Dynamically Managing FPGAs for Efficient Computing

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Department of Electrical and Computer Enginnering

Marie Nguyen

B.S., Electronics and Signal Processing, INPT-ENSEEIHT

M.S., Electronics and Signal Processing, INPT-ENSEEIHT

M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, PA

December 2020

Acknowledgments

Doing a PhD has been an incredibly enriching journey during which I learnt how to think and teach. I would like to express my gratitude to people that have helped me along the way. First, I want to thank Prof. James C. Hoe who is a great person and advisor. James has provided very valuable guidance on my academic work and taught me how to think, write clearly and conduct high-quality research. He has always been supportive of my research and kept encouraging me during the most difficult times of my PhD. I want to thank him for his patience, the dedication he has for his students, and the effort he puts into bringing the best out of his advisees. I am very grateful for his teaching and mentoring, and will remember them well beyond my time at CMU. Thank you James for all the opportunities you gave me.

I want to thank Dasu Aravind, Prof. Srinivasa Narasimhan and Prof. Anthony Rowe for serving on my thesis committee. I would like to thank Dasu for the monthly conversations we have had over the past year that have helped me refine my thesis work. I also want to thank him for being one of my mentors during my internship at Intel. I would like to thank both Prof. Srinivasa Narasimhan and Prof. Anthony Rowe for working with me during the past years on the SmartHeadlight and the AR/VR headset projects. Specifically, I am grateful to Prof. Srinivasa Narasimhan for teaching me the basics of computer vision that I could leverage in my research.

During my time at CMU, I had the opportunity to collaborate with brilliant researchers working at Intel. Specifically, I would like to thank Eriko Nurvitadhi, Scott Weber, and Patrick Koeberl that were my supervisors during my internship at Intel along with Dasu Aravind. Eriko, Scott and Patrick have shown great interest in my research and have helped me refine it. I would like to thank them for the monthly meetings we have had for almost a year. I am grateful to Eriko and Patrick for their technical and professional mentoring. I greatly enjoyed the technical discussions with Scott, one of the world experts in partial reconfiguration. I learnt a lot about partial reconfiguration and its intricacies through these conversations. I have been fortunate to have great technical conversation with current and past students from my research group, namely, Guanglin Xu, Joe Melber, Zhipeng Zhao, Nathan Serafin, Shashank Obla, Yu Wang, Gabriel Weisz, and Michael Papamichael. I would like to thank them for proofreading my work, for their constructive feedback and for helping me with technical issues when I was struggling.

Besides my research group, I met great professors and students on A-level: Prof. Tze Meng Low, Prof. Osman Yagan, Thom Doru Popovici, Anuva Kulkarni, Daniele Spampinato, Richard Veras, Rashad Eletreby, John Filleau, Jin Huang, and Jiyuan Zhang. I have had many interesting conversations with people on A-level

that taught me new perspectives on research and helped me better understand other domains. I especially want to thank Prof. Tze Meng Low that kept challenging me during my early years of the PhD and that gave much-needed feedback on my research. Thom has been a great friend during these years. I want to thank him for the great conversations we have had on the many aspects of research, and for the helpful feedbacks he gave on my writings and presentations. He has always been available and patient to answer all my questions about paper/thesis writing, and non-work related subjects. Anuva has been a good friend with whom I have had many non-technical conversations that helped me relax outside of the lab.

I also met with many great scientists and students when working on different research and class projects at CMU, to name a few, Robert Tamburo, Sezal Jain, Priya Mahajan, Adeola Bannis, and Evgeny Toropov. I would like to thank Sezal, Priya, and Adeola for the great times we have had at CMU, and Evgeny for the very entertaining board game nights and outings he organized.

Beyond CMU, I feel very lucky to have the support from my loving parents and friends from France, notably, Lou and Francois, that have always been by my side when I needed them the most. I can not thank my mom enough for all she has done for the past 30 years. She has been a source of strength, comfort and encouragement in the most difficult times. She taught me to always strive for better and be brave not matter what. Thank you mom for making me who I am today.

