters in Section 5.4 are useful when users decide on the next design point in incremental refinement scenario. We showcase the automated DSE case study that entirely removes user's intervention in the loop and automatically chooses the next design point based on the runtime execution on the hardware.

5.6.1. DSE Experiment Overview

Figure 5.4 shows the overview of our automated DSE experiment. The inputs of the automation system are the HLS source code generator, parameter design-space (`params.json`), and parameter annotation `param_annotate.json`. An example of HLS source code generator is an open-source framework like FINN from AMD Research [92, 16]. An example of `params.json` for one of the

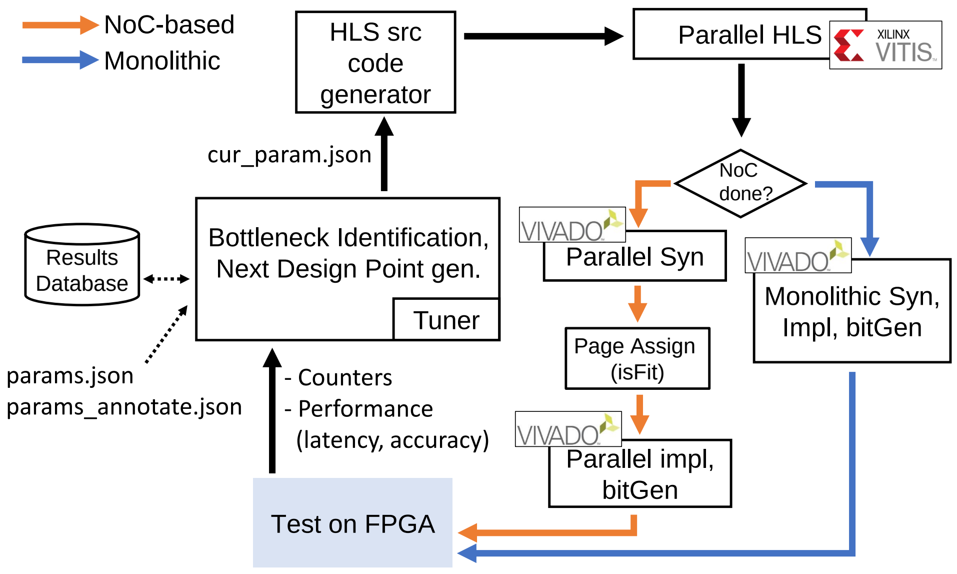


Figure 5.4: Automated DSE Experiment Overview

CNN application from FINN is shown in Listing 5.1. Because both the NoC-based system and the monolithic system support different clock frequencies for kernel operators, kernel clock is one parameter. Parameters like `SIMD1_conv_2` and `PE1_mva_2` control the degree of parallelism in the operators generated by FINN. Possible values for these parameters are listed in Listing 5.1. The goal of DSE could be different metrics. One example of a metric is application execution latency for the input data. In addition to the application latency, Optical Flow and Digit Recognition in Section 5.7.4 have a varying accuracy with different parameter values. Users should specify which parameter is related to which metric in `param_annotate.json`. For example, Digit Recognition has `{‘‘PAR_FACTOR’’: [‘‘latency’’], ‘‘K_CONST’’: [‘‘accuracy’’]}` in `param_annotate.json` to indicate that different values of `‘‘PAR_FACTOR’’` could potentially improve latency and different values of `‘‘K_CONST’’` could improve accuracy. Other potential metrics include resource utilization, power consumption, or a combination of multiple metrics like throughput per unit power. The reason why each parameter has a list of elements in `param_annotate.json` is that some parameters may improve multiple metrics. Throughout the incremental refinement, we have `cur_param.json` that record the parameters for the current design point. An example of `cur_param.json` is shown in Listing 5.2.

```
{
  "kernel_clk": [200, 250, 300, 350, 400],
  "PE1_mva_0": [32],
  "SIMD1_conv_1": [32],
  "PE1_mva_1": [32],
  "SIMD1_conv_2": [8, 16, 32],
  "PE1_mva_2": [8, 16, 32, 64],
  "SIMD1_conv_3": [8, 16, 32, 64],
  "PE1_mva_3": [8, 16, 32, 64],
  "SIMD1_conv_4": [1, 2, 4, 8, 16, 32, 64],
  "PE1_mva_4": [1, 2, 4, 8, 16, 32, 64, 128],
  "SIMD1_conv_5": [1, 2, 4, 8, 16, 32, 64, 128],
  "PE1_mva_5": [1, 2, 4, 8, 16, 32, 64, 128],
  "PE1_mva_6": [1, 2, 4, 8, 16, 32, 64, 128, 256],
  "PE1_mva_7": [1, 2, 4, 8, 16, 32, 64, 128, 256]
}
```

Listing 5.1: Parameter Design-Space Example (CNN-2's `params.json`)

```
{
    "layer_0_0": {
        "kernel_clk": 400,
        "num_leaf_interface": 1
    },
    "layer_0_1": {
        "PE1_mva_0": 32,
        "kernel_clk": 350,
        "num_leaf_interface": 1
    },
    "layer_1_0": {
        "SIMD1_conv_1": 32,
        "kernel_clk": 300,
        "num_leaf_interface": 1
    },
...
```

Listing 5.2: Current Parameter Example (CNN-2's `cur_param.json`)

HLS for each operator is run in parallel. Following our fast incremental refinement strategy (Section 5.3), the automated DSE initially uses the NoC-based system to quickly map the design on hardware. Then, stall counters and full counters are extracted after the application execution finishes. In both the NoC-based system and the monolithic system, the number of output data is counted, and when the number reaches the value, `OUTPUT_SIZE` specified in the host code of the application, the system sends signals to each operator to stop incrementing FIFO counters. Then, FIFO counters are sent back to the host for analysis. Based on these counters, our script identifies the bottleneck and selects the next design point. Listing 5.3 shows an example of collected stall counters from the hardware execution. As explained in Section 5.4.4, stall counters are divided by the kernel clock frequency to account for the fact that a higher clock frequency leads to a higher number of counters. Along with FIFO counters, the correctness of the application or the accuracy of the current design (Optical Flow, Digit Recognition) is evaluated in software based on the output data after hardware execution. When the design reaches the point that the automated DSE explores all the design-space or needs more area, it migrates to the monolithic system.

```
Normalized stalls:
layer_0_0 1287.055
layer_0_1 7827.225
layer_1_0 10829.755
layer_1_1 11637.765
layer_2_0 16523.355
layer_2_1 15396.41
layer_3_0 9518.235
layer_3_1 193.84
layer_4_0 22426.755
layer_4_1 4692.695
layer_5_0 29806.3
layer_5_1 29806.3
layer_last_0 28587.73
layer_last_1 16302.315
layer_last_2 32580.845

>> Bottleneck operator: layer_3_1
>> Param to tune: PE1_mva_3
>> Next parameter: 16
```

Listing 5.3: Runtime Execution Feedback Example (CNN-2)

5.6.2. Greedy Tuner

Our tuner used in the automated DSE case study is as simple as selecting the operator with the lowest number of stalls and changing the operator's design point that can improve the application latency. In Listing 5.3's example, `layer_3_1` has the lowest normalized stall counters, so its parameter, `PE1_mva_3` value (Listing 5.1) changes to 16. The algorithm is detailed in Algorithm 1. If the current design point, the one that is the most recently run on hardware, leads to implementation failure, then we revert to the previous design point. Also, if the current design point leads to worse latency than the best latency recorded, then we revert to the previous design point. In some cases, the latency did not degrade much but could be slightly worse than the best latency because of noise. This could be the case where multiple operators need to be refined to result in better latency. Therefore, we have a margin (`MARGIN` in Algorithm 1, 10% in the experiment) to account for the noise. When we select the next design point (`UPDATE_DESIGN_POINT` in Algorithm 1), we first check whether the NoC bandwidth bottleneck exists because if we do not resolve the NoC bandwidth bottleneck immediately, we may identify the wrong bottleneck operator. If the NoC bottleneck is not detected, then we select the next design point for the bottleneck operator. In Listing 5.3's case, the NoC bottleneck is not detected, so the operator with the least stalls is identified as the bottleneck. We have a list of candidates for the bottleneck operator whose stall counters are similar to the lowest stalls (10% in the experiment) and select the next point for the bottleneck. In the NoC-based system, when we cannot improve the bottleneck operator anymore, then we move to a monolithic system. In the monolithic system, when we cannot improve the bottleneck operator anymore, we end the DSE.

As mentioned in Section 5.6.1, while we focus on improving application latency, in some cases, we want to reach a certain bar of a different performance metric and then optimize for latency. For example, Digit Recognition uses a K-Nearest-Neighbor (KNN) algorithm, and a larger K leads to better accuracy but worse latency. In such cases, the users can indicate the minimum accuracy in `params.json`, and our tuner will prioritize the accuracy metric first.

Some benchmarks could have almost *identical* operators. For example, maybe designs have data-parallel sections where one can allocate many identical data-parallel operators. If there are 10 identical operators, even if we explore only 5 kernel frequencies available in our system (200MHz–400MHz) not the design parameters, it would take 50 iterations. By default, our script updates the parameter values of identical operators together since independently refining one operator at a time would unnecessarily take a long time. Nevertheless, if the computation is data-dependent, separately tuning each operator could be useful because one operator that has a heavy computation load may need to run at 350MHz while another identical operator with a light load can run at

Algorithm 1 Identify Bottleneck and Generate Next Design Point

```
 1: procedure UPDATE_DESIGN_POINT(dp, counters)
 2:     new_dp, is_NoC_bottleneck ← update_design_point_NoC(dp, counters)
 3:     if is_NoC_bottleneck then
 4:         return new_dp
 5:     else
 6:         bottleneck_list ← least_stalls(counters, MARGIN)
 7:         for operator in sorted bottleneck_list do
 8:             for new_dp in operator's dp_space do
 9:                 if not visited(new_dp) then
10:                     return new_dp
11:         return None
12: end procedure
13:
14: procedure GEN_NEXT_DESIGN_POINT(dp, counters, result)
15:     Save the dp, counters, results                          ▷ dp: Design Point
16:     if current impl failed then
17:         dp, counters ← revert_to_prev_dp()
18:     else
19:         if latency×(1-MARGIN) > best_latency then
20:             dp, counters ← revert_to_prev_dp()
21:         if latency < best_latency then
22:             best_latency ← latency
23:     new_dp ← update_design_point(dp, counters)
24:
25:     if NoC based flow and new_dp is None then
26:         Move to Monolithic flow
27:     else if Monolithic flow and new_dp is None then
28:         DSE is done
29: end procedure
```

200MHz. In Section 5.7's Rendering benchmark, we will show the DSE results of both approaches.

What is not shown in Algorithm 1 is how the automated DSE script handles the implementation failure. In the NoC-based system, the reasons of implementation failure could be: (1) there is no valid page assignment available, (2) the tool cannot place all the elements or cannot route completely in an assigned page, and (3) the tool can route completely but violates timing. In the case of (1), our DSE script directly moves to a monolithic system, not going through Algorithm 1 because (1) occurs due to limited resources available in the NoC-based system. In the case of (2), our script tries larger pages (double or quad) for the operators that fail in implementation. If (2) occurs with the largest size of the PR page, then our script moves to the monolithic system. In the case of (3), our script uses Algorithm 1 to revert to the previous design point, and it finds the next design point and tries it.

5.7. Evaluation

In this section, we evaluate the incremental refinement strategy with automated DSE case studies. The "incremental flow" refers to DSE that uses our fast incremental strategy introduced in Section 5.3, and the "monolithic flow" refers to DSE that uses a monolithic system throughout. We evaluate how both incremental flow and monolithic flow improve the application performance with our bottleneck identification. We also compare both flows in DSE time.

5.7.1. Experiment Setup

The target device is AMD ZCU102 evaluation board featuring UltraScale+ ZCU9EG FPGA. We use ZCU102 DFX platform [103] for the NoC-based system as we partially reconfigure each PR page separately in parallel. In the ZCU102 DFX platform we use, the dynamic region, the area that can be partially reconfigured, contains 262,496 LUTs, 1,752 18Kb BRAMs and 2,448 DSPs. We use the ZCU102 platform (non-DFX platform) for the monolithic system. 274,080 LUTs, 1,824 18Kb BRAMs and 2,520 DSPs are available in ZCU102 platform. Since we use the non-DFX platform for the monolithic system, every time we generate a new monolithic bitstream, our DSE script copies the newly generated image, reboots the device, and runs the application. We do not include packaging time to create a boot image file or booting time in the monolithic flow's compile time because this overhead stems from the fact that the monolithic flow uses the non-DFX platform. We use Vitis 22.1 including Vitis HLS and Vivado. We run the automated DSE experiments on a workstation equipped with the 3.4GHz AMD Ryzen 9 5950X 16 Core CPU with 32 processing threads and 128 GB of RAM.

5.7.2. NoC-based Overlay and Monolithic Overlay

Table 4.1 from the previous chapter shows the available resources in the NoC-based system's PR pages used in evaluation. The NoC-based system, shown in Figure 4.3, consists of 20 single-sized pages (7,264–7,919 LUTs, 44–66 18Kb BRAMs, 44–88 DSPs), 11 double-sized pages (14,647–15,840 LUTs, 110–132 18Kb BRAMs, 131–154 DSPs) and 5 quad-sized pages (29,806–31,392 LUTs, 220–264 18Kb BRAMs, 264–308 DSPs). One double page is not subdivided into two single pages in the overlay used in this experiment because we had difficulties in successfully routing and closing 400MHz of timing when we subdivide all 11 double-sized pages. Total 64% of LUTs, 78% of BRAMs, and 63% of DSPs are available in the PR pages. An Abstract Shell for each PR page is generated accordingly, and synthesized operators are mapped to appropriate Abstract Shells with the page assignment algorithm previously mentioned in Section 4.5. We use a BFT NoC as done in [78]. The number of processing elements (PEs) in the NoC is 24, Rent's parameter p [59] is 0.67, and sizes of the packet and payload are 49 bits and 32 bits respectively.

Two PEs are used for the NoC configuration and DMA. The BFT uses 11,297 LUTs, and other peripherals including AXI interconnect use about 27K LUTs. The reason why the NoC in Chapter 4 (11,297 LUTs) costs less than the NoC in Chapter 3 (11,799 LUTs) despite the higher Rent's parameter is that in Chapter 4, the unused subtree that has 8 PEs is removed. The monolithic overlay uses about 23K LUTs. The reason why the NoC-based system uses slightly more resources than the monolithic system in the static overlay (27K (excluding NoC) $>$ 23K) is that the NoC-based system has two static DMA operators and small FIFOs that generates almost-full signals.

5.7.3. Implementation Directives

AMD Vivado supports different directives in the implementation phase. For example, `place_design` has 18 directives, and `route_design` has 8 directives [7]. These directive options could be included in the design-space as done in [53, 106, 93], but we use `ExtraTimingOpt` and `EarlyBlockPlacement` for `place_design`'s directive and `Explore` for `route_design`'s directive.

In the NoC-based flow, we could have different implementation directives for different operators. For example, if the implementation for a certain PR page slightly violates timing, we can explore different directives for the page. However, the monolithic flow allows a single directive per application as the design is monolithically placed and routed. Finer-grained tuning in the NoC-based flow could lead to better performance in the NoC-based flow, and we may need to reconsider the continuum from the NoC-based system to the monolithic system if we want to use our incremental refinement strategy. A related issue is further discussed in Section 7.6.

5.7.4. DSE Time and Performance

Figure 5.5, Figure 5.6, and Figure 5.7 show the benefit of our incremental refinement strategy in DSE time. The incremental strategy uses the NoC-based fast compile and then migrates to the monolithic system (green), and the monolithic flow monolithically compiles the design for the entire DSE (red). Compilations with the NoC-based system are marked as $\times$ and those with the monolithic system are marked as as ●. Both the NoC-based system and the monolithic system use FIFO counters to identify the bottleneck as discussed in Section 5.4. Figure 5.5, Figure 5.6, and Figure 5.7 record the best kernel latency, so if the implementation fails or does not improve the kernel latency, the best kernel latency so far is marked. Table 5.1 shows the resource utilization of the incremental strategy at different stages: the initial design point (Init: application + NoC interfaces) with the NoC-based flow and the final design point after the monolithic flow is over (Mono Final: application + AXI interconnect + peripherals). The reason why "Mono Final" data for Rendering is not available is that the final design point of the NoC flow did not meet the timing for the monolithic system. In such cases, the final design point of the NoC flow is the best design, superior to the final design point of the monolithic-only flow. Table 5.1 also shows improvement in the application latency and DSE time. Since we consider the design that matches the minimum accuracy as a valid design for Digit Recognition and Optical Flow, the latency improvement is calculated as the latency achieved by the monolithic flow that first matches the minimum accuracy divided by the latency achieved by the final design of the fast incremental strategy. For *isFit* classifiers (Section 4.6) in the NoC-based system's page assignment algorithm, the recall threshold of 0.96 is used in the experiments.

Table 5.1: Resource Utilization and DSE Results

Benchmarks		LUT %	FF %	BRAM %	DSP %	Latency Improvement	Speedup in DSE time
Rendering[†]	Init	4	2	9	0	3.8×	2.5×
	Mono Final	20	12	11	1		
Rendering	Init	3	2	6	0	3.9×	1.8×
	NoC Final	15	7	14	1		
	Mono Final	-	-	-	-		
Digit Recognition	Init	9	5	21	0	12.7×	1.3×
	Mono Final	58	34	97	0		
Optical Flow	Init	7	4	11	5	3.8×	0.9×
	Mono Final	15	12	14	10		
Optical Flow[‡]	Init	7	4	11	5	3.9×	1.4×
	Mono Final	14	10	13	4		
CNN-1	Init	13	6	11	0	2.2×	2.7×
	Mono Final	19	13	13	0		
CNN-2	Init	17	8	11	0	2.2×	2.3×
	Mono Final	25	15	13	0		
CNN-3	Init	17	8	11	0	2.2×	1.7×
	Mono Final	31	16	16	0		

Rendering[†]: Rendering when identical operators are separately tuned
Optical Flow[‡]: Optical Flow with a lower accuracy target

Rendering

The design-space of the user parameters for Rendering from Rosetta benchmarks [109] includes parallelization factor for `rasterization` function and `zculling` function. As stated in Section 5.6, by default, we tune identical operators together. For example, `zculling`'s parallelization factor of 4 results in four `zculling` operators, and when one of the `zculling` operator is identified as a bottleneck, the tuner increases the clock frequency for all four identical operators together. However, data-dependent applications like Rendering may require independent tuning for identical operators even if independent tuning takes longer. Figure 5.5's "Rendering" (without [†]) is the DSE results when identical operators are tuned together, and Figure 5.5's "Rendering[†]" is the DSE results when identical operators are refined separately.

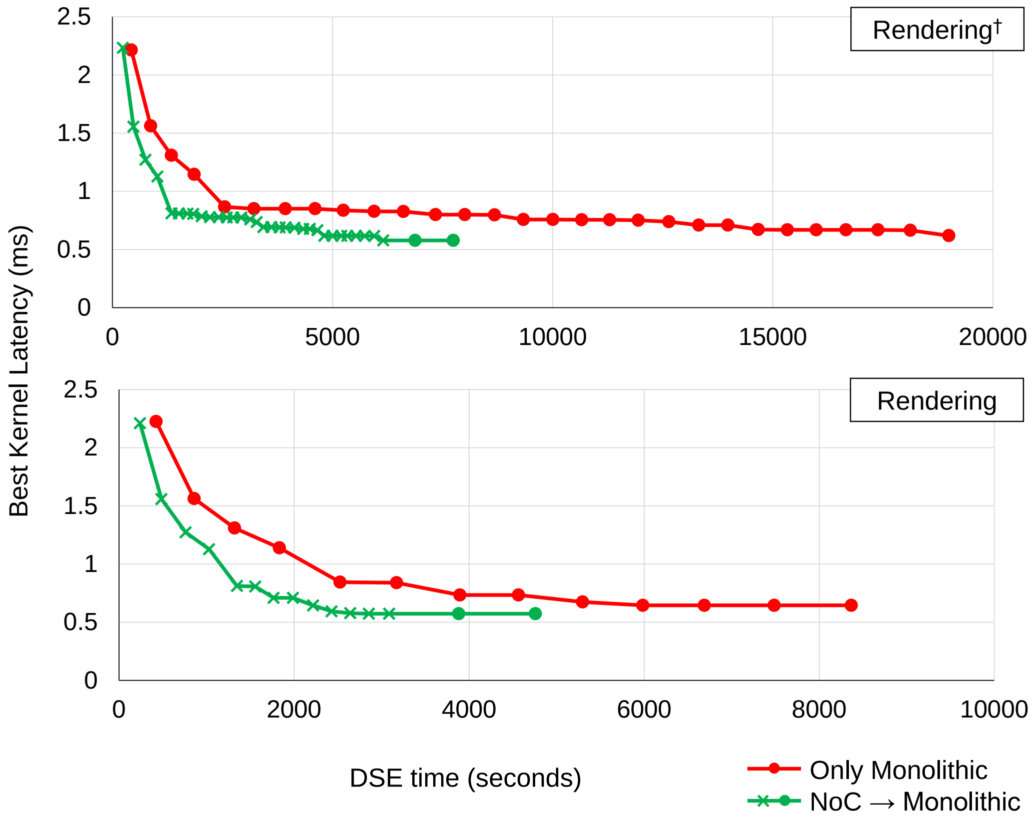


Figure 5.5: DSE Results for Rendering

Rendering†: Rendering when operators are separately tuned.

Table 5.2 illustrates each step of our incremental strategy in Figure 5.5's Rendering†. For the first few iterations, our greedy tuner increases the parallelization factor for `rasterization` function and `zculling` function, generating new operators, and this is why the number of parallel compile runs increases. When the design parameters are all explored, clock frequencies are explored. We can see that the bottleneck initially exists in `rasterization`, and as we improve `rasterization`, the bottleneck moves to `zculling`. In the iteration count of 23, the implementation of `zculling_i2` violates timing, and in this case, it marks 300MHz as visited and tries the next design point, the clock frequency of 350MHz because `zculling_i2` is still the bottleneck operator. After the iteration count of 32, our system indicates that `zculling_i3` is the bottleneck with the lowest number of stalls, but at this point, we have already explored all the design-space for `zculling_i3`. Thus, we migrate to the monolithic *with exactly the same parameter values* (denoted as "Prev config." in Table 5.2). After the iteration count of 33, it indicates `rasterization_i*` and `zculling_i3` as the bottlenecks. Even if 400MHz clock of `zculling_i3` failed in the NoC-based flow, we give another try in the monolithic system in the iteration count of 34. DSE finishes after the iteration count of 34 because while the final design still points to `rasterization` and `zculling` as the bottlenecks, `rasterization` operators already reach the maximum kernel frequency and `zculling` fails at 400MHz. Our strategy achieves 1.8× and 2.5× faster DSE time in Figure 5.5, improving 3.9× and 3.8× in application latency respectively. The iteration count of 33 and 34 are repetitive; we explore the same design point in the monolithic system which was already explored in the NoC-based system. However, to make sure that the discrepancy between our incremental strategy and the monolithic flow is minimal, we rerun the same configuration and retry the failed design points on the monolithic system. Our approach still achieves faster tuning by iterating many initial design points within 2–3 minutes using the fast NoC-based system. The DSE trace for Rendering is appended in Table A.1.

Table 5.2: Fast Incremental Refinement DSE Trace for Rendering†

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	237s	None	init	-	2.23ms	2.23ms	5	NoC
2	241s	`rast2_i1`	`PAR_RAST` = 2	lat.	1.55ms	1.55ms	4	NoC
3	273s	`zculling_i1`	`PAR_ZCUL` = 2	lat.	1.27ms	1.27ms	8	NoC
4	270s	`rast2_i1`	`PAR_RAST` = 4	lat.	1.13ms	1.13ms	9	NoC
5	320s	`zculling_i2`	`PAR_ZCUL` = 4	lat.	0.81ms	0.81ms	13	NoC
6	170s	`rast2_i1`	clk = 250MHz	lat.	0.81ms	0.81ms	1	NoC
7	195s	`rast2_i2`	clk = 250MHz	lat.	0.81ms	0.81ms	1	NoC
8	131s	`rast2_i4`	clk = 250MHz	lat.	0.81ms	0.81ms	1	NoC
9	191s	`zculling_i3`	clk = 250MHz	lat.	0.78ms	0.78ms	1	NoC
10	189s	`rast2_i3`	clk = 250MHz	lat.	0.78ms	0.78ms	1	NoC
11	197s	`rast2_i1`	clk = 300MHz	lat.	0.78ms	0.78ms	1	NoC
12	186s	`rast2_i2`	clk = 300MHz	lat.	0.78ms	0.78ms	1	NoC
13	144s	`rast2_i4`	clk = 300MHz	lat.	0.78ms	0.78ms	1	NoC
14	184s	`rast2_i3`	clk = 300MHz	lat.	0.77ms	0.77ms	1	NoC
15	204s	`zculling_i3`	clk = 300MHz	lat.	0.76ms	0.76ms	1	NoC
16	144s	`zculling_i2`	clk = 250MHz	lat.	0.74ms	0.74ms	1	NoC
17	154s	`zculling_i4`	clk = 250MHz	lat.	0.69ms	0.69ms	1	NoC
18	182s	`rast2_i1`	clk = 350MHz	lat.	0.69ms	0.69ms	1	NoC
19	186s	`rast2_i2`	clk = 350MHz	lat.	0.69ms	0.69ms	1	NoC
20	140s	`rast2_i4`	clk = 350MHz	lat.	0.69ms	0.69ms	1	NoC
21	197s	`rast2_i3`	clk = 350MHz	lat.	0.69ms	0.69ms	1	NoC
22	199s	`zculling_i3`	clk = 350MHz	lat.	0.68ms	0.68ms	1	NoC
23	150s	`zculling_i2`	clk = 300MHz	lat.	-	0.68ms	1	NoC
24	173s	`zculling_i2`	clk = 350MHz	lat.	0.67ms	0.67ms	1	NoC
25	157s	`zculling_i4`	clk = 300MHz	lat.	0.62ms	0.62ms	1	NoC
26	189s	`rast2_i1`	clk = 400MHz	lat.	0.62ms	0.62ms	1	NoC
27	193s	`rast2_i2`	clk = 400MHz	lat.	0.62ms	0.62ms	1	NoC
28	139s	`rast2_i4`	clk = 400MHz	lat.	0.62ms	0.62ms	1	NoC
29	198s	`rast2_i3`	clk = 400MHz	lat.	0.63ms	0.62ms	1	NoC
30	198s	`zculling_i1`	clk = 250MHz	lat.	0.62ms	0.62ms	1	NoC
31	215s	`zculling_i3`	clk = 400MHz	lat.	-	0.62ms	1	NoC
32	204s	`zculling_i4`	clk = 350MHz	lat.	0.58ms	0.58ms	2	NoC
33	728s	Prev config.	Prev config.	lat.	0.60ms	0.58ms	1	Mono
34	865s	`zculling_i3`	clk = 400MHz	lat.	-	0.58ms	1	Mono

Rendering†: Rendering when operators are separately tuned.

Digit Recognition

The design-space of the user parameters for Digit Recognition from Rosetta benchmarks includes parallelization factor KNN algorithm and K value. In the experiment, we set the minimum accuracy of 0.94, so our greedy tuner navigates the parameter (K value) to meet this accuracy first and then tunes for latency. This is why we see an increase in latency for the first four iterations. Our strategy achieves 1.3× faster DSE time compared to the monolithic flow while improving 12.7× in application latency.

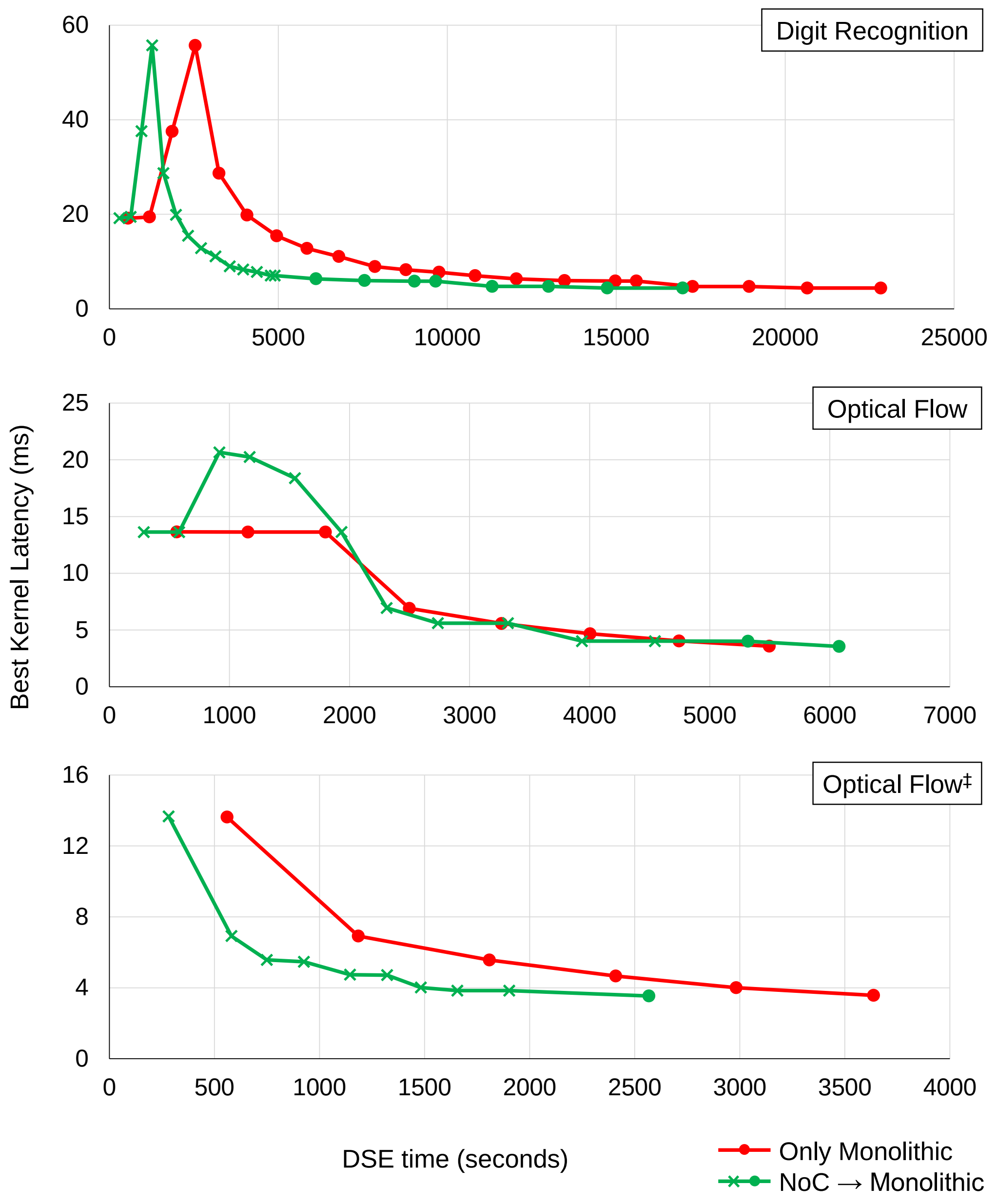


Figure 5.6: DSE Results for Digit Recognition and Optical Flow

Optical Flow‡: Optical Flow with a lower accuracy target.

In Digit Recognition, our strategy shifts to the monolithic system relatively quickly compared to other benchmarks, and this is because our tuner explores the user parameters (parallelization factor) first and then explores kernel frequencies. In Digit Recognition, we have 10 identical operators for the entire tuning and tune them together. The NoC-based system reaches to the parallelization factor that requires more BRAMs than available in the PR pages, and the design is migrated to the monolithic system. Then, different clock frequencies are all explored in the monolithic system, leading to a long tail in Figure 5.6. The DSE trace for Digit Recognition is appended in Table A.2.

Optical Flow

The design-space of the user parameters for Optical Flow from Rosetta benchmarks includes parallelization factor and width of `OUTER_WIDTH` variable that affects the accuracy of the application. Similar to Digit Recognition, it takes three iterations to reach the user-defined minimum accuracy and then tunes for the latency. Optical Flow is the application that our incremental strategy takes longer than the monolithic flow to reach the final design. In the incremental strategy, for the total 11 iterations spent with the NoC-based flow, 3 of them were to mitigate the limited NoC bandwidth by merging operators (Section 4.1.2). These three iterations are "extra" that are not necessary for the monolithic flow. Moreover, as the operators are merged to resolve the NoC bottleneck, the size of the merged operator becomes large, and the benefit of the fast separate compilation approach is reduced. Monolithic flow identifies one obvious bottleneck operator (`tensor_weight_y`) which is still the bottleneck when it is tuned to run with the maximum 400MHz. Optical Flow‡ is the version when the accuracy target is relaxed so that `OUTER_WIDTH` variable does not increase and does not cause NoC bottleneck in the DSE. In this version, our strategy achieves 1.4× faster DSE time than the monolithic flow. The DSE traces for Optical Flow benchmarks are appended in Table A.3 and Table A.4.

CNN

Other than Rosetta Benchmarks, we use FINN open-source framework [92, 16] to generate Convolutional Neural Network (CNN) benchmarks. As the input of the separate FPGA compilation framework is a dataflow graph, FINN, which generates the streaming architecture naturally fits with the separate compilation.

FINN is an end-to-end framework that generates HLS codes for the input architecture and a bitstream in the end. In our demonstration, we use FINN to generate HLS codes and run the compilation using our separate compilation framework. In the generated HLS codes, performance and resource utilization for each layer can be controlled by parameters like `PE` and `SIMD`, and if `target_fps` is given, FINN generates HLS codes with appropriate `PE` and `SIMD` values with FINN's own resource and performance models. We receive a hint from this configuration to set the starting point of the applications instead of starting from the minimum `PE` and `SIMD`. For example, `PE` and `SIMD` of the first two convolution modules and one matrix multiplication module are maxed out from the beginning as previously shown in Listing 5.1.

Based on [92], we create small CNNs with 6 convolutional layers and train the networks for CIFAR-10 dataset. CNNs consist of three successions of two 3×3 convolutional layers followed by one 2×2 maxpool layer. There are two fully connected layers in the end. The size of convolutional channels are 32, 32, 64, 64, 128, and 128. CNN-1 has 1 bit for both weight quantization and activation quantization, CNN-2 has 1 bit for weight and 2 bits for activation, and CNN-3 has 2 bit for both weight quantization and activation quantization. The design-space of the user parameters for CNN benchmark includes `SIMD` values for convolution modules and `PE` values for matrix multiplication modules. Figure 5.7 shows that our DSE system achieves 1.7–2.7× faster DSE time compared to the monolithic flow while improving 2.2× in application latency. The starting configuration is set by FINN assuming a single clock for all the operators. During DSE, as we increase the frequency of the bottleneck layer, the bottleneck moves to the different layer, exploring new `SIMD` values or `PE` values. The final designs in both CNN-1 and CNN-2 identify the first convolutional layer as the bottleneck operator which already reaches the maximum `SIMD` value and maximum clock frequency. Monolithic flow in CNN-3 fails to run the first convolutional layer at 400MHz and cannot visit other design points after that. The incremental flow in CNN-3 successfully runs the first convolutional layer at 400MHz, visits other design points, and identifies the first convolutional layer as the bottleneck, which already reaches the maximum clock frequency. Therefore, in CNN-3, the monolithic flow

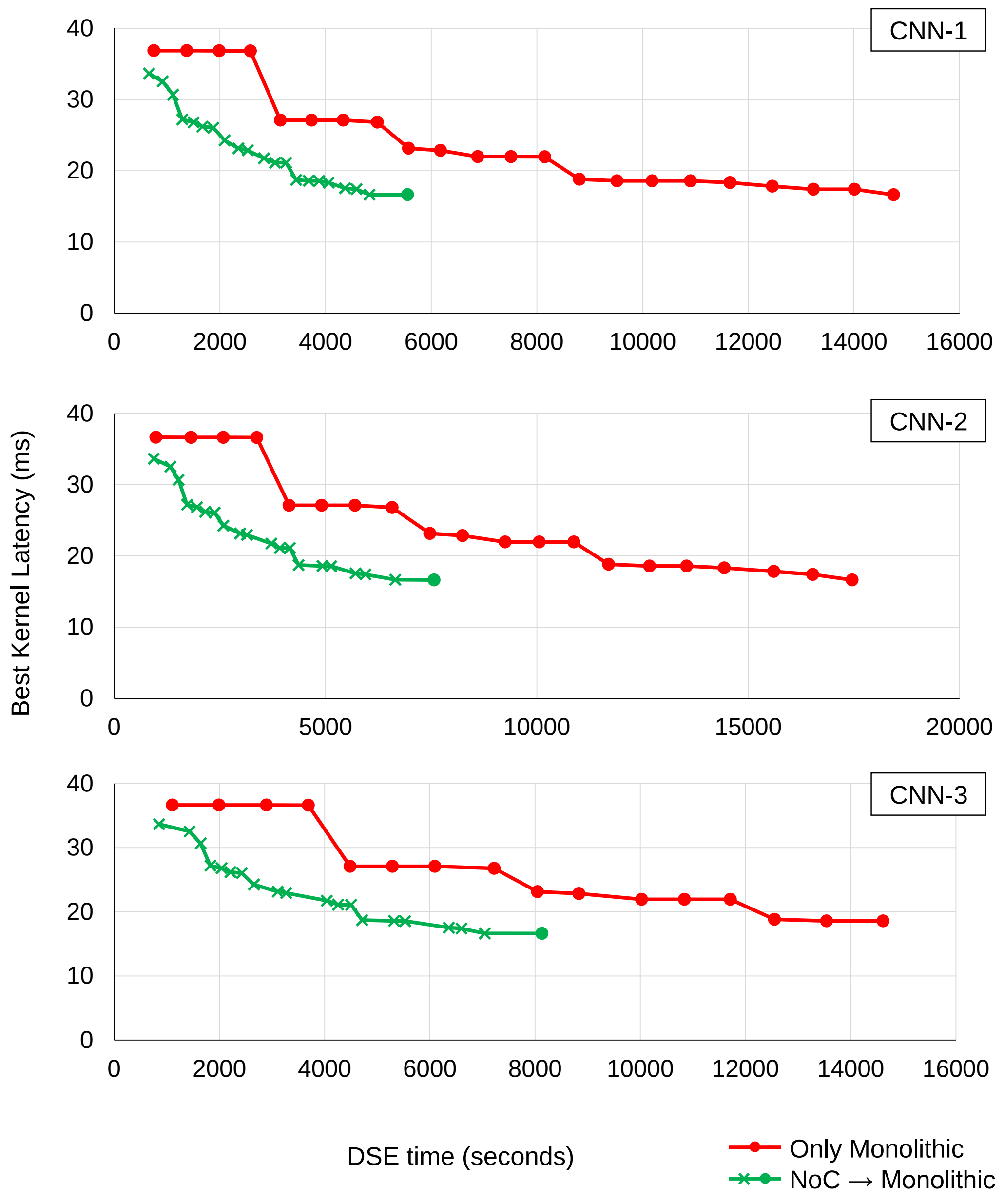


Figure 5.7: DSE Results for CNN

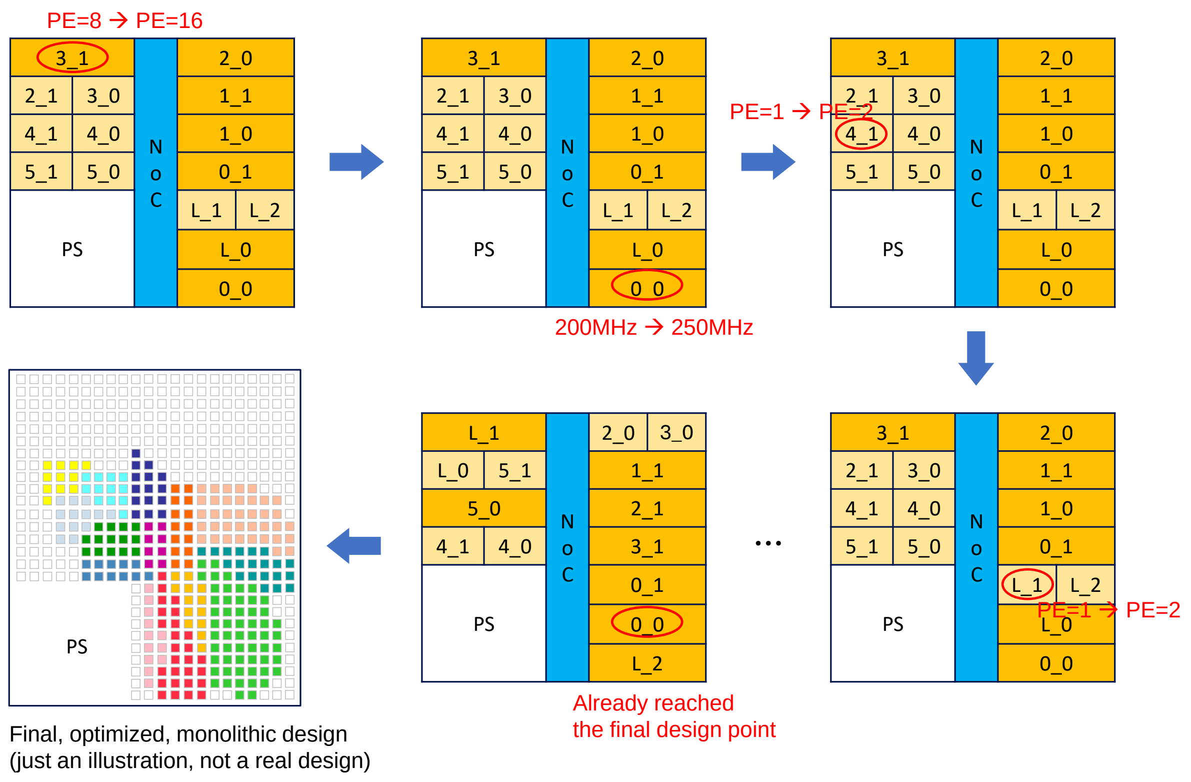

PE=8 → PE=16
3_1
2_0
2_1
3_0
1_1
4_1
4_0
1_0
5_1
5_0
0_1
NoC
L_1
L_2
PS
L_0
0_0
200MHz → 250MHz
PE=1 → PE=2
PE=1 → PE=2
Already reached the final design point
Final, optimized, monolithic design (just an illustration, not a real design)

Figure 5.8: Visualization of Incremental Refinement Example (CNN-2)

achieves a latency of 18.7 ms, and the incremental flow achieves a slightly better latency of 16.6 ms. The DSE traces for CNN benchmarks are appended in Table A.5, Table A.6, and Table A.7.

Figure 5.8 visualizes how our incremental flow iterates design points quickly in the NoC-based system and migrates to the monolithic system for the final design. Circled pages are the bottleneck operators, annotations show the parameter values for the next design point in each step. The final design point shows that 14 operators are monolithically compiled.

5.7.5. Compile Time Analysis

Figure 5.9 shows the compile time breakdown for both the monolithic flow and the incremental strategy. Time to read Vivado design checkpoints and *phys_ opt_ design* is omitted in Figure 5.9 for brevity. As expected, the incremental strategy reduces compile time in all phases from HLS to bitstream generation except for the Optical Flow benchmark in which the NoC-based system spends too many iterations to resolve the NoC bandwidth bottleneck.

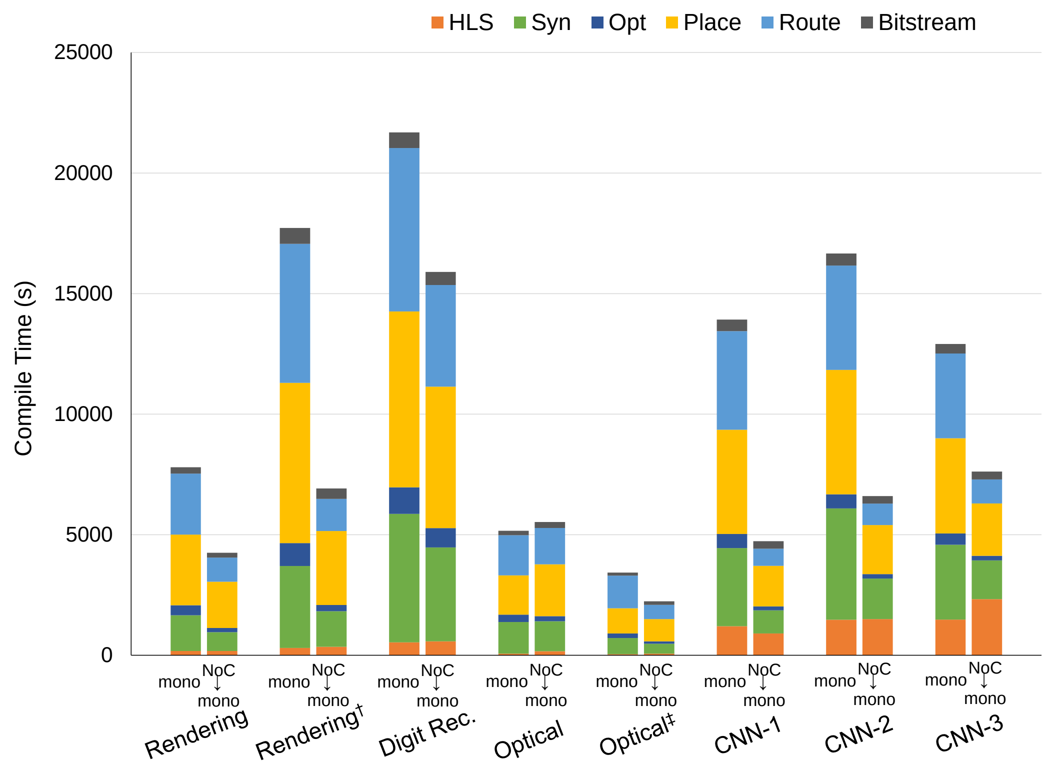


Figure 5.9: Compile Time Breakdown

In CNN benchmarks, HLS takes 16–20% of the entire compile with our strategy whereas HLS takes only 3–5% for other applications. Long HLS runtime in FINN-generated HLS codes is a known issue which the authors in [4] resolve with "RTL weights" instead of embedding weight constants in HLS codes. If HLS runtime decreases with RTL weights, we expect to see even more speedup in DSE time. For example, if we exclude HLS time in DSE time, our incremental strategy achieves 2.9× faster DSE time for CNN-1 benchmark. While we can support RTL weights for necessary modules, we keep all the source codes at the HLS level to be consistent with other benchmarks. For CNN-3, the reason why HLS time is longer in the incremental flow is that the monolithic flow failed earlier in the implementation than the incremental flow as explained before.