Finally, I would like to acknowledge Intel Corporation and Xilinx for their generous financial and equipment donation. Also, this work could not have happened without the support of the National Science Foundation (CCF-1012851) and SRC/JUMP (CONIX).

Abstract

Field-programmable gate arrays (FPGAs) have undergone a dramatic transformation from a logic technology to a computing technology. This transformation is pulled by the computing industry's need for more power/energy efficiency than software can achieve and, at the same time, more flexibility than ASICs. Nonetheless, FPGA designers still share a similar design methodology with ASIC designers. Most notably, at design time, FPGA designers commit to a fixed allocation of logic resources to modules in a design. In other words, FPGAs are mostly still used like an "ASIC" despite being runtime reprogrammable. Through partial reconfiguration (PR), parts of an FPGA design can be reconfigured at runtime while the remainder continues to operate without disruption. PR enables what has been possible on general-purpose processors for decades. For instance, multiple tasks can be time-multiplexed on a smaller FPGA, which can result in area/device cost, power and energy reduction, compared to statically mapping tasks on a larger FPGA. PR can become a relevant technology for an emerging class of AI-driven applications that (1) need to support many compute intensive tasks with real-time requirements and (2) are often deployed on a small, low-end FPGA due to area, cost, power or energy concerns (e.g., smart cars/robots/cameras at the Edge). For such applications, using a large expensive FPGA is typically not a viable option.

Though PR is a promising technology and has been supported by FPGA tools for over a decade, it is still a feature waiting to be proven for its commercial value. The reconfiguration time (between few to tens of milliseconds on today's FPGAs), also referred as PR time, is often considered as one of the major hurdles preventing a more widespread use of PR. While the non-trivial PR time represents a technical challenge, we believe that a more important question to address is "When, how and why should an FPGA designer consider using PR?". Addressing this question requires to (1) identify applications that can tolerate PR time and still benefit from a PR approach, (2) design good architectural and runtime management strategies to build efficient designs leveraging PR, and (3) evaluate whether the area/device cost, power or energy benefits are important enough to justify a transition from a statically mapped design.

This thesis seeks to advance the state-of-the-art in the dynamism of computing FPGAs by tackling the aforementioned challenges. Specifically, we demonstrate that a design exploiting PR can be more area/device cost, power or energy efficient than a statically mapped design (ASIC-style design) with slack. Slack occurs when all resources occupied by an ASIC-style design are not active all the time. Using PR, a designer can attempt to reduce slack by changing the allocation of resources over time. In this work, we identify slack's reduction as the most important opportunity for improvement available to PR-style designs. We refer to a

PR-style design as a design in which logic resources are allocated to different modules of one design over time using PR. We develop efficient PR allocation and execution strategies to reduce slack, and show through analytical modeling and implemented designs that a PR-style design can outperform an ASIC-style design in challenging scenarios that have to deliver required performance under strict area, cost, power, and energy constraints. Further, we leverage the findings and analysis from our theoretical investigation to develop a soft-logic-realized framework for accelerating computer vision with real-time requirements (30+ fps). This framework includes the necessary architectural and runtime management strategies to support spatial and temporal sharing of the FPGA fabric at a very fine-grain (i.e. the time interval between reconfigurations is within millisecond range) while meeting performance requirements. Using the framework, we design and implement efficient PR-style designs to quantify the performance, area/device cost, power and energy benefits of PR-style designs relative to ASIC-style designs and to software implementations. Notably, we show that a PR-style design can be more power and energy efficient than an ASIC-style design even when frequently reconfiguring the fabric (i.e. when more than half of the execution time is spent reconfiguring the fabric) and under specific conditions. We also make projections on the impact of higher PR speed on the costs and benefits of using PR at a very fine-grain. Through our study, we find that, while higher reconfiguration speed can make a PR-style more area/device cost efficient, the power/energy overhead incurred in a PR-style design due to, for instance, fabric reconfigurations and additional data movement can make a PR approach less power/energy efficient than an ASIC-style design.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations, Challenges and Goals