5.7.6. Incremental Compilation

Figure 5.10 shows the distribution of number of parallel compile jobs in the NoC-based system to show that the incremental strategy recompiles only necessary operators just like software compi-

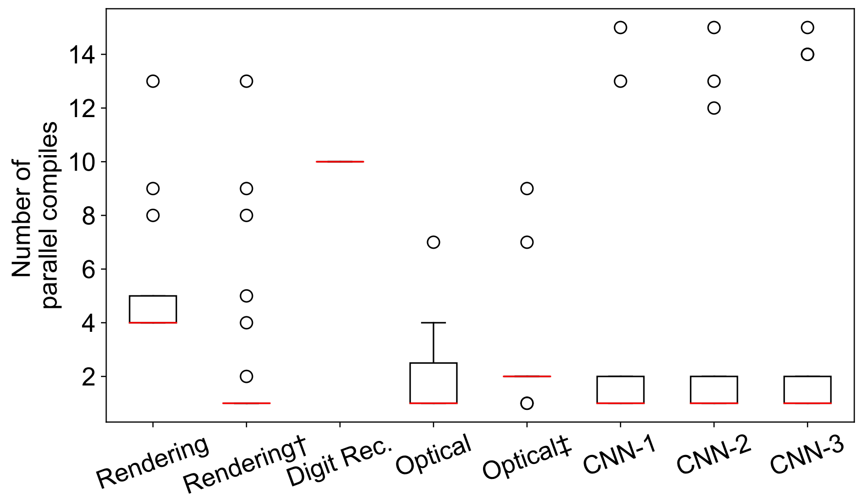


Figure 5.10: Number of Parallel Incremental Page Compile Jobs in the NoC-based System

lation. In most of the benchmarks, only one operator is incrementally refined in the NoC-based system except for Digit Recognition in which 10 identical operators are tuned together throughout the DSE. The reason why the number is not always 1 is that the first compilation runs multiple compile runs in parallel for all operators. If new operators are generated with a new design point (e.g. parallelization factor in Rendering), these new operators need to be compiled together. If the page assignment changes because the newly compiled operator consumes more resources than before, all operators with the new PR pages need to be placed and routed.

5.8. Discussion

5.8.1. Bottleneck Identification and Incremental Refinement Strategy

The motivation of the previous works on fast incremental compilation on FPGA is that when there is a step-by-step refinement in the design, the entire design does not need to be recompiled. But the key missing component is how to identify the slow operator because design parameters and synthesis options (e.g. clock frequency) together create a large design-space. Our experimental results show that our fast bottleneck identification (1) guides the users or the automation through the impactful design points that decrease application latency for both NoC-based and monolithic design flows, and (2) our incremental compilation reduces DSE time by recompiling only changed operators. Initial working designs are available in minutes, and improved designs become available every few minutes.

Except for Optical Flow, the NoC-based incremental compilation produces lower latency designs than the purely monolithic flow for any compile time budget (the incremental curves are under the monolithic curves). Despite the limitations of the NoC platform, the final performance of the monolithic designs accelerated by the NoC-based flow in the early iterations is comparable to the performance of the final, monolithic-only optimization.

In Optical Flow, optimizations to repair NoC-bandwidth limitations eliminate the compile time benefits of the incremental compilation scheme. While we currently use a single NoC-based system in the experiments, we can use different NoC-based overlays to support different bandwidth requirements per application as mentioned in Section 3.6.1.

One challenge in the idea of refining one operator at a time is that the implementation results can be noisy; sometimes an implementation for one design is not successful, but an implementation for a more complicated design could be successful. For example, in the CNN-3 benchmark, the monolithic flow fails to run the first convolutional layer at 400MHz and stops the DSE because our automation indicates it as the bottleneck but cannot improve it. However, we have seen that this is not a final, optimized design point. If we increase the parallelization factor for an *unrelated* operator or run the unrelated operator at a higher clock frequency, suddenly the tool may successfully meet the timing for the entire design in the monolithic system, running the first convolutional layer at 400MHz. In fact, for the CNN-3, our incremental flow outputs the final, monolithic design which runs the first convolutional layer at 400MHz. In the monolithic system, even a minor change in the design impacts the entire design. Such changes, even if they make the design seemingly more complex, may unexpectedly enable the tool to successfully place and route the design. In case of small negative slacks in timing, one solution would be to try different directives as discussed in Section 5.7.3.

5.8.2. Synergy with a Model-based DSE

Although we showcase that our incremental strategy can be integrated with performance/resource models in CNN benchmarks the idea can be generalized. We can categorize (1) applications that have to be evaluated in runtime with real data because of data-dependent functions (e.g. Rendering) and (2) applications that can have reasonably good starting points from model-based methods (CNNs). In (2) case, as shown in CNN benchmarks, model-based methods can reduce the design-space so that the fully mapped, NoC-based system can start from a realistic design.

5.8.3. Limitations of a Greedy Tuner

The greedy tuner in Section 5.6.2 may be stuck in the local minima. For example, when improving the application performance requires refining multiple operators, any degradation in application latency at a given design point prevents the greedy tuner from progressing. Overcoming such limitations requires a more sophisticated tuning algorithm that is capable of escaping local minima, and such algorithm requires significantly more design points than our case studies. For such an algorithm, our fast compilation framework is not sufficient because it still takes 2–3 minutes to evaluate a single design point, not less than a second. In this case too, the combination of a synthesis-based DSE like ours and a model-based DSE (Section 5.8.2) could be useful. We can use a model-based DSE to explore the design points, and periodically, we should launch our fast separate compilation framework to ensure that the current design point is consistent with the model predictions in terms of application performance, probing information, and resource utilization.

Another future direction to address the limited number of design points in a greedy tuner is to compile multiple operators in the cloud, rather than compiling one operator at a time. These operators are a set of operators that are "likely" to be refined with a more sophisticated tuning algorithm. With this approach, each iteration does not have to take a few minutes but could be less than a second, the time to partially reconfigure with the pre-generated bitstream. More data points generated with this approach could enable a more advanced tuning algorithm.

5.8.4. Analysis on FIFO Counters

In Section 5.4, our bottleneck identification scheme is explained with a high-level intuition. In this section, we present simulation results for different cases to elaborate on our strategy. Figure 5.11 shows the experiment setup. There are a sender operator, a receiver operator, and two asynchronous FIFOs. In a real application (both the NoC-based system and the monolithic system), a sender and a receiver should have both input FIFO and output FIFO. In this simple experiment setup, a sender

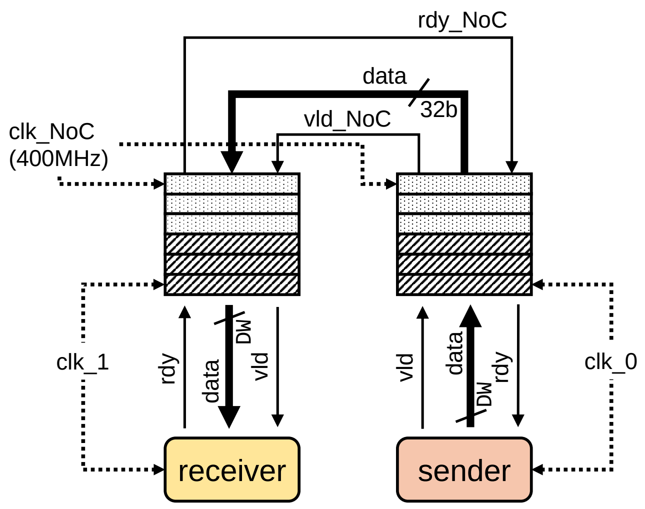


Figure 5.11: FIFO Counter Analysis Experiment Setup

has an output FIFO only, and a receiver has an input FIFO only. The NoC switches are omitted; the sender's output FIFO and the receiver's input FIFO are directly connected to handshaking signals. There are five parameters for different experiment configurations. The first one is datawidth for the operators, which is labeled with `DW` in the Figure 5.11. We can also configure the sender's write initiation interval (II) and the receiver's read II. If the sender's II is 1, it means that the sender outputs the data every cycle, and if the value is 2, it means that the sender outputs the data every two cycles. Finally, the sender and receiver can run at different clock frequencies while the NoC runs at fixed 400MHz, reading from the sender's output FIFO and writing to the receiver's input FIFO.

Figure 5.12, Figure 5.13, and Figure 5.14 show the screenshots of simulation results for different configurations, and the screenshot is labeled with (Datawidth, sender's II, receiver's II, sender's clock frequency, receiver's clock frequency). We use stall counters (Section 5.4.1, Figure 5.3 (a)) to identify the bottleneck operator, and stall counters are related to input FIFO's empty counter and output FIFO's full counter. In Figure 5.11's setup, *the stall counter for the sender is equivalent to its output FIFO's full counter* because we assume there's no stall in the sender's input FIFO. Similarly, *the stall counter for the receiver is equivalent to its input FIFO's empty counter* because we assume there's no stall in the receiver's output FIFO. The results for the experiments are summarized in Table 5.3, and the detailed explanations are followed in the next sections.

Table 5.3: Summary of Experiment Results for FIFO counters

Configuration (`DW`, sender's II, receiver's II, F_{sender}, $F_{receiver}$)	Bottleneck Operator	NoC BW bottleneck
(32, 1, 1, 200MHz, 200MHz)	not detected (correct)	not detected (correct)
(32, 1, 1, 200MHz, 400MHz)	sender (correct)	not detected (correct)
(32, 1, 1, 400MHz, 200MHz)	receiver (correct)	not detected (correct)
(64, 1, 1, 200MHz, 200MHz)	not detected (correct)	not detected (correct)
(64, 1, 1, 200MHz, 400MHz)	sender (correct)	not detected (up to interpretation)
(64, 1, 1, 400MHz, 200MHz)	-	detected (correct)
(128, 1, 1, 200MHz, 200MHz)	-	detected (correct)
(128, 1, 1, 200MHz, 400MHz)	-	detected (correct)
(128, 1, 1, 400MHz, 200MHz)	-	detected (correct)
(128, 2, 2, 200MHz, 200MHz)	not detected (correct)	not detected (correct)

Datawidth = 32 Case

In Figure 5.12 (a), the data can be transferred at a full rate. We know that both the sender and receiver do not stall because the sender's full counter and the receiver's empty counter do not increment. Therefore, neither the sender nor the receiver is identified as a bottleneck, and this bottleneck identification is correct. There is no NoC bandwidth bottleneck, and our approach, which uses full counters to detect NoC bandwidth bottleneck (Figure 5.3 (b)), should also work as intended.

In Figure 5.12 (b) too, the data can be transferred at a full rate. In this case, the receiver stalls, and we can see that the input FIFO's empty counter, which is equivalent to the receiver's stall counter in our experiment setup, increments. Therefore, the receiver should have higher stall counters than the sender, and the sender will be identified as a bottleneck. The sender is the correct bottleneck in Figure 5.12 (b) because the receiver consumes data faster than the sender produces the data.

We normalize the counters by dividing the counters by the operating frequency, but even after the normalization, the sender is correctly identified as a bottleneck in Figure 5.12 (b).

Similarly, in Figure 5.12 (c), the sender's stall counter should be higher than the receiver's stall counter, and the receiver will be correctly identified as a bottleneck. Our system still does not detect any NoC bandwidth bottleneck in both Figure 5.12 (b) and Figure 5.12 (c), and this is correct because the NoC can transfer 32b of data at a frequency of 400MHz.

Datawidth = 64 Case

In Figure 5.13 (a), the data can still be transferred at a full rate because the NoC bandwidth (32b $\times$ 400MHz) can match the sender's write rate (64b $\times$ 200MHz) and the receiver's read rate (64b $\times$ 200MHz). There is no bottleneck operator, and there is no NoC bandwidth bottleneck.

In Figure 5.13 (b), the receiver, running at 400MHz, has higher stall counters than the sender, so the sender is correctly identified as a bottleneck operator. Our system will not detect any NoC bandwidth bottleneck because in Section 5.4.2, we use only the full counters to identify the bottleneck, not the empty counters. It is true that the NoC does not keep up the rate the receiver wants the data, so we *could* consider this as the case with the NoC bandwidth bottleneck. However, the NoC already supports the full rate of the data from the sender (64 bits at 200MHz), so we conclude that the NoC is not the bottleneck in this case. Once we refine the bottleneck operator, sender, and then our system will assert the NoC bandwidth bottleneck. Depending on the interpretation, others may want to assert the NoC bandwidth bottleneck in this case, and this is why we label "up to interpretation" in Table 5.8.4 for this case. In fact, in Figure 5.3 (b), we could have considered there exists the NoC bandwidth bottleneck *if the difference in the empty counters (input FIFO's empty counters of B - output FIFO's empty counters of A) is large enough.* Then, our automation would have alerted that there exists the NoC bandwidth bottleneck in Figure 5.13 (b) case.

In Figure 5.13 (c), the sender, running at 400MHz, has higher stall counters than the receiver, so the receiver is correctly identified as a bottleneck operator. However, in this case, our system will detect the NoC bandwidth bottleneck according to the full counter difference (Figure 5.3 (b)). Our tuner prioritizes resolving the NoC bandwidth bottleneck, so it will use multiple NoC interfaces for the sender or merge the sender and the receiver after Figure 5.13 (c)'s execution. Therefore, we will ignore the bottleneck identification for the operator, and this is why we have "-" in Table 5.8.4 for this case. Let us say that we merge the sender and the receiver after Figure 5.13 (c). Merging itself does not improve the application performance, but if the merged operator runs at the maximum clock of the frequencies of the two merged operators, the application performance improves. In this case, after merging, if the receiver operator runs at 400MHz as well, the application performance improves.

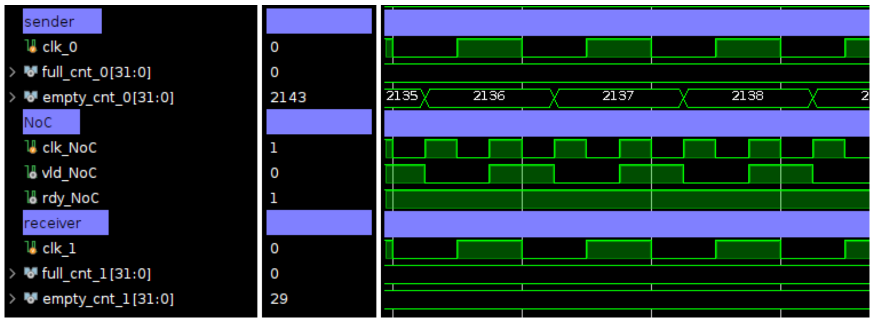
Datawidth = 128 Case

In Figure 5.14 (a), the data can not be transferred at a full rate, and the NoC bandwidth bottleneck is identified with the difference in the full counters. If two operators are merged, the application performance will improve, so the NoC bandwidth bottleneck does exist in this case. Similarly, in both Figure 5.14 (b) and Figure 5.14 (c), the NoC bottleneck is correctly detected.

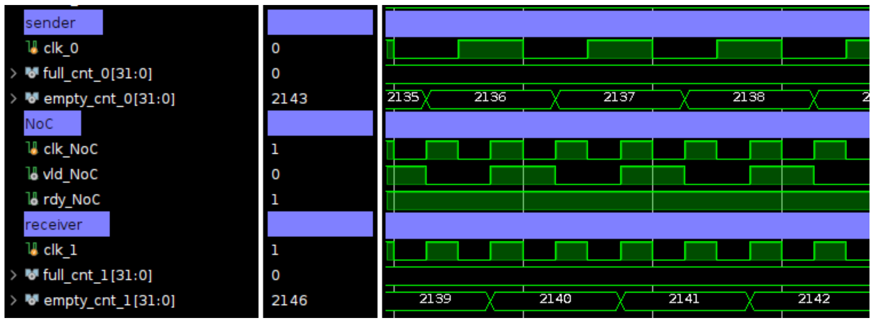
In Figure 5.14 (d), neither NoC bandwidth bottleneck nor bottleneck operator is detected. The bottleneck identification is correct because sending 128 bits with II = 2 is equivalent to sending 64 bits with II = 1. The NoC has enough bandwidth to support the data movement, and the receiver can receive the data at full rate.

5.8.5. Potential Limitations with the Operator-level Probing

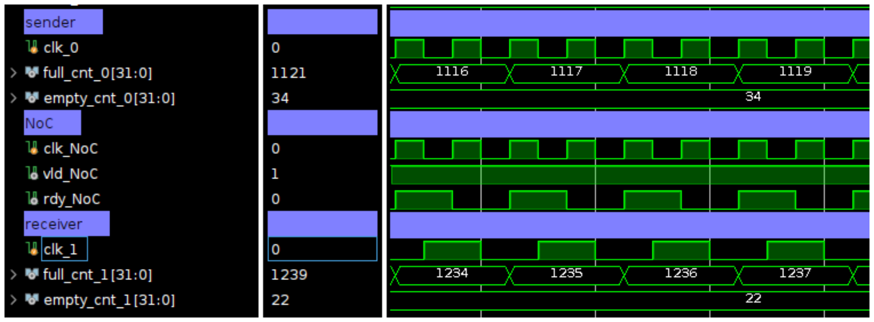
This section introduces potential problems with the bottleneck identification that instantiates stall counters per operator. We currently have full, empty, and read counters per each FIFO of an operator but have one stall counter per an operator. The problem stems from the fact that the granularity of an operator is determined by the users when they design the application. The identical stall counters for operators could imply different information depending on the internal architecture of the operator.



(a) (32, 1, 1, 200MHz, 200MHz)

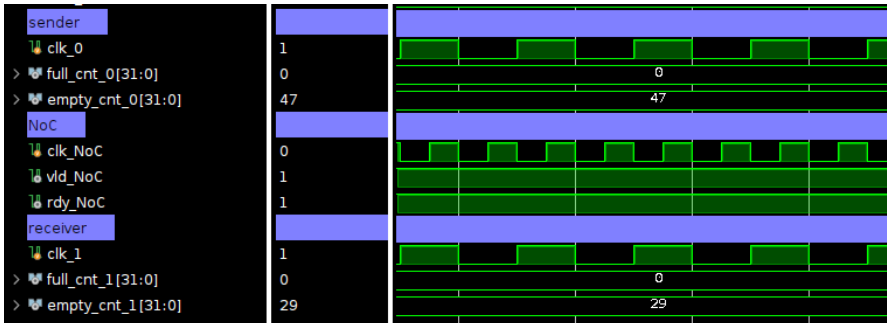


(b) (32, 1, 1, 200MHz, 400MHz)

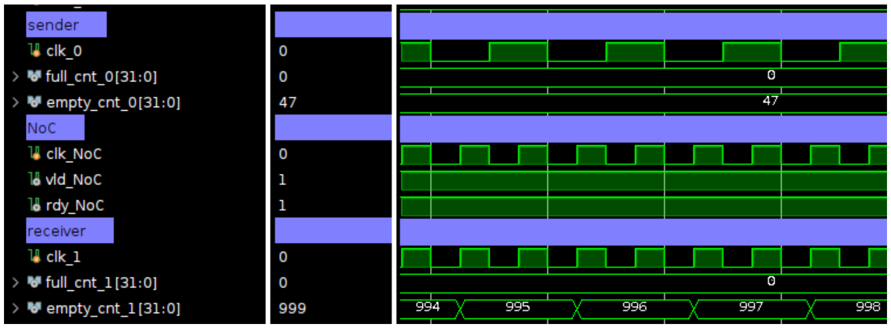


(c) (32, 1, 1, 400MHz, 200MHz)

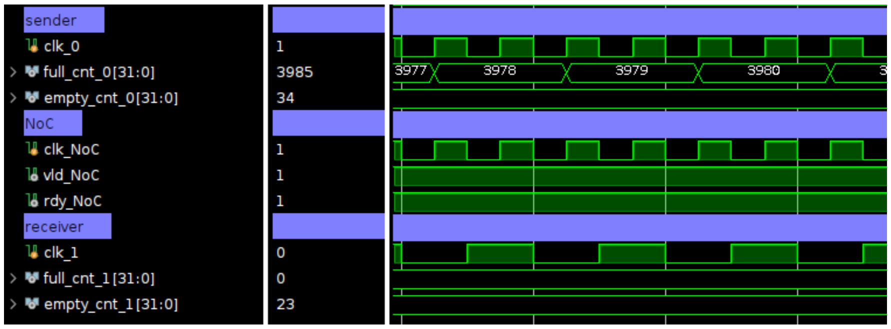
Figure 5.12: FIFO Counter Analysis, datawidth = 32
(`DW`, sender's II, receiver's II, F_{sender}, $F_{receiver}$)



(a) (64, 1, 1, 200MHz, 200MHz)

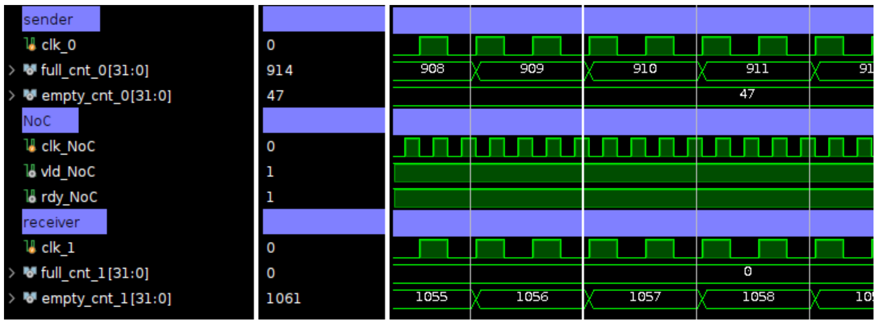


(b) (64, 1, 1, 200MHz, 400MHz)

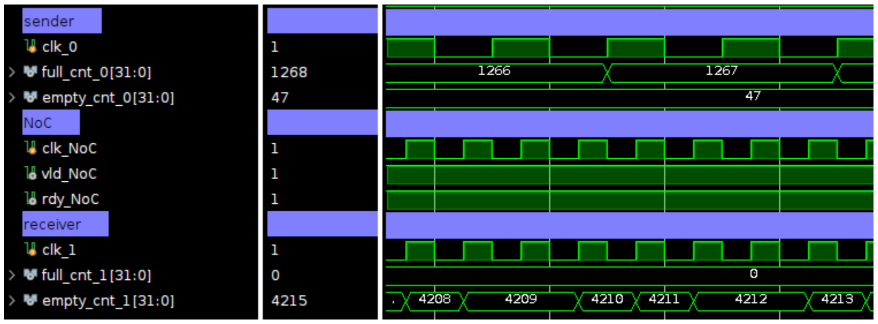


(c) (64, 1, 1, 400MHz, 200MHz)

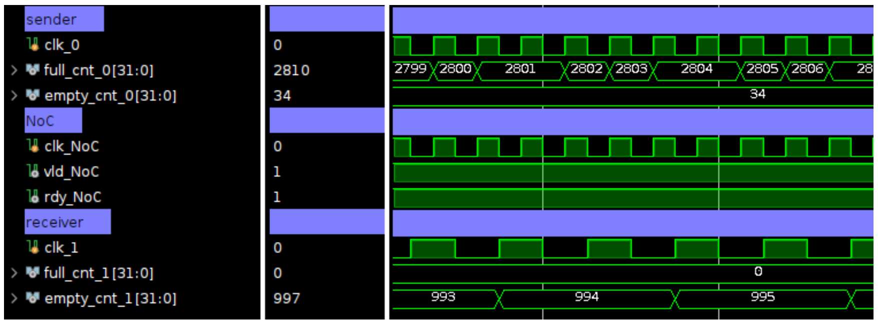
Figure 5.13: FIFO Counter Analysis, datawidth = 64
(DW, sender's II, receiver's II, F_{sender}, $F_{receiver}$)



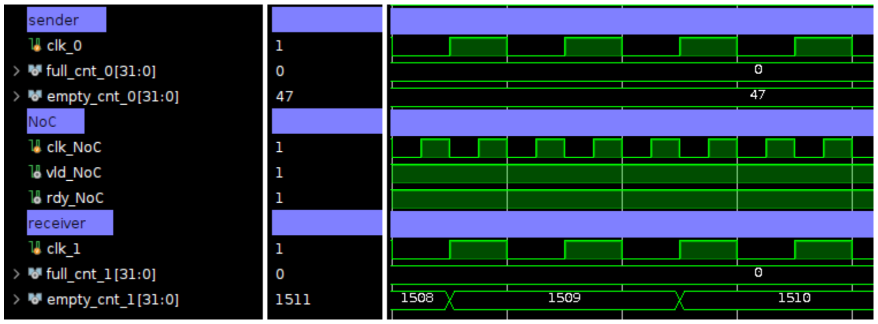
(a) (128, 1, 1, 200MHz, 200MHz)



(b) (128, 1, 1, 200MHz, 400MHz)



(c) (128, 1, 1, 400MHz, 200MHz)



(d) (128, 2, 2, 200MHz, 200MHz)

Figure 5.14: FIFO Counter Analysis, datawidth = 128
(DW, sender's II, receiver's II, F_{sender}, $F_{receiver}$)

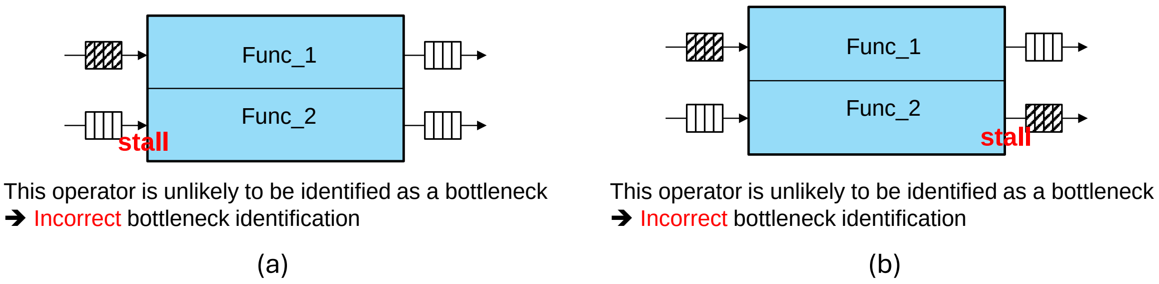

Func_1
Func_2
stall
This operator is unlikely to be identified as a bottleneck
➔ Incorrect bottleneck identification
(a)
Func_1
Func_2
stall
This operator is unlikely to be identified as a bottleneck
➔ Incorrect bottleneck identification
(b)

Figure 5.15: An Operator with Sub-functions

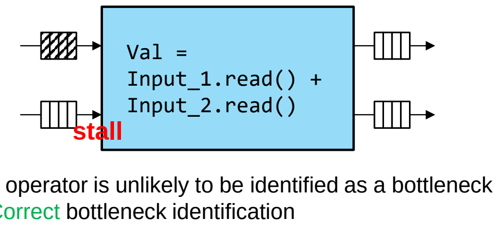

Val =
Input_1.read() +
Input_2.read()
stall
This operator is unlikely to be identified as a bottleneck
➔ Correct bottleneck identification

Figure 5.16: An Operator without Sub-functions

An Operator with Sub-functions

Figure 5.15 and Figure 5.16 illustrate examples of incorrect and correct bottleneck identification, respectively, despite having similar stall counters. In Figure 5.15, a single operator consists of two independent sub-functions. In both cases, Func_1 has room for improvement but Func_2 stalls. In Figure 5.15 (a), Func_2 stalls because it does not receive the input data, and in Figure 5.15 (b), Func_2 stalls because it cannot output the data. In our bottleneck identification, because one of the inputs or the outputs stall, the stall counters for both cases are high, and in both cases, this operator will not be identified as a bottleneck although Func_1 needs to be improved.

One may question why we assert the stall condition *at least* one input or output stalls. The reason is that if we have a code segment like `Input_1.read() + Input_2.read()` in the operator as shown in Figure 5.16, if either `Input_1` or `Input_2` is not available, it is true that this operator stalls. The first issue with this problem is that we have a single stall counter per operator. This issue can be easily resolved by instantiating a stall counter per each input or output port at the cost of using more resources. Nonetheless, the second issue is that even if we have a stall counter per port, there is no visibility inside the operator. If we have a stall counter for all the ports, for both Figure 5.15 (a) and Figure 5.16 cases, we know that the second input stalls, but without visibility inside the operator, we do not know whether the large stall counter for the second input indicates the operator's stall (Figure 5.16) or some subfunctions of the operator can be improved (Figure 5.15 (a)). If we can determine whether an operator belongs to Figure 5.15 or Figure 5.16 through code analysis at the HLS level or analysis on Intermediate Representation (IR), we can disambiguate what the high stall counter implies.

Figure 5.15's cases may not be the most natural granularity of an operator because we could have created separate operators for Func_1 and Func_2. One example where users *need* to have independent sub-functions in an operator is when sub-functions share some storage or buffer structure. Figure 5.17 shows two operators, Operator_1 and Operator_2. Operator_1 consists of two independent sub-functions, Func_1 and Func_2, and Operator_2 consists of one sub-function, Func_3. Func_1 and Func_2 share some storage which Func_1 reads the data from, writes the data to, and produces an output (Step 1). Then, Func_3 computes to produce an output (Step 2), which is the input of Func_2. Finally, Func_2 reads data from the storage, writes data to the storage, and produces the output (Step 3). In this example, Func_1 and Func_2 are independent except that they share a data structure indicated as "Shared" in Figure 5.17. In Figure 5.15, we have already shown that if an operator has multiple independent sub-functions, our stall counters may not work correctly, Therefore, *it is not the feedback loop that causes a problem with our bottleneck identification but the lack of visibility inside a single operator that causes a problem* as shown in Figure 5.15. However, one scenario where users inevitably have independent sub-functions in an operator is when multiple sub-functions share some data structure as shown in Figure 5.17, and this creates a feedback loop between operators.

Sub-functions with Different Latencies

To further illustrate how two independent sub-functions could confuse our bottleneck identification, in Figure 5.18, we present different cases when sub-functions have different latencies. Figure 5.18 is an extension of Figure 5.15 that has two independent sub-functions. Explanations below each

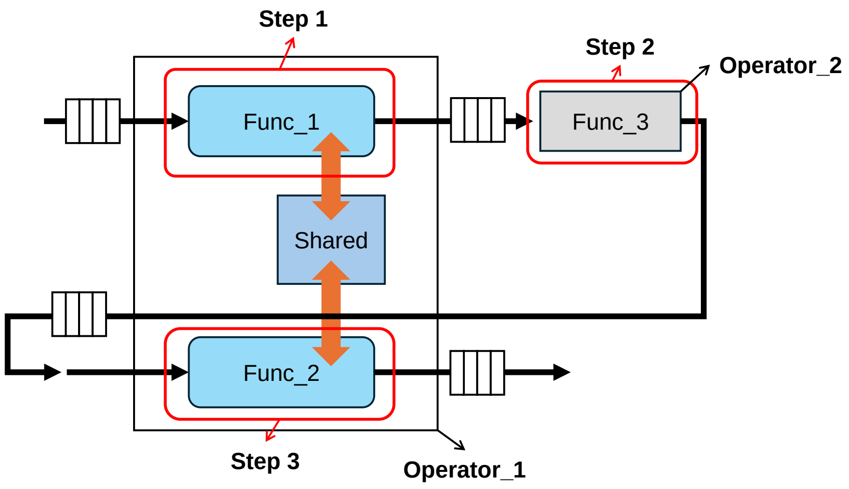


Figure 5.17: An Example of a Feedback Loop between Operators

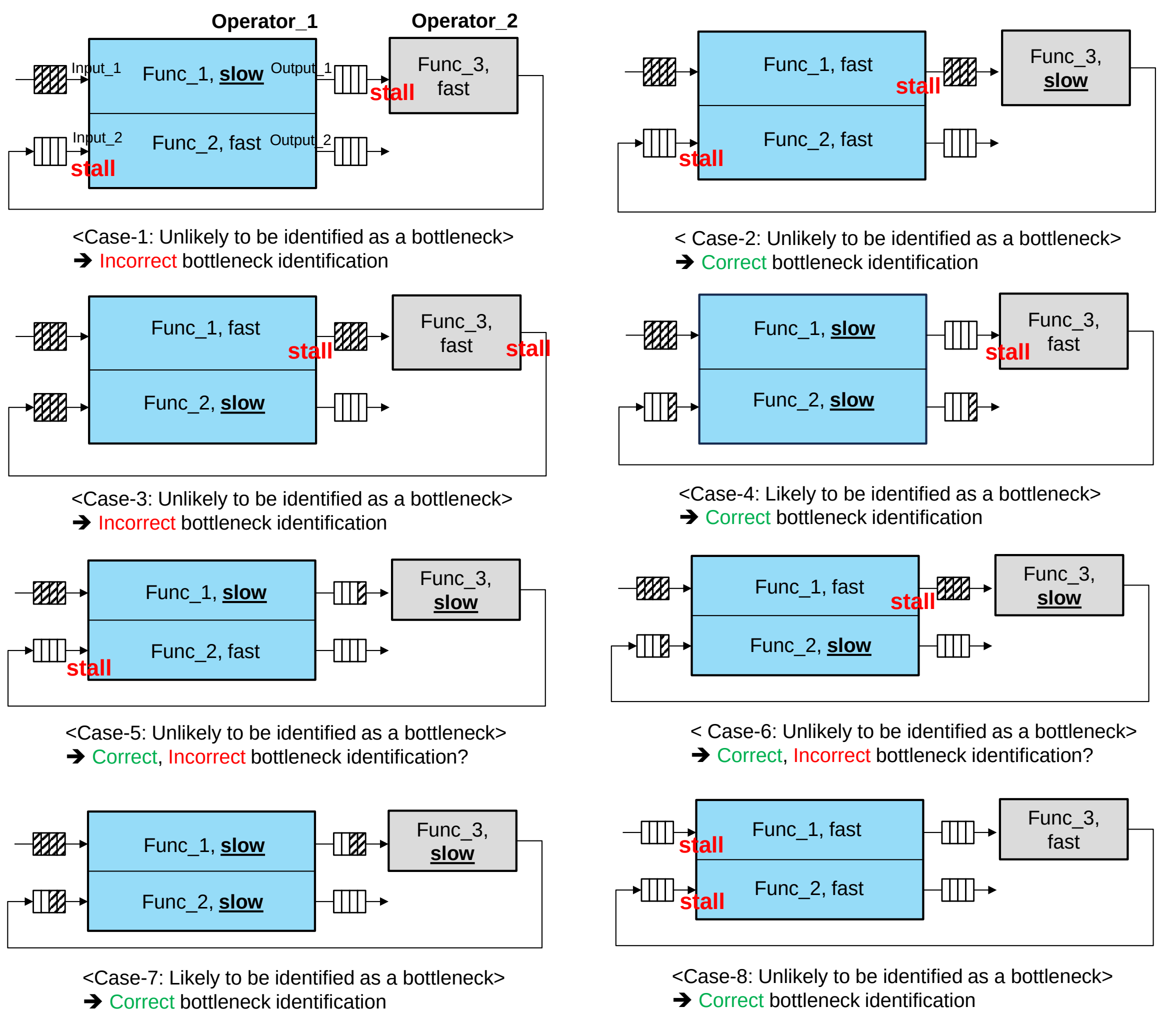
case focus on bottleneck identification of Operator_1. We assume that there is one stall counter per operator and we do not have visibility inside the operator.

In Case-1, Operator_1 is unlikely to be identified as a bottleneck because Input_2, the input of Func_2, stalls. This is an incorrect bottleneck identification because if we increase the clock frequency of Operator_1, for example, then the overall application performance is likely to improve since Func_1 processes data faster.

In Case-2, Operator_1 is unlikely to be identified as a bottleneck because Output_1 stalls. This may be a correct bottleneck identification because Operator_2 needs to be accelerated to improve the overall performance.

In Case-3, Operator_1 is unlikely to be identified as a bottleneck because Output_1 stalls. This is an incorrect bottleneck identification because Operator_1's Func_2 needs to be accelerated to improve the overall performance.

In Case-4, Operator_1 is likely to be identified as a bottleneck. This may be a correct bottleneck identification because Operator_1's Func_1 and Func_2 need to be accelerated to improve the overall performance.



<Case-1: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification

< Case-2: Unlikely to be identified as a bottleneck>
➔ Correct bottleneck identification

<Case-3: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification

<Case-4: Likely to be identified as a bottleneck>
➔ Correct bottleneck identification

<Case-5: Unlikely to be identified as a bottleneck>
➔ Correct, Incorrect bottleneck identification?

< Case-6: Unlikely to be identified as a bottleneck>
➔ Correct, Incorrect bottleneck identification?

<Case-7: Likely to be identified as a bottleneck>
➔ Correct bottleneck identification

<Case-8: Unlikely to be identified as a bottleneck>
➔ Correct bottleneck identification

Figure 5.18: Cases with a Feedback Loop, Sub-functions with Different Latencies

In Case-5, Operator_1 is unlikely to be identified as a bottleneck because Input_2 stalls. Both Operator_1's Func_1 and Operator_2 need to be accelerated to improve the overall performance. If Operator_1 is not identified as a bottleneck, then Operator_2 needs to be identified as a bottleneck. Because Operator_2 does not stall, Operator_2 is likely to be identified as a bottleneck. Once Operator_2 is refined, the system becomes equivalent to Case-1, which is incorrect bottleneck identification. The first bottleneck identification of Operator_2 can be considered correct, but the design eventually results in incorrect bottleneck identification.

Case-6 is similar to Case-5. Operator_2 is likely to be identified as a bottleneck first. After refinement on Operator_2, the system is equivalent to Case-3, which is incorrect bottleneck identification. Similarly, the first bottleneck identification of Operator_2 can be considered correct, but the design eventually results in incorrect bottleneck identification.

In Case-7, Operator_1 is likely to be identified as a bottleneck. Both Operator_1 and Operator_2 need to be accelerated to improve the overall performance, so we assume that this is a correct bottleneck identification.

In Case-8, Operator_1 is unlikely to be identified as a bottleneck because Input_1 and Input_2 stall, and this is a correct bottleneck identification.

Sub-functions with Different Read/Write Rates

The examples in Figure 5.19 are another extension of Figure 5.15 and illustrate different cases when sub-functions have different read/write rates. Explanations below each case focus on bottleneck identification of Operator_1. "100:1" indicates that it takes 100 inputs every cycle to output 1 output. Similarly, "1:1" indicates that it takes 1 input every cycle to output 1 output. All other operators not shown in the graph are assumed to have read and write rates of 1:1.

In Case-1, Operator_1 is unlikely to be identified as a bottleneck because Input_2 stalls. This is an incorrect bottleneck identification because we need to accelerate Operator_1's Func_1 to improve the overall performance.

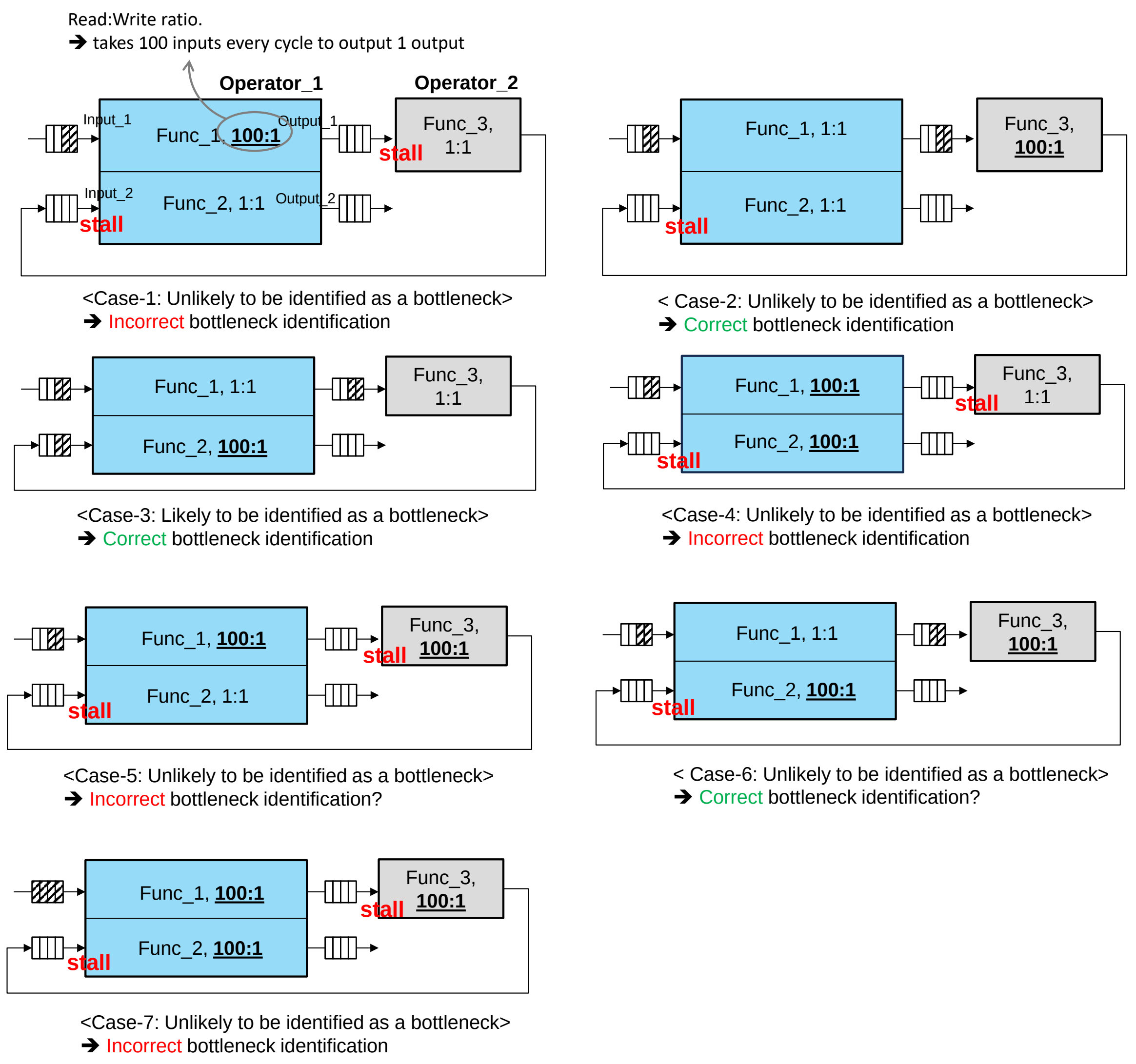

Read:Write ratio.
➔ takes 100 inputs every cycle to output 1 output
Operator_1
Operator_2
Input_1
Func_1, 100:1
Output_1
Func_3, 1:1
stall
Input_2
Func_2, 1:1
Output_2
stall
<Case-1: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification
Func_1, 1:1
Func_3, 100:1
Func_2, 1:1
stall
< Case-2: Unlikely to be identified as a bottleneck>
➔ Correct bottleneck identification
Func_1, 1:1
Func_3, 1:1
Func_2, 100:1
<Case-3: Likely to be identified as a bottleneck>
➔ Correct bottleneck identification
Func_1, 100:1
Func_3, 1:1
stall
Func_2, 100:1
stall
<Case-4: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification
Func_1, 100:1
Func_3, 100:1
stall
Func_2, 1:1
stall
<Case-5: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification?
Func_1, 1:1
Func_3, 100:1
Func_2, 100:1
stall
< Case-6: Unlikely to be identified as a bottleneck>
➔ Correct bottleneck identification?
Func_1, 100:1
Func_3, 100:1
stall
Func_2, 100:1
stall
<Case-7: Unlikely to be identified as a bottleneck>
➔ Incorrect bottleneck identification

Figure 5.19: Cases with a Feedback Loop, Sub-functions with Different Read/Write Rates

In Case-2, Operator_1 is unlikely to be identified as a bottleneck because Input_2 stalls. This is a correct bottleneck identification because Operator_2 needs to be accelerated to improve the overall performance.