Motivations. Field-programmable gate arrays (FPGAs) have undergone a dramatic transformation from a logic technology to a computing technology [14,45,65]. This transformation is pulled by the computing industry's need for more power/energy efficiency than software can achieve and, at the same time, more flexibility than ASICs. Nonetheless, FPGA designers still share a similar design methodology with ASIC designers. Most notably, at design time, FPGA designers commit to a fixed allocation of logic resources to modules in a design. In other words, FPGAs are mostly still used like an "ASIC" despite being runtime reprogrammable. Through partial reconfiguration (PR), parts of an FPGA design can be reconfigured at runtime while the remainder continues to operate without disruption. PR enables what has been possible on general-purpose processors for decades. For instance, multiple tasks can be time-multiplexed on a smaller FPGA, which can result in area/device cost, power and energy reduction, compared to statically mapping tasks on a larger FPGA.

Partial Reconfiguration Overview. Using PR, an FPGA is divided into multiple runtime reconfigurable regions called reconfigurable partitions (RPs) or PR regions. The non reconfigurable part of the design typically includes the I/O infrastructure defining the interconnection between the PR regions to the rest of the system. The number, sizing and interconnection between reconfigurable partitions are decided at design time during the floorplanning process. Reconfigurable partitions can be reprogrammed at runtime with module bitstreams built offline. Any module built at design time can be separately mapped in a specific reconfigurable partition if the module has matching I/O interfaces with the reconfigurable partition and

consumes less resources than available in the reconfigurable partition. The set of module bitstreams built at design time constitutes a module library.

At runtime, the reconfigurable partitions can be reprogrammed with different modules over time to accelerate different tasks. The reprogramming of the PR regions can be triggered from outside the FPGA (e.g., an external CPU), by logic on the fabric, or by an embedded processor.

Challenges. Though PR is a promising technology and has been supported by FPGA tools for over a decade, it is still a feature waiting to be proven for its commercial value. The non-trivial reconfiguration time (between few to tens of milliseconds on today's FPGAs), also referred as PR time (assumed to be proportional to the PR region size), is often considered as one of the major hurdles preventing a more widespread use of PR. However, despite the steady improvement of PR technology over the past decade (faster PR time, better tool support), PR still remains an under-appreciated capability. While the non-trivial PR time represents a technical challenge, we believe that a more important question to address is "When, how and why should an FPGA designer consider using PR". Addressing this question requires to (1) identify applications that can tolerate PR time overhead, (2) design good architectural and runtime management strategies to build efficient designs leveraging PR, and (3) evaluate whether the area/device cost, power or energy benefits of design are important enough to justify a transition from a statically mapped design.

Goals. This thesis seeks to advance the state-of-the-art in the dynamism of computing FPGAs by tackling the aforementioned challenges. Specifically, we demonstrate that a design exploiting PR can be more area/device cost, power or energy efficient than a statically mapped design (ASIC-style design) with slack. Slack occurs when all resources occupied by an ASIC-style design are not active all the time, that is, resources are under-utilized in the ASIC-style design. Using PR, a designer can attempt to reduce under-utilization by changing the allocation of resources over time. In this work, we find that reducing slack is the most important opportunity for improvement available to PR-style designs. We refer to a PR-style design as a design in which logic resources are allocated to different modules of one design over time. In return, a PR-style design may be faster, smaller, and/or consume less power/energy than an ASIC-style design. We develop efficient PR allocation and execution strategies to reduce slack, and show through analytical modeling and implemented designs that a PR-style design can outperform an ASIC-style design in challenging design scenarios that have to deliver required performance under strict area, cost, power, and energy constraints (e.g., [48,86]).