In Case-3, Operator_1 is likely to be identified as a bottleneck. This is a correct bottleneck identification because Operator_1's Func_2 needs to be accelerated to improve the overall performance.

In Case-4, Operator_1 is unlikely to be identified as a bottleneck. This is an incorrect bottleneck identification because Operator_1's Func_1 and Func_2 need to be accelerated to improve the overall performance.

In Case-5, Operator_1 is unlikely to be identified as a bottleneck. Operator_1's Func_1 and Operator_2's Func_3 both need to be accelerated to improve the performance. In this case, Operator_2 is also unlikely to be identified as a bottleneck. Therefore, we consider this as an incorrect bottleneck identification.

In Case-6, Operator_1 is unlikely to be identified as a bottleneck. Operator_1's Func_2 and Operator_2's Func_3 both need to be accelerated to improve the performance. In this case, Operator_2 is likely to be identified as a bottleneck, and once Operator_2 is improved, Operator_1 can be identified as a bottleneck. Therefore, we consider this as a correct bottleneck identification.

Lastly, in Case-7, Operator_1 is unlikely to be identified as a bottleneck. Operator_2 is also unlikely to be identified as a bottleneck. This is an incorrect bottleneck identification.

Summary

Examples in Figure 5.18 and Figure 5.19 represent only subsets of all possible cases. In more realistic applications, sub-functions will have different latencies *and* different read/write rates. The purpose of these thought experiments is to show that when an operator contains independent sub-functions, our stall counters may not work as intended. The root of the operator-level probing lies in the fact that operators are designed by the users, and stall counters could imply different information depending on the internal architecture of the operators. As stated earlier, the first step to resolve this issue is to have stall counter information per port instead of per operator. The second and more challenging step is to determine whether each input and output port are correlated or not. With this additional information, we can treat sub-functions as independent operators if they are uncorrelated, and we can apply the same bottleneck identification scheme based on stall counters.

5.9. Conclusions

FPGA development is different from software programming because of the lack of visibility on the inner state of the design and slow compilation. While there exist previous works on hardware profiling and fast FPGA compilation, both efforts need to be integrated to support a software-like development experience for FPGAs. Our integrated stream FIFO counters automatically identify bottlenecks that limit performance. Our case studies show that our incremental refinement strategy using these lightweight counters and fast incremental compilation on acyclic designs iterates initial yet important design points in 2–3 minutes and achieves 1.3–2.7$\times$ faster DSE time compared to the monolithic compilation while improving the kernel execution latency by 2.2–12.7$\times$.

CHAPTER 6

ASYMMETRY IN BUTTERFLY-FAT-TREE NOC

The key to our fast incremental refinement strategy is the continuity between the NoC-based system and the monolithic system. What we have not discussed much is congestion within the packet-switched network. When workloads are highly unbalanced, multiple packets may contend for limited channels, causing some packets to experience delays. The congestion degrades the application performance in the NoC-based system. Furthermore, the bottleneck identification based on FIFO counters in the previous chapter may not work as intended. For example, if packets arrive late at the input FIFO due to NoC congestion, the stall counter at the destination operator increments. This could lead to an incorrect interpretation that the destination operator was waiting for the slower source operator, when in reality, two operators may have been running at similar rates. The only reason for the stall in the destination operator was NoC congestion.

While the simplest solution is to have richer switches in the NoC, this solution reserves more FPGA resources for the NoC, widening the gap between the NoC-based system and the monolithic system. In this chapter, to expand the design-space of the Butterfly-Fat-Tree (BFT) NoC we have used in the previous chapters, we explore asymmetry in the BFT NoC. We aim to support heterogeneous bandwidth, given the similar resource budget with the traditional, symmetric BFT NoC. This chapter was previously published in [Dongjoon Park, Zhijing Yao, Yuanlong Xiao, and André DeHon. Asymmetry in Butterfly Fat Tree FPGA NoC. International Conference on Field-Programmable Technology. 2023.] [80]. I led the project and was in charge of the system implementation.

6.1. Background

While our fast separate compilation can use any NoC topology, we choose BFT NoC. Although mesh [76, 43] may be more layout-friendly in today's island-style FPGA routing architecture [57, 17], BFT is known to be cost-effective [66, 29, 30] and outperforms other state-of-the-art network topologies [48].

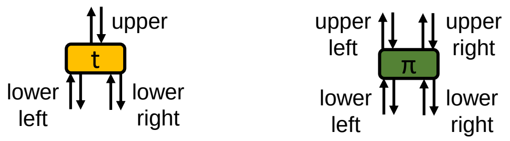


Figure 6.1: t switch and π switch

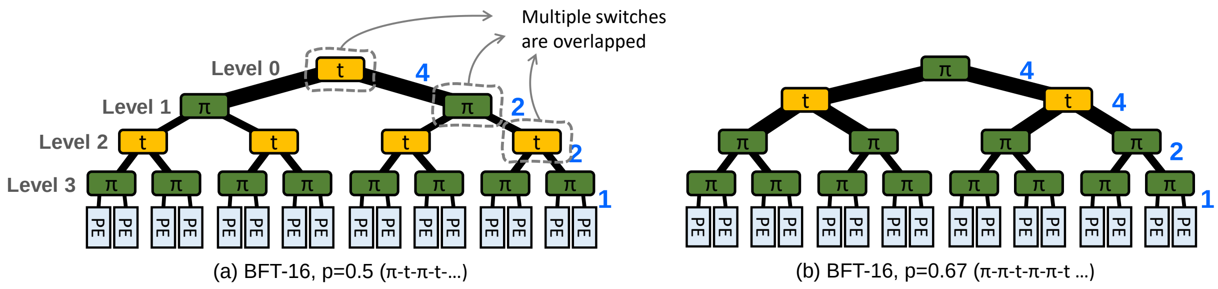

Multiple switches are overlapped
Level 0
Level 1
Level 2
Level 3
4
2
2
1
PE
(a) BFT-16, p=0.5 (π-t-π-t-...)
4
4
2
1
(b) BFT-16, p=0.67 (π-π-t-π-π-t ...)

Figure 6.2: Symmetric BFT-16 with Different p Values

BFT has a hierarchical structure with the PEs located on the lowest level. The network packet should contain address bits, and depending on the address bits, switches in each level determine where the packet should be directed.

The wiring capacity of a network can be described with Rent parameter p [59] where the larger value of p indicates the larger bisection bandwidth ($IO = cn^p, 0 \leq p \leq 1$). The primitive building blocks for BFT are t switches that have one parent port and π switches that have two parent ports as shown in Figure 6.1. *Arity* refers to the number of the children ports, and in this paper, we consider arity-2 switches like the ones in Figure 6.1 while different arity values are possible in the BFT architecture. For example, authors in [60] show that BFT with Arity-4 can be mapped on FPGA with less LUT cost compared to BFT with Arity-2 by utilizing wide muxes available in modern FPGA fabric. One differentiating factor of BFT compared to other topologies is that the bandwidth of each level of BFT can be configured by properly selecting t switches and π switches [31]. If we want more bandwidth between the specific levels, we can simply compose the layer with π switches. This flexibility sharply contrasts with other topologies like mesh where all channel widths need to increase together to support more bandwidth.

Figure 6.2 shows BFTs with 16 PEs. The width of connection between layers in Figure 6.2 represents the communication bandwidth, and in the Fat-Tree-based topology, the communication is thicker at the higher level in the hierarchy. Blue numbers represent the channel widths in the communication, so in the non-lowest level in the hierarchy, there are multiple switches that make up each switching node. BFT-16 with $p = 0.67$ (Figure 6.2 (b)) provides more bandwidth between level 1 and level 2 than BFT-16 with $p = 0.5$ (Figure 6.2 (a)). In [48], Kapre proposes localized deflection routing to adapt the BFT routing algorithm to lightweight, bufferless Hoplite style [50, 51]. Our fast NoC-based system [79, 99, 78, 77] also uses the packet-switched, deflection-routed BFT from [48]. In this chapter, too, we build upon deflection-routed BFT.

BFT's hierarchical structure offers finer-grained control on network channel bandwidth compared to other topologies. Nevertheless, in a symmetric BFT, like the ones in Figure 6.2, since each level is homogeneously composed of either t switches or π switches, to provide more bandwidth, all t switches in the level are replaced with π switches, requiring more switch area. For instance, when placed and routed on Xilinx UltraScale+ ZU9EG, BFT-16 with $p = 0.67$ (Figure 6.2 (b)) costs 7276 LUTs and 5991 FFs while BFT-16 with $p = 0.5$ (Figure 6.2 (a)) costs 6248 LUTs and 5223 FFs.

6.2. Motivation

When a graph is mapped on a BFT, one fast, heuristic-based approach is to (1) bi-partition the graph in a way that the inter-partition communication is minimized to prevent the unnecessary traffic over the NoC and (2) assign each sub-graph to each subtree of a BFT [49]. In section 4.5, we use the recursive bi-partitioning to perform a fast page assignment. However, there is no guarantee that the communication in each partition is the same. After the bi-partitioning, some portion of the graph could require more bandwidth.

Figure 6.3 is a real example of graph workloads from [67] where one-fourth of the nodes are located in each quadrant, and the thickness of the edge represents the communication volume. In Figure 6.3, we use `metis` [54] to cluster the graph workloads into 256 parts using a recursive bisection scheme with the objective to minimize the edge cuts. `metis` numbers each node, and the labels imply how the graph is bi-partitioned. For example, the indexes from 1 to 128 belong to one bisected part,

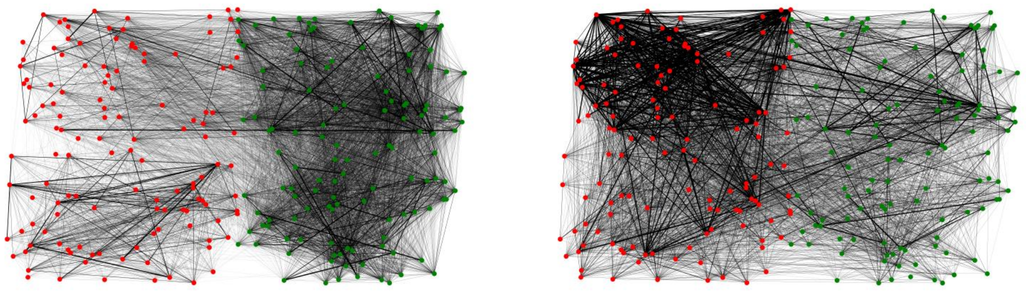


Figure 6.3: Examples of Unbalanced Realistic Workloads after Partitioning

and the indexes from 129 to 256 belong to another bisected part. Therefore, in Figure 6.3, node indexes from 1 to 64 are mapped to the one quadrant, indexes from 65 to 128 are mapped to another quadrant, and so on. x and y values are randomly generated within each quadrant for each node. In Figure 6.3, inter-partition communication is reduced with bi-partitioning (between the green nodes and the red nodes), but the communication requirements of the one-half are heavier than the other half. We would like to selectively provide more bandwidth to specific subtrees, but in traditional BFT (Figure 6.2), this customization is not possible, and more bandwidth in one subtree leads to more bandwidth in all other subtrees, consuming more FPGA resources.

6.3. Asymmetric BFT

To support heterogeneous bandwidth with the similar resource usage, we explore asymmetry in the BFT. We define an asymmetric BFT as a BFT that has different switches in a given level. Figure 6.4 is an example of an asymmetric BFT that has 256 PEs. The color scheme for the switches is consistent with that of Figure 6.2, and PEs are omitted for brevity. Type 1's subtree consists of π-π-π-π-π-π, Type 2's subtree consists of π-t-π-t-π-π, and Type 3's subtree consists of t-π-t-π-t-π-t switches. Therefore, the Type 1 subtree is denser and provides more bandwidth for the PEs, and the Type 3's subtree is sparser and provides less bandwidth. We expect that this architecture leads to better performance when one partition of the graph communicates more heavily than another.

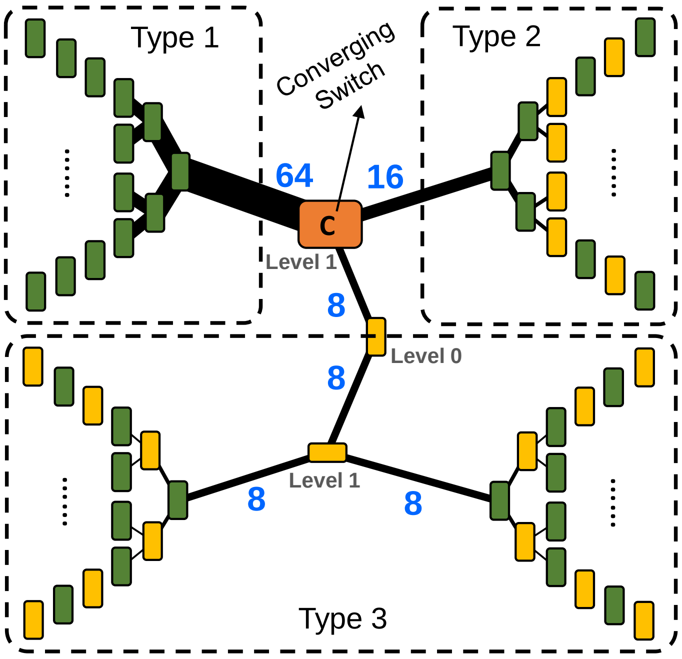


Figure 6.4: Example of Asymmetric BFT-256

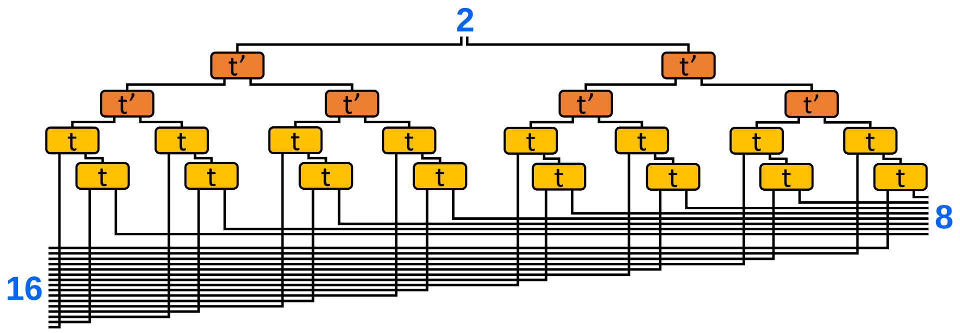


Figure 6.5: 16-8-2 Converging Switch Built with t Switches and $t - random$ (t') switches
Note: The two lowest levels consist of t switches, and the upper levels consist of $t - random$ switches.

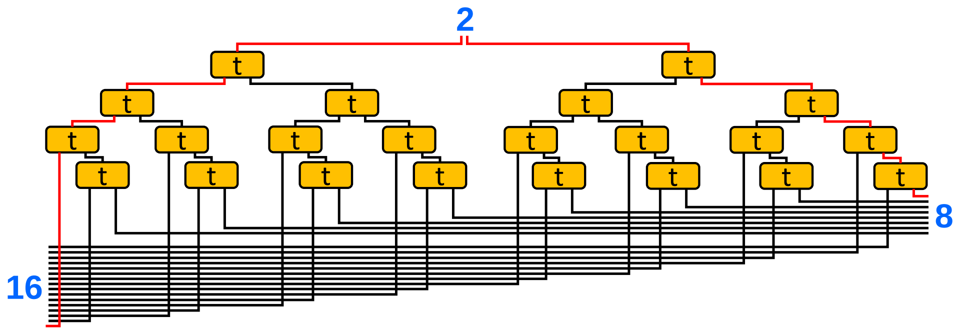


Figure 6.6: 16-8-2 Converging Switch Built with only t Switches
Note: Only the leftmost path or the rightmost path is the desirable direction when the packet travels from the upper level to the lower level.

In Figure 6.4's example, Type 1's subtree results in the channel width of 64 to the level 1, and Type 2's subtree results in the channel width of 16 to the level 1. Type 3's subtree results in the channel width of 8 to the level 0. Thus, we need a component in the level 1 that reduces the channel width to match the bandwidth of the top and the bottom subtrees. We introduce a *converging switch* that reduces the channel width from the top subtrees to the bottom subtree.

While it is possible to build the converging switch out of the original t switches only as shown in Figure 6.6, using only t switches would not spread out the traffic to exploit the wider channels; instead, it would concentrate the traffic and leave portions of the rich bandwidth unused. Therefore, we use standard t switches in the lowest level and use $t - random$ switches (t') in the higher levels as shown in Figure 6.5. In the switches of deflection-routed BFT, there are the *desirable direction* where the packet wants to go and the *final direction* where the packet ends up going if the contention exists. As all the t switches in the converging switch belong to the same level (level 1 in Figure 6.4's case), in the strawman implementation of the converging switch that uses only standard t switches (Figure 6.6), the *same* bits in the packet are used to specify the desirable direction in all the stages of the converging switch. This means the desirable direction for a packet that is climbing down is either the lower left for all the t switches or the lower right for all the t switches (colored red in Figure 6.6), potentially causing congestion inside the converging switch. In the $t - random$ switch, on the other hand, the desirable downward direction ignores the destination bit and is set to the lower left at the one cycle and set to the lower right at the next cycle, alternately. These $t - random$ switches in the non-lowest levels of the converging switch spread out the traffic, and in the lowest level, the t switches send the packet based on the address bits in the packet to make sure that the packet is delivered to the correct subtree. The only difference between a $t - random$ switch and a t switch is the desirable downward direction (left or right). The rest of the architecture, like the packet arbitration, stays the same. For example, a deflected packet takes priority and is immediately turned back in the next cycle, as done in the base deflection-routing. Because we build on top of the deflection-routed BFT, deflection-routed scheme's randomness already requires reorder buffers (reassembly buffers) in PEs. Our $t - random$ switch adds no new requirements for reordering beyond those that already exist for the deflection-routed BFT.

Table 6.1: Symmetric BFT-256s (S0, S1) and Asymmetric BFT-256s (AS0, AS1) Example (52b, single flit packet)

	LUTs	Switch Composition	Converging Switch	
			Composition	LUTs
S0	122778	$p = 0.5$ π-t-π-t-π-t-π	-	-
S1	143870	$p = 0.5$ π-π-t-t-π-π-t	-	-
AS0	142896	st-0,1: π-π-π-t-π-π-c	32-32-8	15829
		st-2,3: t-π-t-π-t-π-t		
AS1	143029	st-0: π-π-π-π-π-π-c	64-16-8	21480
		st-1: π-t-π-t-π-π-c		
		st-2,3: t-π-t-π-t-π-t		

st-i: subtree-i, c: Converging switch

All the PEs in an asymmetric BFT can still communicate with each other but with higher bandwidth on specific subtrees and lower bandwidth on other subtrees. Figure 6.5 is a converging switch that reduces the channel width of 16 and 8 to the channel width of 2, but the architecture can be extended to other power of two combinations, like 64-16-8 in Figure 6.4.

6.4. Methodology

Table 6.1 describes two symmetric BFTs (S0, S1) and two asymmetric BFTs (AS0, AS1) that we use for the evaluation. These BFTs are all deflection-routed BFTs [48]. The number of PEs is 256. When BFT-256 has four subtrees with 64 PEs (st-0,1,2,3), both AS0 and AS1 have two dense subtrees and two sparse subtrees. Subtrees of different densities are connected with a converging switch. The switch composition column refers to the switch type from the lowest level (leftmost) to the second-highest level (rightmost), so Figure 6.4's asymmetric BFT corresponds to AS1. For Rent parameter p of symmetric BFTs (S0, S1), $p = 0.5$ is chosen because $p = 0.5$ is known to be area-universal, meaning that the networking resources are relatively well-balanced and scalable with the computation [66, 28]. While we provide two examples of asymmetric BFTs, the idea can be extended to any number of different subtrees with appropriate converging switches.

We run synthesis, placement and routing with Xilinx Vivado 2022.1 targeting UltraScale+ ZU9EG FPGA to extract resource usage. Dummy PEs are attached to the NoC for testing purposes. A packet consists of a single flit with 1 valid bit, 8 bits of PE address, an 11-bit sequence number and 32 bits of data. The packet composition can change depending on design requirements, but it should be the same for both symmetric and asymmetric BFTs for a fair comparison.

The converging switch columns in Table 6.1 shows the configuration of the converging switch and the resource usage of the converging switch. AS1's converging switch (channel width of 64 is converged to the channel width of 8) has a deeper hierarchy than AS0's (channel width of 32 is converged to the channel width of 8), and the resource usage for the converging switch increases accordingly. We have a script to generate Verilog codes for asymmetric BFTs, given the switch configurations. Asymmetric BFTs for the evaluation are selected so that they consume fewer LUTs than symmetric BFT-256 (S1) to be a fair comparison with the symmetric BFTs.

Table 6.2: Worst Negative Slack (ns) for Different Switch Types when Placed and Routed on ZU9EG (Packet Size = 52b)

SW type	clock period (ns)						
	1.0	1.2	1.4	1.6	1.8	2.0	2.2
t	-1.21	**0.012**	**0.174**	**0.244**	**0.381**	**0.459**	**0.585**
$t - random$	-0.181	-0.055	**0.157**	**0.168**	**0.328**	**0.487**	**0.703**
π	-1.217	-0.878	-0.720	-0.434	-0.340	-0.003	**0.053**

We also run synthesis, placement and routing on t switches, $t - random$ switches, and π switches separately, setting the switch level as 7 (the lowest level). Table 6.2 shows the worst negative slacks when switches are routed with different system clock speeds. Max clock frequencies of t switch, $t - random$ switch, and π switch are estimated as 833 MHz, 769 MHz, 476 MHz based on the clock period and the slack. As π switch has more complex routing within the switch, max clock frequency is slower than t switch, consistent with the results from [48]. LUT costs for t switch, $t - random$ switch, and π switch are 171, 172, and 285–287 respectively. FF costs are 156, 157, and 209. The $t - random$ switch has almost the same logic complexity as the t switch; since the $t - random$ is still faster than the π switch, it will not limit the clock frequency of the system with proper floorplanning.

We use `iverilog` to run simulations for realistic workloads and synthetic traffic patterns. We have an option to generate synthetic test patterns every simulation run, but for the repeatability, we pre-generate the test patterns and use the same pattern across all BFTs. After the simulation is

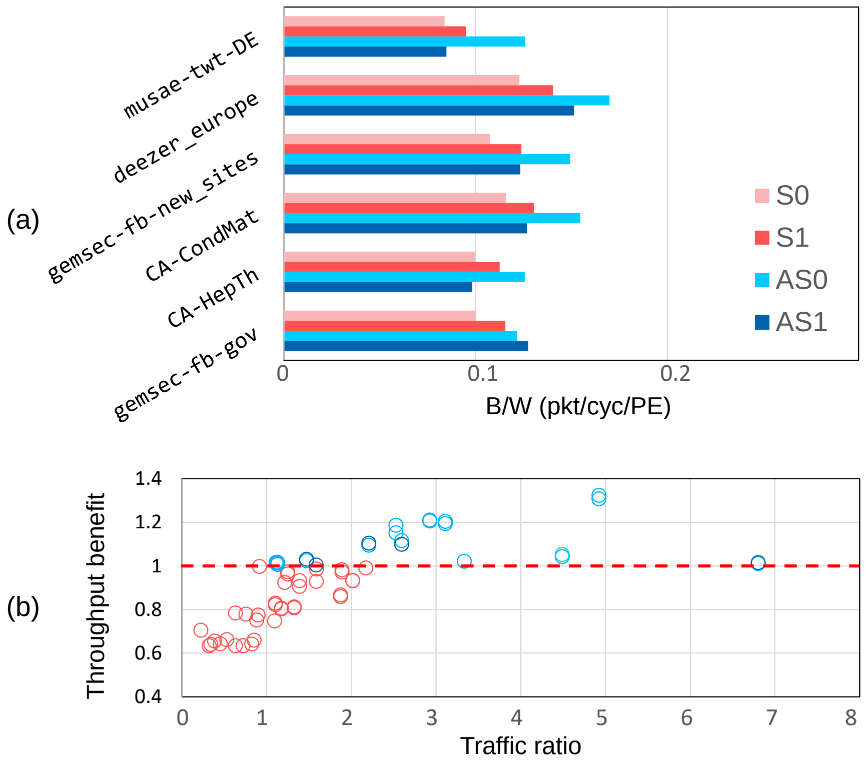


Figure 6.7: (a): Throughput Comparison on Selected Realistic Benchmarks,
(b): Throughput Benefit (max(AS0,AS1)/max(S0,S1)) for Different Traffic Ratios (# of messages ending in st-0,1/# of messages ending in st-2,3) in All Realistic Benchmarks

finished, we check whether the messages are all properly transferred. Then, the worst-case latency and throughput are recorded.

6.5. Evaluation

6.5.1. Realistic Workloads

Figure 6.7 illustrates the throughput advantage of asymmetric BFTs for realistic, Graph Analytics workloads from [67]. The datasets are undirected graphs, and each graph edge counts for two packets, swapping the sender and the receiver. The total number of packets for each benchmark ranges from 16K to 485K. We use `metis` [54] to cluster the graph into 256 parts using a recursive bisection scheme with the objective to minimize the edge cuts. We check the number of messages whose destination is in the dense subtree (st-0,1) and the number of messages whose destination is in the sparse subtree (st-2,3). If the number of messages traveling to the sparse subtree is larger, we simply reverse the placement so that node 0 becomes node 255, node 1 becomes node 254, and so on. Within the dense subtree, if the number of messages traveling to st-1 is larger than st-0, we try reversed placement as well to benefit from AS1 which offers a large bandwidth in st-0. The number of total datasets is 60 including 32 reversed node placement versions. Thus, the number of unique datasets is 28. The *injection rate*, the rate that each PE sends messages to the network, is set to 100%, which means that all the PEs attempt to send valid packets every cycle. The throughput (pkt/cycle/PE) is the average packet delivery rate computed as the total number of packets divided by the total elapsed cycles, divided by the number of PEs.

Figure 6.7 (a) shows that in realistic workloads, the asymmetric BFTs can achieve up to 32% higher throughput than the symmetric BFTs. Not all real-world applications exhibit asymmetric traffic, and for applications where the loads are relatively balanced, it is natural that symmetric BFTs are the better options. But even after the graph is bi-partitioned, one partition's communication can be heavier than another partition's like the benchmarks in Figure 6.7 (a), and in such cases, asymmetric BFTs have an advantage over symmetric BFTs. In Figure 6.7 (a), we selectively include benchmarks that exhibit at least 10% improvement in throughput with asymmetric BFTs (`musae-twt-DE`, `deezer-europe`, `gemsec-fb-new-sites`, `CA-CondMat`, `CA-HepTh`, `gemsec-fb-gov`).

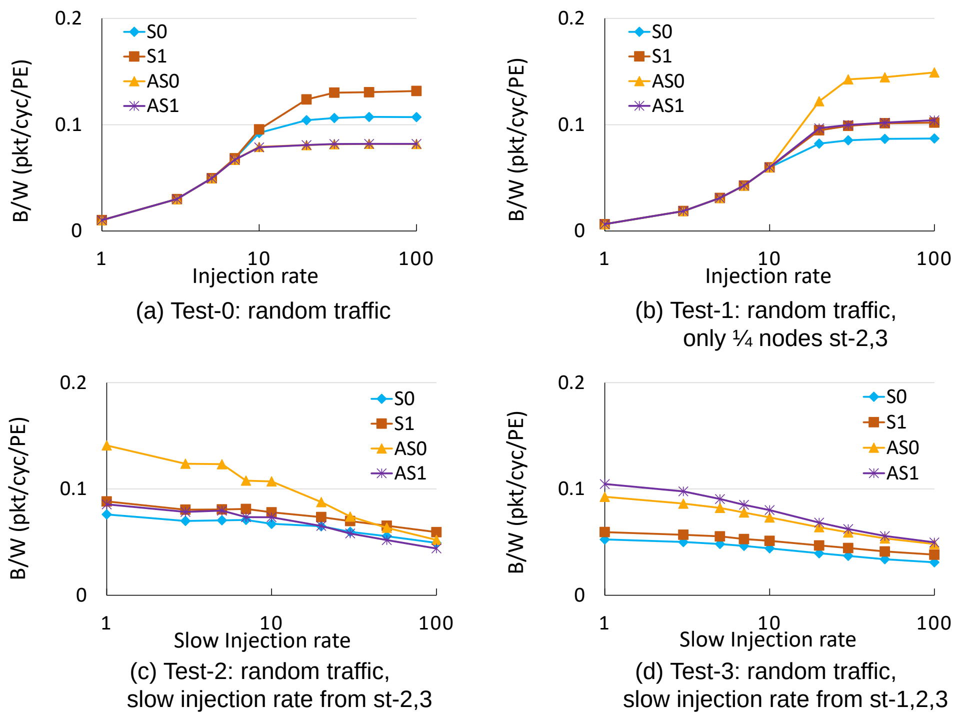
We also evaluate the correlation (Figure 6.7 (b)) between the throughput benefit of asymmetric BFTs and the traffic ratio of the dense subtrees (st-0,1) and the sparse subtrees (st-2,3) for all benchmarks. Throughput benefit is computed as the maximum throughput achieved by AS0 and AS1 divided by the maximum throughput achieved by S0 and S1. The traffic ratio is the number of messages delivered to PEs in dense subtrees divided the number of messages delivered to PEs in sparse subtrees. The color schemes of the markers correspond to the BFT type that performs the best for the graph workload. Therefore, markers whose throughput benefits are less than 1 are colored red (S0 or S1) and markers whose throughput benefits are greater than 1 are colored blue (AS0 or AS1). `musae-twt-DE` that shows 32% better throughput in asymmetric BFTs has a traffic ratio of 4.92. This is consistent with our expectation that asymmetric BFTs are better when there exists more traffic in the dense subtrees (traffic ratio > 2.2) and there exists less traffic in the sparse subtree. However, the larger traffic ratio does not guarantee the better throughput advantage, and the traffic ratio greater than 1 does not guarantee the throughput benefit. This is because the current traffic ratio formulation does not take into account local communication. For example, if the number of messages delivered to PEs in the dense subtree is large, then the traffic ratio is large in our definition. However, if these messages travel close to neighboring PEs, asymmetric BFTs are not necessarily better than symmetric BFTs.

6.5.2. Random Traffic

To better characterize the underlying phenomena, we also evaluate asymmetric BFTs with four different synthetic traffic patterns:

- **Test-0**: each PE randomly sends to another.
- **Test-1**: all PEs in st-0,1 are active and only 1/4 of PEs in st-2,3 are active. Each PE randomly sends to another.
- **Test-2**: each PE in st-0,1 randomly sends to another PE in st-0,1. The PEs in st-2,3 randomly send to PEs in st-0,1 with slow injection rate.
- **Test-3**: each PE in st-0 randomly sends to another PE in st-0. The PEs in st-1,2,3 randomly send to PEs in st-0 with slow injection rate.

Figure 6.8 illustrates the throughput performances for symmetric BFTs and asymmetric BFTs on different traffic patterns. The number of messages per PE is set to 1024. We see that symmetric BFTs are better in Test-0 because the lower bandwidth in s-2,3 causes congestion in asymmetric BFTs. Test-1 and Test-2 are the simulated scenarios that st-0,1 are more active than st-2,3. As expected, AS0 that has more bandwidth in st-0,1 exhibits up to 46% (Test-1) and 60% (Test-2) improved throughput than S0 and S1. Test-3 is the simulated scenario that st-0 is more active than the others. As expected, AS1 that has more bandwidth in st-0 exhibits up to 76% improved throughput than S0 and S1. In Test-2 and Test-3, when the slow injection rate from the sparse subtrees is low enough, asymmetric BFTs perform better. But as the slow injection rate increases, the



(a) Test-0: random traffic
(b) Test-1: random traffic, only ¼ nodes st-2,3
(c) Test-2: random traffic, slow injection rate from st-2,3
(d) Test-3: random traffic, slow injection rate from st-1,2,3

Figure 6.8: Throughput Comparison on Different Random Traffic Patterns

benefit in throughput decreases because the sparse subtrees in asymmetric BFT become congested.

6.5.3. t-random Switch

To characterize the benefits of using switches inside the converging switch as described in Section 6.3, AS0_S and AS1_S in Figure 6.9 refer to corresponding asymmetric BFTs with a strawman implementation of a converging switch that consists of only standard t switches. In Test-1, $t - random$ switches significantly improve the worst-case latency (orders of magnitude) and the throughput (up to 65%), relieving the congestion in the leftmost switches and the rightmost switches.

6.6. Discussions

6.6.1. More Design-Space for Soft NoC

There is no single best soft NoC for all applications, but there are soft NoCs with different compositions that can perform better for specific applications. We expand the design-space of soft NoC so that users can tailor the NoC to their applications, more fully exploiting FPGA's reconfigurability. Unlike previous literature that recommends a specific type of BFT based on the LUT budget on the

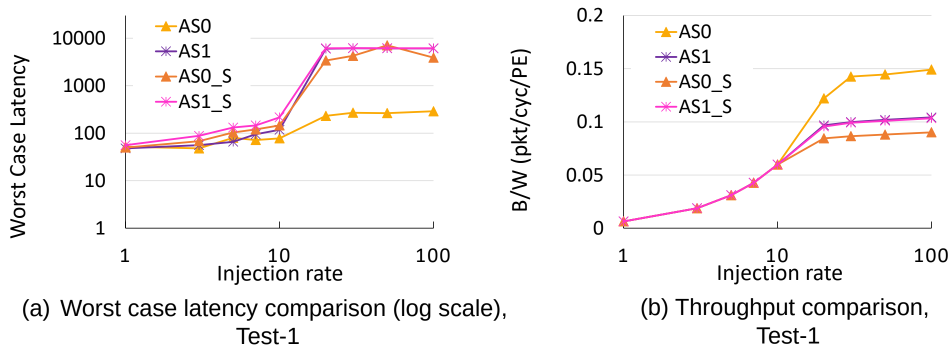


Figure 6.9: Benefit of $t - random$ switches in Converging Switch for Test-1

FPGA [48, 71] independent of the application, we propose asymmetric BFT architectures that could exhibit better throughput than symmetric BFT for applications where the loads are unbalanced.

6.6.2. Limitations

To take full advantage of asymmetric BFT, the users need to have some understanding of the application's traffic pattern because they need to configure the asymmetric BFT to provide more bandwidth where needed. We believe such a constraint is acceptable as soft NoC can always be reconfigured with other logic on FPGA. For example, when the NoC congestion is detected with the symmetric BFT in the steady traffic pattern, the user can adopt the asymmetric BFT overlay with proper subtrees and reconfigure the device. Because the resource utilization of symmetric BFT and the resource utilization of asymmetric BFT are similar, users can select the NoC overlay, leaving PEs untouched.

Ideally, based on the simulated or runtime traffic, the appropriate asymmetric BFTs can be generated within the FPGA resource budget. In this chapter ([80]), however, the configurations for the asymmetric BFTs are the user input.

In Section 6.4, we have shown that $t - random$ switches should not limit the overall operating frequency of the NoC because they are faster than π switches. However, in reality, achieving competitive frequency as the number of PEs increases is not trivial [71]. Because subtrees in an asymmetric BFT have different resource usage and routing complexity, floorplanning is expected to be more challenging. More studies need to be done to further investigate the operating frequency of asymmetric BFT.

6.6.3. Integration with our NoC-based System

In Chapter 5's NoC-based system [77], we use only symmetric BFT. In section 6.4 and section 6.5, the size of asymmetric BFTs used to demonstrate the benefit is 256. We observe the asymmetric BFT's benefit when the number of PEs is large in the BFT NoC and when the traffic is highly unbalanced. However, the number of PEs used in Chapter 5's NoC-based system is only 24 and the Rent parameter p is 0.67. For benchmarks used in Chapter 5, we have seen the cases where the small payload size (32 bits) of the NoC limits the application performance. Nevertheless, we believe NoC congestion is not a major issue that limits the performance because of the small size of the NoC and reasonably good Rent parameter $p = 0.67$. Thus, we use a symmetric BFT in our fast separate compilation framework and leave it as a future work to provide more NoC-based overlays with different NoC options. Because our asymmetric BFT uses similar resources to symmetric BFT, we can potentially have a PR region for the NoC and use the appropriate partial bitstream for the NoC.

6.7. Conclusions

To ensure continuity from the NoC-based system to the monolithic system in our incremental refinement, we explore the design-space of BFT NoC to support highly unbalanced workloads. We demonstrate that given the same LUT budget, in realistic workloads and different random traffic patterns, asymmetric BFTs with converging switches built with $t - random$ switches can achieve up to 32% and 76% more throughput than symmetric BFTs.

CHAPTER 7

DISCUSSIONS

7.1. Scalability

In Chapter 3 and Chapter 5, we evaluate our fast compilation framework with AMD ZCU102 with ZU9EG FPGA that has about 270K LUTs. The idea can extend to larger FPGA devices that have over a million LUTs. With a larger device, our separate compilation framework is expected to achieve a better compilation speedup because (1) the size of the PR pages can stay small and (2) the monolithic compilation is expected to scale superlinearly as stated in Section 2.4. The size of single-sized PR pages can stay at 7K–8K LUTs,[6] but the number of PR pages increases from 22 (Chapter 3, Chapter 5) to about 100. Therefore, assuming the monolithic compilation scales linearly, we will already achieve 4–5 times more parallelism in the degree of separate compilation, leading to a better compilation speedup. If the monolithic compilation suffers from poor scalability, the compilation speedup will only increase.

However, for a larger FPGA device, compiling for the same, single-sized PR page should take longer even on the same workstation because loading device information and reading design checkpoints take longer for larger devices. Thus, we do not expect to have 2–3 minutes of incremental compilation as we have seen in Chapter 3 and Chapter 5 but expect to have a longer incremental compilation. When compiling for a page, the entire device does not need to be loaded into the memory but the related segment of the device can be loaded. Nevertheless, today's vendor tool still loads the entire device information, potentially limiting the benefit of our separate compilation approach based on PR since we need to load the static design and link the netlist on the PR region. The overhead of loading a large device information or a design checkpoint in Vivado can be mitigated with the use of the open-source FPGA toolchain [1, 84], as the feasibility of using such toolchain for separate FPGA compilation is introduced in [99].

[6]The number can change depending on the FPGA family. For UltraScale+ FPGA, 7–8K is about the right number with the reasons explained in Section 3.5.1. If the FPGA has more number of CLBs in one clock region height, the number should increase.

Creating a static design with Hierarchical PR pages requires multiple sequential implementations. Because each top-level implementation takes more time and the number of implementations grows with more PR pages, the overall process should become more time-consuming. Nevertheless, this is a one-time cost and can be hidden from application developers.

7.2. Application Decomposition

As mentioned in Section 3.4 and Section 3.5.3, because our NoC-based system assumes operators connected with dataflow streams, we modify the provided code from Rosetta Benchmark Suite [109] to create an array of operators that have inputs and outputs of HLS Stream (`hls::stream`). The original applications in the Rosetta Benchmark Suite are not written in a dataflow graph, showing that our framework can accommodate general HLS designs with some code refactoring. For applications like Rendering, Optical Flow [109] and CNN applications from FINN [92], the decomposition is straightforward. For example, FINN generates HLS source code for a streaming architecture for neural network, requiring minimal additional effort to use our framework. However, for some applications that share storage or consist of a single compute engine, application decomposition is not intuitive. If the application developers explicitly design the applications in a way that the inputs and outputs are in HLS Stream, mapping to our framework would not require any refactoring, and developers can easily accelerate FPGA compilation with our framework.

In this dissertation, one operator is mapped on one PR region, and the granularity of the operators is determined by the user. A separate HLS source code indicates one operator, and our framework maps the top function in the source code to the appropriate PR page based on the post-synthesis resource utilization and the application graph. The approach that allocates one operator per PR region could potentially result in internal fragmentation when the sizes of operators are small. For instance, if an application has 100 operators whose sizes are 100 LUTs, the application cannot be mapped on the target device with our framework although the entire application is even smaller than one double-sized page. Each operator will be mapped on a single page that has 7K LUTs, and 6,900 LUTs are left not used. In the case of many small operators, we can improve our framework by merging multiple operators and map on a single page. The same bottleneck identification based on FIFO counters can be used. The current framework in [77] has FIFO counters per a PR page. However, in the case when multiple operators are mapped on one page, we can instantiate similar FIFO counters for all the operators and retrieve them back to the host to profile in the software.

7.3. Hard NoC

While the majority of FPGA devices in today's market do not have an embedded NoC, Versal from AMD [35] and Speedster7t from Achronix [3] have a hard NoC along with the programmable logic. Recent academic literatures [19, 18] also explore novel reconfigurable accelerators with a hard NoC. A hard NoC can support high-bandwidth data movement between memory and programmable fabric so that compute kernels can focus on computation instead of communication. A hard NoC also promotes a modular design methodology that aligns with our divide-and-conquer philosophy. Our fast separate compilation framework should be portable with any FPGA devices that have or do not have a hard NoC because we are using a soft NoC. Nevertheless, it is natural to expand our idea to modern FPGAs with a hard NoC since hard NoCs provide higher bandwidth and do not consume FPGA resources. SPADES [75], for example, creates 12 "sockets" on the endpoint of the Versal NoC, an equivalent concept to our pages, and it uses RapidWright [63] for stitching and replication as done in [91]. Although SPADES uses RapidWright in design manipulation, a hard NoC and PR can create a natural synergy to enhance modularity in the design and to accelerate FPGA compilation, just like we have shown in [79, 99, 78, 77] with a soft NoC and PR to support arbitrary communication between operators and to avoid stitching. A downside of using a hard NoC is that the degree of parallelism in separate compilation is limited by the number of the hard NoC's endpoints. For instance, a large FPGA device with a million LUTs can accommodate over 100 single-sized pages with 8K LUTs with a soft NoC. Nevertheless, with a hard NoC, the number of pages is bounded by the number of endpoints of the hard NoC, possibly resulting in lower compilation speedup. An easy solution is to have a soft switch logic inside a hard NoC's pages. In this hybrid approach, we can first create large pages for all the endpoints. If there are 30 endpoints in the hard NoC, the size of each page is about 33K LUTs in the device with a million LUTs. Then, as done in our Hierarchical PR pages (Chapter 3), we can subdivide each page into multiple smaller pages to provide both flexibility and finer-grained separate compilation.

In Chapter 5's fast incremental refinement strategy, the reason why we migrate to the monolithic system is to remove the area overhead and limited bandwidth of the soft NoC. With a hard NoC, no migration is expected. No programmable resource is occupied by the NoC, and when even the high bandwidth from a hard NoC is not sufficient, we can merge operators as done in Chapter 3 and Chapter 5.

7.4. Vivado PR

Modern PR technology from the vendor [14] has a few limitations to fully support our vision. We use PR to separately compile each page, and we do not dynamically program a part of the device on the fly. Thus, if PR's dynamic nature is related to the issues that will be discussed in this section, we need PR technology without its dynamic reconfiguration aspect.

7.4.1. Identical PR Regions

Both AMD PR and Altera PR allow static routing over the PR regions. As stated in Section 3.5.1 and Section 4.3, static routing over PR pages makes pages more heterogeneous by blocking logical resources and stealing routing resources. In this work, we mitigate the static routing by using `CONTAIN_ROUTING true` for the static pblock. By manually setting the Partition Pins to the same relative locations for PR regions, it should be possible to create clean PR regions with no static routing and identical IOs.

7.4.2. Partial Bitstream Relocation

While academic works [58, 82, 36, 72] support module relocation in the bitstream level, Vivado PR does not officially support partial bitstream relocation. This means that even if we create the same reconfigurable pblocks that have the same amount of static routing over, the same amount of blocked resources, the same resource composition, and the same IOs, there is no vendor tool support to replicate the partial bitstream from one PR region to another PR region. First, this limitation prevents us from taking full advantage of the software principle of reusing pre-compiled objects with simple linking. Even when an operator's source code remains unchanged, if the PR page for an operator changes, the operator must be re-placed and re-routed for a new PR page. One scenario is that if one operator gets to use a larger page, then the adjacent operator needs to be evicted to a different PR page. Second, when testing a different page mapping, we want to quickly swap partial bitstreams for different pages instead of recompiling operators for different pages every time, but this is not possible at the moment. For a larger device with PR pages over 100, the impact of the page assignment on the application performance increases, and relocating modules at the bitstream level should be a useful feature. While it is theoretically possible to compile an operator for all 100 PR regions in the cloud (if there are 10 operators, this application requires 10×100 compilations in the cloud), using relocatable partial bitstreams offers a much more elegant solution.

7.4.3. Stacking Multiple PR Regions within a Clock Region

The last issue with AMD Vivado's PR technology [14] is that multiple reconfigurable pblocks cannot be stacked within a single clock region as stated in Section 3.5.1. This limitation may be related to AMD FPGA's configuration frame and the dynamic nature of PR which programs one reconfigurable partition while other parts of the design are operating. In our usage, we use PR just to separately compile each operator, and this limitation may be unnecessary. Our goal is to create finer-grained pages which lead to higher compilation speedup in our strategy. A design rule check option, which we should turn on in dynamic PR but can turn off in the separate compilation like our usage, could be useful. In Altera's PR technology [5], multiple reconfigurable modules can exist in single clock region, suggesting a potential for finer-grained PR pages (even smaller than 7–8K LUTs with a reasonable aspect ratio).