In this work, we also leverage PR to spatially and temporally share the FPGA fabric by multiple tasks

for domain-specific acceleration. Specifically, we investigate the costs and benefits of using PR at a very fine-grain (i.e. the time interval between reconfigurations is within the millisecond range). Note that, in this work, we do not focus on using PR in a "role-and-shell" approach [9,42]. In this use-case, the FPGA is reconfigured very infrequently i.e. at a coarse-grain (minutes, hours or days between reconfigurations). Most FPGA resources are contained in a single PR region that is enclosed by a static shell region that provides I/O and isolation. Independent designs with different functionalities, or roles, can be loaded as required in the PR region over time (e.g., [9,42]). Each role is an ASIC-style design created to use the entire PR region alone, with no consideration for sharing resources or interacting with other roles.

The work presented in this thesis is especially relevant for an emerging class of AI-driven applications that (1) need to support many compute intensive tasks with real-time requirements (e.g., smart cars/robots/cameras at the Edge) and (2) are often deployed on a small, low-end FPGA due to area, cost, power or energy concerns [10,97]). For such applications, using a large expensive FPGA is typically not a viable option.

1.2 Partial Reconfiguration for Design Optimization

In this thesis, we address the questions of when, how and why FPGA designers should consider using PR. Notably, we demonstrate that a PR-style design can reduce slack with better resource scheduling, and therefore, improve upon an ASIC-style. By improving upon an ASIC-style design, we mean either that (1) a PR-style design is smaller or consumes less power/energy than the ASIC-style design while achieving the same performance, or (2) a PR-style design achieves better performance and uses less area or power/energy than the ASIC-style design. The first part of the thesis covers the theoretical foundations to tackle the aforementioned questions and focuses on applications with a single task. The second part of this work leverages the analysis and findings from the first part, and discusses the practical challenges, costs and benefits of using PR for domain-specific acceleration with multiple tasks spatially and temporally sharing the fabric. In the remainder of this section, we give an overview of our work.

Reducing Slack with PR. In this work, we find that reducing slack is the main means for PR-style designs to improve over ASIC-style designs. Slack stems from hard-to-avoid source of inefficiencies (e.g., operation dependencies or pipeline imbalancing) and results in some of the occupied resources being left idle or under-utilized. In other words, all occupied resources in the design are not active all the time. In ASIC-style designs, a module must occupy logic resources even when it is inactive since resources are statically

allocated. Slack can result in (1) the design not running at the desired performance given an area budget, or (2) the design running at the desired performance but being too big to fit in the given area. In an ASIC-style design, the designer is not exploiting the fact that FPGA logic resources can be re-allocated over time. Using PR, a designer can attempt to reduce slack by changing the allocation of resources over time.

PR Execution Strategies. To reduce slack and improve over the performance-area Pareto front of ASIC-style designs, we develop a set of PR execution strategies (allocation and scheduling) for a non-trivial range of applications. An application consists of a set of independent tasks. A task is modeled as a directed graph where nodes are processing stages, and edges represent stage dependencies. Each processing stage is accelerated on the FPGA by a hardware module that includes the necessary FPGA logic, hardened compute block and memory resources. Modules can execute concurrently and have multiple implementation variants with different performance-area trade-offs. Dependent modules share data either through (1) external memory or (2) on-chip memory. This choice of an execution model allows to cover a range of applications that benefit from FPGA acceleration (e.g., streaming pipelines). For instance, in video analytics or image processing applications, data is streamed from an input source (e.g., camera) directly to a processing pipeline mapped on the FPGA. Pipeline stages are accelerated by modules that share data through on-chip connections.

Analytical Modeling. Further, while FPGA tools have been improving at a rapid pace, FPGA design development is still tedious and even more so when designing with PR. Consequently, it is important to determine whether a PR-style design can improve upon an ASIC-style design during the early stage of design development i.e. before crafting a PR-style design. To this end, we propose a first-order analytical model to help a designer (1) determine a suitable PR execution strategy for a given problem and (2) analyze the throughput and latency of ASIC-style and PR-style designs. The model enables quick exploration of the design space to help decide if a PR-style design can be beneficial. The model accounts for the impact of PR time on performance, and assumes the existence of a module library that has been pre-characterized in terms of latency, throughput, area, etc.