7.5. FPGA Architecture

The obvious prerequisite for bitstream relocatability mentioned in the previous section is regular resource distribution in the FPGA. Modern FPGA consists of multiple columns of different resource types, but as mentioned in Section 3.5.1, the distribution is not regular. If bitstream relocation is supported by the vendor tool, it is possible to relocate the bitstream to the same-sized PR page in the same column. To relocate bitstreams horizontally as well, we need an FPGA with a grid of regular regions. Authors in [91] also argue that irregularities in AMD's UltraScale+ architecture lead to resource wastage in their copy-and-paste approach to reduce FPGA compilation. Recently, FPGA vendors promote a regular grid of programmable resources [42, 25, 35], opening up opportunities for a divide-and-conquer strategy like ours. Creating a high-performance static design in a multi-SLR device is challenging, and the difficulties related to the device architecture will be explained in Section 7.6.1.

7.6. Divide-and-Conquer to Achieve a Better Maximum Frequency

7.6.1. High-performance Static Design in PR

Related works [41, 40, 75], which utilize RapidWright instead of PR, demonstrate a better maximum operating frequency with a divide-and-conquer strategy. However, in this work, we mainly focus on accelerating compilation time although a modular design methodology with a NoC is a widely accepted design approach to achieve a better frequency in the routing-intensive design. AMD DFX user guide [14] implies that PR can be used to close timing with a divide-and-conquer strategy, but, to the best of our knowledge, whether the achieved frequency is better than the frequency achieved by the monolithic implementation is not thoroughly addressed. Whether we use a soft NoC [99, 98, 78, 77] or directly connect operators as done in [97, 100], the challenge with the divide-and-conquer using Vivado PR technology lies in *creating a high-performance static design in the first place.*

There are some engineering considerations when creating a high-performance static design in PR. First, if we use large reconfigurable modules as placeholders, the overall design size increases, and it is difficult to achieve high clock frequency for the static design. If we use almost empty placeholders to create a static design, then a lot of static design routes over the PR regions, so we may need to constrain the routing of the static design with `CONTAIN_ROUTING` property as done in Section 4.3.2. The next aspect to consider is floorplanning. In AMD ZCU102 with ZU9EG FPGA used in Chapter 3 and Chapter 5, the Processing System is located in the lower left, and when floorplanning PR pages, it is essential to ensure that the NoC and other static elements can be placed adjacent to the Processing System. The problem is exacerbated in AMD Alveo devices [6] that have multi-SLRs and have limited Laguna channels between different SLRs. Alveo devices also have a PCIe on one side and High Bandwidth Memory (HBM) on another side, imposing additional constraints for static design placement and routing.

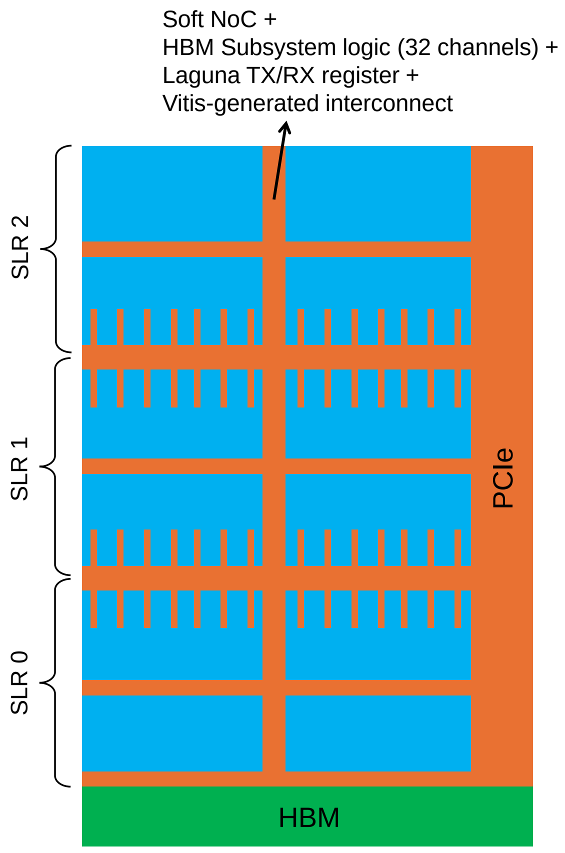

Soft NoC +
HBM Subsystem logic (32 channels) +
Laguna TX/RX register +
Vitis-generated interconnect
SLR 2
SLR 1
SLR 0
PCIe
HBM

Figure 7.1: A Conceptual View of High-performance Static Design in AMD U280 Device

The main problem with creating a high-performance static design is that the vendor tool may use the same optimization objective in the placement stage when creating a static design or compiling for an operator in a PR region. When compiling for an operator, the cells are generally placed in a "round shape". In contrast, when compiling for the static design, we want the design to be spread out across the device, and by using a constraint like `CONTAIN_ROUTING`, we want the static design to be placed and routed "densely", leaving as much logic and routing resources as possible for the PR Regions. Our works are compatible with Vitis Acceleration flow [12], and the auxiliary Vitis-generated logic (27K LUTs in ZCU102 DFX Platform) in addition to the soft NoC makes it more challenging to close timing for the static design. Figure 7.1 shows a conceptual view of high-performance static design (colored orange) in AMD U280 device [6]. In this case, the static logic in PR should consist of PCIe logic, a soft NoC, all 32 channels of HBM Subsystem, TX/RX registers used in SLR crossing, and Vitis-generated interconnect logic. We want this logic to be densely packed in a static pblock that has `CONTAIN_ROUTING true`. A systematic design methodology for setting up this static logic that runs at high clock frequency would be a valuable study for the designs that struggle to achieve a high operating clock frequency. For instance, [45] implements a Sparse matrix–vector multiplication application (SpMV) on AMD U280 device, aiming to maximize HBM bandwidth utilization. Despite using a modular and lean SpMV architecture, the authors achieve a SpMV kernel frequency of 310MHz, which still falls short for 450MHz required to fully utilize the HBM bandwidth available in the U280 device.[7] In this case, the static logic that runs at 450MHz, thereby fully utilizing HBM bandwidth, can decouple the implementation of communication and computation. A single SpMV kernel can then be separately compiled within a PR region, running at over 400MHz, as already demonstrated in [45].

To create a static design that is densely packed and capable of running at a high clock frequency, we can constrain the pipeline registers as done in Section 4.4.1 or guide the placement of the NoC and other elements with blocks. Because too much of floorplanning constraints often result

[7]A kernel frequency of 450MHz is required to fully utilize the HBM bandwidth in the U280, assuming a datawidth of 256 bits. If the datawidth is increased to 512 bits, the required kernel frequency is halved to 225MHz at the cost of using more FPGA resources for communication, yet making it a more relaxed target to achieve on an UltraScale+ device. For example, [90] achieves the full HBM bandwidth utilization with 512 bits $\times$ 225MHz in SpMV application.

in inferior clock frequency [8], we need to provide *appropriate* amount of constraints to Vivado. For the same reason, automatic generation of the NoC-based overlay for different devices could be nontrivial. While it is easy to automatically generate the overlay that runs at low clock frequencies like 200MHz, when we run the NoC at 400MHz and feed different clock frequencies ranging up to 400MHz, the NoC overlay generation requires some careful engineering for UltraScale+ devices. In summary, based on our experience, creating a high-performance static design requires meticulous engineering and poses a significant challenge in achieving a high maximum frequency with the commercial PR technology. Nevertheless, the systematic approach to generating a static design presents a promising area for further research.

7.6.2. Continuity between the NoC-based System and the Monolithic System

If we have a highly optimized NoC-based system which can run at a higher clock frequency than the monolithic design, then there is a discontinuity in our fast incremental refinement strategy (Section 5.3). Similar issue is mentioned in Section 5.7.3 when having an implementation directive per PR page in the NoC-based system could lead to a more performant design than having a single implementation directive for the entire design in the monolithic system. An assumption in our incremental strategy is that the NoC-based system explores design points that would have been explored by the monolithic flow. If the NoC-based system ends in the design point that is unrelated to monolithic flow, then we need to revisit our incremental refinement strategy.

7.7. Relationship with other Fast FPGA Compilation Studies

Our approach is different from the aforementioned approaches using pre-compiled macros to accelerate FPGA compilation (Section 2.5) because ours is not limited by the pre-compiled IPs stocked in the library. Arbitrary user designs can be compiled with our divide-and-conquer approach, and fast compilation comes from the smaller problem size. This means that the divide-and-conquer philosophy and other works that utilize pre-compiled macros can be integrated. For example, benchmarks used in DynaRapid are less than 10K LUTs in size, and as the design size increases, the stitching for the pre-routed circuits should become more challenging both in terms of runtime and quality. To maintain a sub-minute of C-to-bitstream compilation, the usage of macros on top of modular design methodology like ours could be a promising direction.

Similarly, fast FPGA placement and fast FPGA routing have been extensively studied [23, 88, 20, 10], and some approaches excel in a small problem but lack scalability. For these studies, explicitly dividing a large compilation problem into multiple smaller subproblems offers a direction to sustain competitive placement or routing time. The same idea is applied to the vendor tool's placement and routing engine. Vivado's implementation algorithms are not exposed to the users, and whether we compile for a small page or we compile for an entire chip, we may end up using a very similar algorithm. It would be an interesting feature to have a Vivado's implementation directive that performs well only in a small design. If scalability was a problem for such an algorithm to be adopted in the commercial tool, our separate compilation framework provides a good application space for the algorithm to be useful.

7.8. Future Work

The vision for future work integrates all the components mentioned in this chapter. First, we want to target a larger device with a hard NoC such as VCK190 board [11] featuring AMD Versal XCVC1902 which has 899,840 LUTs. As outlined in Section 7.3, we can create Hierarchical PR pages for each endpoint of the NoC. By adding soft switch logic to the endpoints of the hard NoC, we can create a "hybrid NoC" with an increased number of finer-grained pages. Given the hard NoC's 28 endpoints, each page should have 32K = 899,840/28 LUTs, just about the same size as the quad-sized PR page from Chapter 3 and Chapter 5. If we create two more hierarchies inside the page, the smallest, single-sized page will have about 8K LUTs.

On the application side, we can explore a high-level domain-specific framework like FINN [92] for a smooth integration with our philosophy. With minimal hardware expertise, the users should be able to design an application composed of operators connected through dataflow streams. The subsequent steps can be similar to our work. The framework can perform HLS and logic synthesis in parallel for operators, and based on the post-synthesis resource utilization, an appropriate size for the PR page is assigned. Then, each operator can generate its own partial bitstream.

As discussed in Section 7.2, we can group operators into clusters per PR page instead of a single operator on each page. In this thesis, “operators” refer to logical computational blocks, but in the future work, these could become more primitive and fine-grained. The challenge is that clusters must be defined before the logic synthesis because the NoC interface is synthesized along with the cluster. If the target applications are limited to a specific set of domain-specific applications, it may be easier to develop models to predict the resource utilization as done in the FINN project. Using these resource models, we can form clusters of operators, and each cluster, as an operator in this thesis, can be compiled in parallel.

We can continue to use a similar bottleneck identification based on FIFO counters from Section 5.4. Even with the clusters, we can keep the bottleneck identification at operator-level, with each operator expected to be a finer-grained computing block.

The conceptual view is illustrated in Figure 7.2. The idea is elaborated by assuming that we are using a hard NoC and PR, but the vision can be stretched using RapidWright if PR imposes engineering challenges like the ones discussed in Section 7.1 and Section 7.6.1.

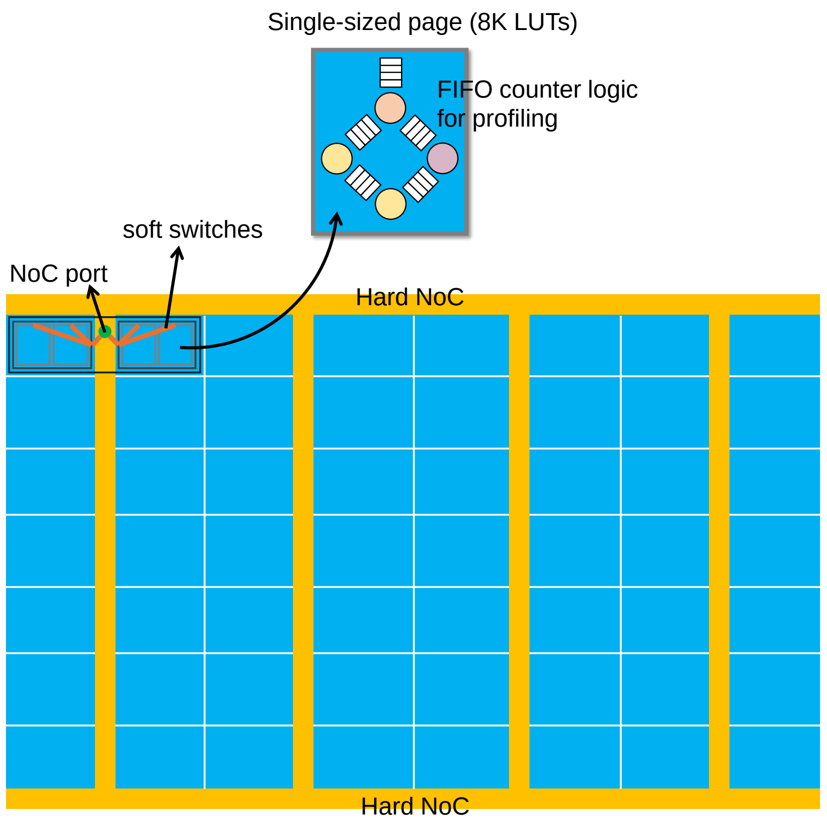


Figure 7.2: Future Direction with a Hard NoC and Fine-grained Operators
Note: This is a conceptual view. Previously mentioned limitations in the PR (e.g. PR regions can't be stacked in a clock region) could limit the realization of the idea.

CHAPTER 8

CONCLUSIONS

Software programmers start from a design that is barely optimized and incrementally refine the design. In software, the source codes for the initial design point are compiled in parallel and linked together to generate an executable file. Then, programmers profile the design, identify the bottleneck, and optimize the bottleneck function. When refined functions are integrated into the design, only the changed functions can be recompiled and linked with the object files that were already compiled. Therefore, even if the design size increases, the compilation time does not increase unless the users recompile the system from scratch. In FPGA design optimization, such incremental refinement is not realized. Inspired by the software development principle, in this thesis, we support software-like FPGA development. The first problem we challenge is the long monolithic FPGA compilation. To narrow the performance gap between FPGAs and ASICs, FPGA vendors have traditionally focused on delivering high-quality solutions through their compilation process rather than prioritizing compilation speed. As a result, vendor tools try to optimize an entire design, even if it means a long compilation time. What is worse is that when there is a fix in one component, the entire design needs to be recompiled, leading to another long compilation. To resolve this issue, we support parallel compilation using pre-implemented NoC and PR pages. Since the problem size is smaller, compilation is faster, explicitly utilizing more cores in today's multi-core workstations or compute servers. Because the compilation of each page is independent of each other, our divide-and-conquer strategy also supports incremental compilation where only the changed operator, not the entire design, can be recompiled.

A problem including our pioneering work and related work is that the size of pages is fixed. If the size of the pages is small, it is the user's responsibility to decompose a design into smaller operators. Additional effort to decompose a design into regularly sized small operators is against our intent to make software-like FPGA development where we want users to quickly test the design on FPGA just like they start from barely functional design in software. Furthermore, applications with unnaturally decomposed operators sometimes suffer from limited NoC bandwidth. On the other hand, if the size of the pages is large, the degree of parallelism in a separate compilation strategy is limited. Utilizing Hierarchical PR, we provide variable-sized PR pages. We support single-sized pages as done in the previous separate compilation framework, but now, the single-sized pages can be recombined to offer double-sized pages or quad-sized pages. This flexibility removes a burden for the users to have some idea of how much resources the operators would consume. Also, applications do not suffer from limited NoC bandwidth because of unnaturally decomposed operators since larger operators now can be mapped in larger, recombined pages. When tested with a realistic Rosetta HLS benchmark suite, variable-sized pages lead to up to 4.9× application latency improved compared to the fixed-sized pages by mitigating the limited NoC bandwidth issue. Our new framework with variable-sized pages still achieves 2.2–5.3× speedup in compilation time compared to the commercial tool.

Design optimization is an iterative process, and to utilize our framework's incremental compilation, we need a profiling capability to identify the bottleneck operator to refine. In contrast to software development where users enjoy rich profiling tools to analyze their design, there is less visibility on the inner state of the hardware design. Using FIFOs that are already embedded in dataflow applications, we increment stall counters and full counters under different conditions. Based on these FIFO counters, users or an automation script can identify the bottleneck operator. We can also determine whether the limited NoC bandwidth in our separate compilation framework is limiting the application performance. Altogether, we propose a fast incremental refinement strategy for FPGA design. The idea is to start from our fast NoC-based system to iterate initial design points as many as possible. Based on the FIFO counters extracted from the runtime execution on hardware, our script chooses the next design point that can improve the application's performance. When the design needs more resources or the design-space is all explored in the NoC-based system, we migrate to the monolithic system that directly connects operators without area and bandwidth overhead from the NoC infrastructure. The fast incremental refinement strategy is evaluated with Rosetta HLS benchmark suite and FINN framework, and our strategy reduces tuning time by 1.3–2.7× compared to the flow that uses only a monolithic system to iterate every design point.

Our solution exploits FPGA's reconfigurability, iteratively mapping different design points on hardware. FPGA compilation does not have to be a button that can finally be pushed only when the design is completely ready. Unlike ASICs, FPGA engineers can and should evaluate different design points as quickly as possible. Additionally, our modular approach enables the use of placement and routing algorithms that lack scalability because we compile only for pages, significantly smaller than the size of the entire device.

Still, challenges remain. In this thesis, we use Vivado's PR technology to accelerate Vivado's compilation time. However, using PR in non-traditional way has introduced challenges, including the impact of static design's size on compilation time, excessive static routing over PR regions, limitations in the size of reconfigurable pblocks within a clock region, partial bitstream relocatability, and difficulties in creating high-performance static logic. With the advent of new reconfigurable SoC architecture already on the market, we believe our approach can generate a broader impact with active vendor support for the toolchain or the adoption of an open-source toolchain.

APPENDIX A

FAST INCREMENTAL REFINEMENT DSE TRACES

Table A.1, Table A.2, Table A.3, Table A.4, Table A.5, Table A.6, and Table A.7 show the detailed DSE traces for our fast incremental refinement strategy in Section 5.7.4.

Table A.1: Fast Incremental Refinement DSE Trace for Rendering

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	240s	None	init	-	2.21ms	2.21ms	5	NoC
2	245s	`rast2_i1`	`PAR_RAST` = 2	lat.	1.56ms	1.56ms	4	NoC
3	275s	`zculling_i1`	`PAR_ZCUL` = 2	lat.	1.27ms	1.27ms	8	NoC
4	269s	`rast2_i1`	`PAR_RAST` = 4	lat.	1.13ms	1.13ms	9	NoC
5	319s	`zculling_i2`	`PAR_ZCUL` = 4	lat.	0.81ms	0.81ms	13	NoC
6	208s	`rast2_i1`	clk = 250MHz	lat.	0.81ms	0.81ms	4	NoC
7	210s	`zculling_i3`	clk = 250MHz	lat.	0.71ms	0.71ms	4	NoC
8	222s	`zculling_i3`	clk = 300MHz	lat.	-	0.71ms	4	NoC
9	229s	`zculling_i3`	clk = 350MHz	lat.	0.64ms	0.64ms	4	NoC
10	211s	`rast2_i1`	clk = 300MHz	lat.	0.59ms	0.59ms	4	NoC
11	213s	`rast2_i1`	clk = 350MHz	lat.	0.58ms	0.58ms	4	NoC
12	214s	`rast2_i1`	clk = 400MHz	lat.	0.57ms	0.57ms	4	NoC
13	232s	`zculling_i3`	clk = 400MHz	lat.	-	0.57ms	4	NoC
14	795s	Prev cofing.*	Prev cofing.*	lat.	-	0.57ms	1	Mono
15	876s	`zculling_i3`	clk = 400MHz	lat.	-	0.57ms	1	Mono

Prev cofing.*: Because the previous config resulted in implementation failure, use the most recently successful design point.

Table A.2: Fast Incremental Refinement DSE Trace for Digit Recognition

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	298s	None	init	-	19.16m	19.16ms*	10	NoC
2	333s	None	`K_CONST` = 2	acc.	19.45m	19.45ms*	10	NoC
3	321s	None	`K_CONST` = 3	acc.	37.58m	37.58ms*	10	NoC
4	313s	None	`K_CONST` = 4	acc.	55.74m	55.74ms	10	NoC
5	335s	`update_knn_i1`	`PAR_FACTOR` = 20	lat.	28.69m	28.69ms	10	NoC
6	372s	`update_knn_i1`	`PAR_FACTOR` = 30	lat.	19.87m	19.87ms	10	NoC
7	357s	`update_knn_i2`	`PAR_FACTOR` = 40	lat.	15.44m	15.44ms	10	NoC
8	385s	`update_knn_i2`	`PAR_FACTOR` = 50	lat.	12.81m	12.81ms	10	NoC
9	425s	`update_knn_i2`	`PAR_FACTOR` = 60	lat.	11.09m	11.09ms	10	NoC
10	423s	`update_knn_i2`	`PAR_FACTOR` = 80	lat.	8.98ms	8.98ms	10	NoC
11	398s	`update_knn_i2`	`PAR_FACTOR` = 90	lat.	8.31ms	8.31ms	10	NoC
12	406s	`update_knn_i2`	`PAR_FACTOR` = 100	lat.	7.79ms	7.79ms	10	NoC
13	409s	`update_knn_i2`	`PAR_FACTOR` = 120	lat.	7.04ms	7.04ms	10	NoC
14	118s	`update_knn_i2`	`PAR_FACTOR` = 150	lat.	-	7.04ms	10	NoC
15	1224s	Prev cofing.**	Prev cofing.**	lat.	6.35ms	6.35ms	1	Mono
16	1437s	`update_knn_i2`	`PAR_FACTOR` = 180	lat.	5.99ms	5.99ms	1	Mono
17	1477s	`update_knn_i2`	`PAR_FACTOR` = 200	lat.	5.84ms	5.84ms	1	Mono
18	627s	`update_knn_i2`	`PAR_FACTOR` = 240	lat.	-	5.84ms	1	Mono
19	1672s	`update_knn_i2`	clk = 250MHz	lat.	4.75ms	4.75ms	1	Mono
20	1671s	`update_knn_i2`	clk = 300MHz	lat.	5.12ms	4.75ms	1	Mono
21	1734s	`update_knn_i2`	clk = 350MHz	lat.	4.40ms	4.40ms	1	Mono
22	2234s	`update_knn_i2`	clk = 400MHz	lat.	4.76ms	4.40ms	1	Mono

*: These designs have not achieved the target accuracy yet, so the best latency increases in the next design point.
Prev cofing.**: The flow migrates to the monolithic system because of lack of resources. So, try `PAR_FACTOR` = 150 in the monolithic system.

Table A.3: Fast Incremental Refinement DSE Trace for Optical Flow

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	287s	None	init	-	13.64ms	13.64ms*	7	NoC
2	294s	None	`OUTER_WIDTH` = 32	acc.	13.64ms	13.64ms*	4	NoC
3	335s	None	`OUTER_WIDTH` = 48	acc.	20.67ms	20.67ms	4	NoC
4	252s	NoC bottleneck	Merging operators	lat.	20.25ms	20.25ms	1	NoC
5	378s	NoC bottleneck	Merging operators	lat.	18.38ms	18.38ms	1	NoC
6	387s	NoC bottleneck	Merging operators	lat.	13.64ms	13.64ms	1	NoC
7	377s	`flow_calc`	`PAR_FACTOR` = 2	lat.	6.95ms	6.95ms	1	NoC
8	426s	`flow_calc`	clk = 250MHz	lat.	5.60ms	5.60ms	1	NoC
9	584s	`flow_calc`	clk = 300MHz	lat.	-	5.60ms	1	NoC
10	616s	`flow_calc`	clk = 350MHz	lat.	4.02ms	4.02ms	1	NoC
11	607s	`flow_calc`	clk = 400MHz	lat.	-	4.02ms	1	NoC
12	777s	Prev cofing.**	Prev cofing.**	lat.	4.01ms	4.01ms	1	Mono
13	759s	`flow_calc`	clk = 400MHz	lat.	3.55ms	3.55ms	1	Mono

*: These designs have not achieved the target accuracy yet, so the best latency increases in the next design point.
Prev cofing.**: Because the previous config resulted in implementation failure, use the most recently successful design point.

Table A.4: Fast Incremental Refinement DSE Trace for Optical Flow[‡]

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	282s	None	init	-	13.66ms	13.66ms	7	NoC
2	299s	`t_w_y_i1`	`PAR_FACTOR` = 2	lat.	6.92ms	6.92ms	9	NoC
3	168s	`t_w_y_i2`	clk = 250MHz	lat.	5.57ms	5.57ms	2	NoC
4	177s	`t_w_y_i2`	clk = 300MHz	lat.	5.47ms	5.47ms	2	NoC
5	219s	NoC bottleneck	`flow_calc`, # NoC interface = 2	lat.	4.74ms	4.74ms	1	NoC
6	177s	NoC bottleneck	`outer_prod`, # NoC interface = 2	lat.	4.71ms	4.71ms	1	NoC
7	160s	`t_w_y_i1`	clk = 350MHz	lat.	4.02ms	4.02ms	2	NoC
8	174s	`t_w_y_i1`	clk = 400MHz	lat.	3.84ms	3.84ms	2	NoC
9	247s	NoC bottleneck	`g_xyz_calc`, `g_w_y`, # NoC interface = 2	lat.	4.42ms	3.84ms	2	NoC
10	665s	Prev cofing.	Prev cofing.	lat.	3.54ms	3.54ms	1	Mono

Optical Flow[‡]: Optical Flow with a lower accuracy target

Table A.5: Fast Incremental Refinement DSE Trace for CNN-1

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	444s	None	init	-	-	-	15	NoC
2	216s	None	Larger PR page[*]	-	33.64ms	33.64ms	13	NoC
3	259s	`layer_3_1`	`PE` = 16	lat.	32.54ms	32.54ms	1	NoC
4	195s	`layer_0_0`	clk = 250MHz	lat.	30.67ms	30.67ms	1	NoC
5	176s	`layer_4_1`	`PE` = 2	lat.	27.20ms	27.20ms	2	NoC
6	214s	`layer_last_1`	`PE` = 2	lat.	26.79ms	26.79ms	2	NoC
7	168s	`layer_last_0`	`PE` = 2	lat.	26.20ms	26.20ms	2	NoC
8	206s	`layer_0_1`	clk = 250MHz	lat.	26.05ms	26.05ms	1	NoC
9	214s	`layer_0_0`	clk = 300MHz	lat.	24.25ms	24.25ms	1	NoC
10	258s	`layer_3_0`	`SIMD` = 16	lat.	23.15ms	23.15ms	2	NoC
11	178s	`layer_1_0`	clk = 250MHz	lat.	22.84ms	22.84ms	1	NoC
12	308s	`layer_1_1`	clk = 250MHz	lat.	21.73ms	21.73ms	1	NoC
13	212s	`layer_0_0`	clk = 350MHz	lat.	21.14ms	21.14ms	1	NoC
14	207s	`layer_0_1`	clk = 300MHz	lat.	21.14ms	21.14ms	1	NoC
15	189s	`layer_4_0`	`SIMD` = 2	lat.	18.72ms	18.72ms	2	NoC
16	240s	`layer_2_1`	`PE` = 16	lat.	18.59ms	18.59ms	2	NoC
17	197s	`layer_0_0`	clk = 400MHz	lat.	18.57ms	18.57ms	1	NoC
18	180s	`layer_1_0`	clk = 300MHz	lat.	18.33ms	18.33ms	1	NoC
19	309s	`layer_1_1`	clk = 300MHz	lat.	17.54ms	17.54ms	1	NoC
20	216s	`layer_0_1`	clk = 350MHz	lat.	17.40ms	17.40ms	1	NoC
21	248s	`layer_2_0`	`SIMD` = 16	lat.	16.63ms	16.63ms	2	NoC
22	722s	Prev config.	Prev config.	lat.	16.63ms	16.63ms	1	Mono

[*]: One or more operator fails in implementation. Assign larger PR pages for failed operators.

Table A.6: Fast Incremental Refinement DSE Trace for CNN-2

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	718s	None	init	-	-	-	15	NoC
2	218s	None	Larger PR page*	-	33.64ms	33.64ms	12	NoC
3	395s	`layer_3_1`	`PE` = 16	lat.	32.53ms	32.53ms	1	NoC
4	191s	`layer_0_0`	clk = 250MHz	lat.	30.67ms	30.67ms	1	NoC
5	201s	`layer_4_1`	`PE` = 2	lat.	27.19ms	27.19ms	2	NoC
6	234s	`layer_last_1`	`PE` = 2	lat.	26.83ms	26.83ms	2	NoC
7	194s	`layer_last_0`	`PE` = 2	lat.	26.20ms	26.20ms	2	NoC
8	226s	`layer_0_1`	clk = 250MHz	lat.	26.07ms	26.07ms	1	NoC
9	208s	`layer_0_0`	clk = 300MHz	lat.	24.25ms	24.25ms	1	NoC
10	393s	`layer_3_0`	`SIMD` = 16	lat.	23.15ms	23.15ms	2	NoC
11	162s	`layer_1_0`	clk = 250MHz	lat.	22.95ms	22.95ms	1	NoC
12	577s	NoC bottleneck	Merging operators	lat.	21.73ms	21.73ms	1	NoC
13	198s	`layer_0_0`	clk = 350MHz	lat.	21.12ms	21.12ms	1	NoC
14	239s	`layer_0_1`	clk = 300MHz	lat.	21.11ms	21.11ms	1	NoC
15	208s	`layer_4_0`	`SIMD` = 2	lat.	18.72ms	18.72ms	2	NoC
16	563s	`layer_2_1`	`PE` = 16	lat.	18.58ms	18.58ms	2	NoC
17	207s	`layer_0_0`	clk = 400MHz	lat.	18.55ms	18.55ms	1	NoC
18	574s	`layer_1_1`	clk = 300MHz	lat.	17.54ms	17.54ms	1	NoC
19	236s	`layer_0_1`	clk = 350MHz	lat.	17.40ms	17.40ms	1	NoC
20	706s	`layer_2_0`	`SIMD` = 16	lat.	16.66ms	16.66ms	13	NoC
21	921s	Prev config.	Prev config.	lat.	16.62ms	16.62ms	1	Mono

*: One or more operator fails in implementation. Assign larger PR pages for failed operators.

Table A.7: Fast Incremental Refinement DSE Trace for CNN-3

Iteration Count	Compile Time	Bottleneck	Design Point	Metric	Latency	Best Latency	# parallel runs	Flow
1	851s	None	init	-	33.66ms	33.66ms	15	NoC
2	582s	`layer_3_1`	`PE` = 16	lat.	32.53ms	32.53ms	14	NoC
3	210s	`layer_0_0`	clk = 250MHz	lat.	30.67ms	30.67ms	1	NoC
4	184s	`layer_4_1`	`PE` = 2	lat.	27.20ms	27.20ms	2	NoC
5	212s	`layer_last_1`	`PE` = 2	lat.	26.80ms	26.80ms	2	NoC
6	171s	`layer_last_0`	`PE` = 2	lat.	26.19ms	26.19ms	2	NoC
7	217s	`layer_0_1`	clk = 250MHz	lat.	26.04ms	26.04ms	1	NoC
8	228s	`layer_0_0`	clk = 300MHz	lat.	24.25ms	24.25ms	1	NoC
9	452s	`layer_3_0`	`SIMD` = 16	lat.	23.15ms	23.15ms	2	NoC
10	161s	`layer_1_0`	clk = 250MHz	lat.	22.93ms	22.93ms	1	NoC
11	773s	NoC bottleneck	Merging operators	lat.	21.74ms	21.74ms	1	NoC
12	215s	`layer_0_0`	clk = 350MHz	lat.	21.11ms	21.11ms	1	NoC
13	247s	`layer_0_1`	clk = 300MHz	lat.	21.11ms	21.11ms	1	NoC
14	208s	`layer_4_0`	`SIMD` = 2	lat.	18.71ms	18.71ms	2	NoC
15	609s	`layer_2_1`	`PE` = 16	lat.	18.58ms	18.58ms	14	NoC
16	210s	`layer_0_0`	clk = 400MHz	lat.	18.55ms	18.55ms	1	NoC
17	831s	`layer_1_1`	clk = 300MHz	lat.	17.54ms	17.54ms	1	NoC
18	238s	`layer_0_1`	clk = 350MHz	lat.	17.40ms	17.40ms	1	NoC
19	443s	`layer_2_0`	`PE` = 16	lat.	16.66ms	16.66ms	2	NoC
20	1091s	Prev config.	Prev config.	lat.	16.63ms	16.63ms	1	Mono

BIBLIOGRAPHY

[1] F4pga documentation. https://f4pga.readthedocs.io/en/latest/, 2024. [Online; accessed 6-December-2024].

[2] M. Abbas and V. Betz. Latency insensitive design styles for fpgas. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2018.

[3] Achronix Semiconductor Corporation. *Speedster7t Network on Chip User Guide (UG089)*, 2019.

[4] S. A. Alam, D. Gregg, G. Gambardella, T. Preusser, and M. Blott. On the rtl implementation of finn matrix vector unit. *ACM Transactions on Embedded Computing Systems.*, 2022.

[5] Altera. *Quartus Prime Pro Edition User Guide Partial Reconfiguration*, April 2024.

[6] AMD. *UG1120: Alveo Data Center Accelerator Card Platforms*, October 2023.

[7] AMD. *UG904: Vivado Design Suite User Guide: Implementation*, May 2023.

[8] AMD. *UG906: Vivado Design Suite User Guide: Design Analysis and Closure Techniques*, May 2023.

[9] AMD. *UG974: UltraScale Architecture Libraries Guide*, October 2023.

[10] AMD. Runtime-first fpga interchange routing contest @ fpga'24. https://xilinx.github.io/fpga24_routing_contest/index.html, 2024. [Online; accessed 8-December-2024].

[11] AMD. *UG1366: VCK190 Evaluation Board*, September 2024.

[12] AMD. *UG1393: Vitis Unified Software Platform Documentation: Application Acceleration Development*, July 2024.

[13] AMD. *UG1399: Vitis High-Level Synthesis User Guide*, July 2024.

[14] AMD. *UG909: Vivado Design Suite User Guide: Dynamic Function eXchange*, June 2024.

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

[16] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, 11(3), 2018.

[17] A. Boutros and V. Betz. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[18] A. Boutros, E. Nurvitadhi, and V. Betz. Architecture and application co-design for beyond-fpga reconfigurable acceleration devices. *IEEE Access*, 10:95067–95082, 2022.

[19] A. Boutros, E. Nurvitadhi, and V. Betz. Rad-sim: Rapid architecture exploration for novel reconfigurable acceleration devices. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 438–444, 2022.

[20] I. Bustany, G. Gasparyan, A. Gupta, A. B. Kahng, M. Kalase, W. Li, and B. Pramanik. The 2023 mlcad fpga macro placement benchmark design suite and contest results. In *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023.

[21] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

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

[23] D. Chen, J. Cong, and P. Pan. Fpga design automation: A survey. *Foundations and Trends® in Electronic Design Automation*, 1(3):195–330, 2006.

[24] Y.-k. Choi, P. Zhang, P. Li, and J. Cong. Hlscope+: Fast and accurate performance estimation for fpga hls. In *Proceedings of the International Conference on Computer-Aided Design*, page 691–698. IEEE Press, 2017.

[25] J. Chromczak, M. Wheeler, C. Chiasson, D. How, M. Langhammer, T. Vanderhoek, G. Zgheib, and I. Ganusov. Architectural enhancements in intel® agilex™ fpgas. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2020.

[26] J. Coole and G. Stitt. Bpr: fast fpga placement and routing using macroblocks. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, page 275–284, New York, NY, USA, 2012. Association for Computing Machinery.

[27] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 129–132, 2018.

[28] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% lut utilization). In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 69–78, February 1999.

[29] A. DeHon. Compact, multilayer layout for butterfly fat-tree. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 206–215. ACM, July 2000.

[30] A. DeHon. Rent's rule based switching requirements. In *Proceedings of the System-Level Interconnect Prediction Workshop*, pages 197–204. ACM, March 2001.

[31] A. DeHon. Unifying mesh- and tree-based programmable interconnect. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1051–1065, October 2004.

[32] A. DeHon. Fundamental underpinnings of reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):355–378, March 2015.

[33] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek. Stream computations organized for reconfigurable execution. *Journal of Microprocessors and Microsystems*, 30(6):334–354, September 2006.

[34] L. Du, T. Liang, S. Sinha, Z. Xie, and W. Zhang. FADO: Floorplan-aware directive optimization for high-level synthesis designs on multi-die fpgas. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2023.

[35] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer. Xilinx adaptive compute acceleration platform: Versal architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2019.

[36] B. Gottschall, T. Preußer, and A. Kumar. Reloc — an open-source vivado workflow for generating relocatable end-user configuration tiles. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 211–211, 2018.

[37] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. ACM SIGPLAN, ACM, June 1982. SIGPLAN Notices, Volume 17, Number 6.

[38] A. Guerrieri, S. Guha, C. Lavin, E. Hung, L. Josipović, and P. Ienne. Dynarapid: Fast-tracking from c to routed circuits. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 24–32, 2024.

[39] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 81—-92, New York, NY, USA, 2021. ACM.

[40] L. Guo, P. Maidee, Y. Zhou, C. Lavin, E. Hung, W. Li, J. Lau, W. Qiao, Y. Chi, L. Song, Y. Xiao, A. Kaviani, Z. Zhang, and J. Cong. Rapidstream 2.0: Automated parallel implementation of latency–insensitive fpga designs through partial reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems*, 16(4), 2023.

[41] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong. Rapidstream: Parallel physical implementation of FPGA HLS designs. In

Proceedings of the International Symposium on Field-Programmable Gate Arrays, pages 1–12, 2022.

[42] D. L. How and S. Atsatt. Sectors: Divide conquer and softwarization in the design and validation of the Stratix 10 FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 119–126, May 2016.

[43] Y. Huan and A. DeHon. FPGA optimized packet-switched NoC using split and merge primitives. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 47–52. IEEE, December 2012.

[44] Intel. *Intel Agilex® 7 M-Series FPGA Network-on-Chip (NoC) User Guide*, July 2023.

[45] A. K. Jain, C. Ravishankar, H. Omidian, S. Kumar, M. Kulkarni, A. Tripathi, and D. Gaitonde. Modular and lean architecture with elasticity for sparse matrix vector multiplication on fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 133–143, 2023.

[46] L. Josipović, A. Guerrieri, and P. Ienne. Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[47] 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.

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

[49] N. Kapre and A. DeHon. An NoC traffic compiler for efficient FPGA implementation of parallel graph applications. In *Proceedings of the Reconfigurable Communication-Centric Systems on Chip*, pages 87–94, May 2010.

[50] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 1–8, 2015.

[51] N. Kapre and J. Gray. Hoplite: A deflection-routed directional torus NoC for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 10(2):14:1–14:24, Mar. 2017.

[52] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet-switched vs. time-multiplexed FPGA overlay networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–213. IEEE, 2006.

[53] N. Kapre, H. Ng, K. Teo, and J. Naude. Intime: A machine learning approach for efficient selection of fpga cad tool parameters. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 23–26, New York, NY, USA, 2015. Association for Computing Machinery.

[54] G. Karypis and V. Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. http://www.cs.umn.edu/~metis, 2009.

[55] D. Koch and C. Beckhoff. Hierarchical reconfiguration of fpgas. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 1–8, 2014.

[56] D. Koch, C. Beckhoff, and J. Teich. Recobus-builder — a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *2008 International Conference on Field Programmable Logic and Applications*, pages 119–124, 2008.

[57] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges.* Now Foundations and Trends, 2008.

[58] 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, 2016.

[59] B. S. Landman and R. L. Russo. On pin versus block relationship for partitions of logic circuits. *IEEE Transactions on Computers*, 20, 1971.

[60] I. Lang, Z. Huang, and N. Kapre. Exploring the impact of switch arity on butterfly fat tree FPGA nocs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2020.

[61] M. Langhammer, G. Baeckler, and S. Gribok. Spiderweb - high performance fpga noc. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 115–118, 2020.

[62] C. Lavin and E. Hung. Invited paper: Rapidwright: Unleashing the full power of fpga technology with domain-specific tooling. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–7, 2023.

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

[64] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, 2011.

[65] 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 the International Conference on Field-Programmable Logic and Applications*, September 2011.

[66] C. E. Leiserson. VLSI theory and parallel supercomputing. MIT/LCS/TM 402, MIT, 545 Technology Sq., Cambridge, MA 02139, May 1989.

[67] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[68] C. Lo and P. Chow. Multi-fidelity optimization for high-level synthesis directives. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 272–279, 2018.

[69] C. Lo and P. Chow. Hierarchical modelling of generators in design-space exploration. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 186–194, 2020.

[70] S. Ma, Z. Aklah, and D. Andrews. Just in time assembly of accelerators. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 173–178, 2016.

[71] G. S. Malik and N. Kapre. Enhancing butterfly fat tree NoCs for FPGAs with lightweight flow control. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2019.

[72] K. Manev, J. Powell, K. Matas, and D. Koch. byteman: A bitstream manipulation framework. In *Proceedings of the International Conference on Field-Programmable Technology*, pages 1–9, 2022.

[73] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for fpgas. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 111–117, 1995.

[74] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz. Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems*, 13(2), June 2020.

[75] T. Nguyen, Z. Blair, S. Neuendorffer, and J. Wawrzynek. Spades: A productive design flow for versal programmable logic. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2023.

[76] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 37–46, 2012.

[77] D. Park and A. DeHon. REFINE: Runtime Execution Feedback for INcremental Evolution on FPGA designs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2024.

[78] D. Park, Y. Xiao, and A. DeHon. Fast and flexible FPGA development using hierarchical partial reconfiguration. In *Proceedings of the International Conference on Field-Programmable Technology*, 2022.

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

[80] D. Park, Z. Yao, Y. Xiao, and A. DeHon. Asymmetry in butterfly fat tree FPGA noc. In *Proceedings of the International Conference on Field-Programmable Technology*, 2023.

[81] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[82] K. D. Pham, E. Horta, and D. Koch. BITMAN: A tool and api for FPGA bitstream manipulations. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*, 2017.

[83] B. C. Schafer and Z. Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2020.

[84] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. Yosys+nextpnr: An open source framework from verilog to bitstream for commercial fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 1–4, 2019.

[85] L. Shannon and P. Chow. Maximizing system performance: using reconfigurability to monitor system communications. In *Proceedings of the International Conference on Field-Programmable Technology*, 2004.

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

[87] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems*, 2022.

[88] M. Stojilović. Parallel fpga routing: Survey and challenges. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 1–8, 2017.

[89] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel. Network-on-chip programmable platform in versaltm acap architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2019.

[90] A. R. Tareen, M. Meyer, C. Plessl, and T. Kenter. Hihispmv: Sparse matrix vector multiplication with hierarchical row reductions on fpgas with high bandwidth memory. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 32–42, 2024.

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

[92] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2017.

[93] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. Lamda: Learning-assisted multi-stage autotuning for fpga design closure. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 74–77, 2019.

[94] K. Vipin and S. A. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys*, 51(4):72.1–72.39, 2018.

[95] Y. Xiao. *Accelerating FPGA Developments from C to Bitstreams by Partial Reconfiguration*. PhD thesis, University of Pennsylvania, 2023.

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

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

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

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

[100] Y. Xiao, D. Park, Z. J. Niu, A. Hota, and A. Dehon. Exhipr: Extended high-level partial reconfiguration for fast incremental fpga compilation. *ACM Transactions on Reconfigurable Technology and Systems*, 17(2), 2024.

[101] Xilinx. 66314 - vivado congestion. https://support.xilinx.com/s/article/66314?language=en_US, 2020. [Online; accessed 2-December-2024].

[102] Xilinx. *UG984: MicroBlaze Processor Reference Guide*, October 2021.

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

[104] Xilinx, Inc. *UG946: Vivado Design Suite Tutorial: Hierarchical Design*, April 2015.

[105] Xilinx, Inc. *DS890: UltraScale Architecture and Product Data Sheet: Overview*, May 2019.

[106] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang. A parallel bandit-based approach for autotuning fpga compilation. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 157–166, New York, NY, USA, 2017. Association for Computing Machinery.

[107] S. Yazdanshenas and V. Betz. Interconnect solutions for virtualized field-programmable gate arrays. *IEEE Access*, 2018.

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

[109] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 269–278, 2018.

[110] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt. Rwroute: An open-source timing-driven router for commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 15(1):1–27, 2021.