Framework for Computer Vision Acceleration. Starting from Chapter 5, we discuss our investigation on designing and implementing PR-style designs for domain-specific acceleration. Specifically, we present the design and implementation of a soft-logic-realized framework for accelerating computer vision applications with real-time performance requirements (30+ fps). Computer vision is picked as an example domain that can benefit from PR since many vision applications can (1) be accelerated by streaming pipelines, and therefore, benefit from FPGA acceleration, (2) are deployed on systems with limited area, power or energy

budget, and (3) have slack and tolerance for current DPR overhead. We explain in more details the latter point in Chapter 5.

In the framework, vision tasks can spatially and temporally share the FPGA. Spatial sharing means that multiple tasks can be mapped and executed simultaneously on the FPGA. Temporal sharing means that different tasks can be reprogrammed over time. The framework has the following components:

- a plug-and-play architecture with multiple PR regions that communicate with each other and the rest of the system via a static I/O infrastructure,
- a runtime system which manages the reprogramming of the FPGA with tasks specified in an application code. In our implemented prototype, the runtime system and the application code both run on an embedded CPU.
- a module library which contains pre-built modules that are used to reprogram the PR regions.

While most works use PR at a coarse grain (hours or days between reconfigurations, for instance, in a "role-and-shell" approach) due to the significant reconfiguration time on today's FPGAs, we further explore a more aggressive, fine-grain use of PR (few milliseconds between reconfiguration) referred to as real-time time-sharing. In real-time time-sharing, multiple vision pipelines can round-robin execute in the time scale of camera frame (16.7 ms): in 16.7 ms, multiple pipelines are reconfigured on the FPGA and each pipeline processes one frame. To support such an aggressive PR usage, the framework includes the necessary architectural and runtime mechanisms for hiding, amortizing, and reducing the reconfiguration time.

PR Benefits Evaluation and Projection. Using the framework, we implement PR-style designs for accelerating vision applications to: (1) demonstrate that real-time time-sharing is feasible despite the non-trivial reconfiguration time on today's FPGAs. By designing the appropriate architectural and runtime strategies to mitigate PR time, we show that multiple pipelines can be real-time time-shared and achieve useful performance (30+ fps). (2) quantify the area/device cost, power and energy benefits of PR-style designs relative to ASIC-style designs and software implementations in case studies of implemented designs. Notably, we demonstrate that using PR at fine-grain can be beneficial in terms of area/device cost, power or energy even when most of the execution time is spent reconfiguring the fabric.

We conduct a limit study to further our understanding of the costs and benefits when doing real-time time-sharing, and notably, make projections for higher PR speeds. Real-time time-sharing serves as a proxy for very aggressive, fine-grain PR usages that can be highly beneficial in terms of area/device cost but also

incur the most overhead in terms of time and power/energy due to (1) the very frequent reconfigurations, (2) the difference in clock frequency compared to an ASIC-style design to compensate for PR time, and (3) the additional data movement required compared to an ASIC-style design (to fetch the PR bitstreams and to load/store intermediate data from/to off-chip memory). In this study, we highlight the importance of accounting for the power/energy overhead when using PR at a very fine-grain for a fair comparison between PR-style and ASIC-style designs. While the area/device cost efficiency of a PR-style design grows with increased PR speed, we demonstrate that a PR-style design can be less power/energy efficient than an ASIC-style design due to the power/energy overhead incurred when using PR. This finding emphasizes the importance of developing analytical models and performing preliminary analyses to gain a better understanding of the potential costs and benefits of a PR-style design prior to implementing a physical solution.

1.3 Contributions

The work of this thesis explores the questions of when, how, and why FPGA designers should consider using PR. Specifically, the contributions are as follows:

- We identify slack as an opportunity that can be exploited by PR-style designs to improve upon ASIC-style designs. Slack occurs when some occupied resources in a design are not active all the time due to hard-to-avoid sources of inefficiencies (e.g., operation dependencies, pipeline imbalancing).

- We develop a set of PR execution strategies to reduce slack in ASIC-style designs and improve a design's performance or area. We also propose an analytical model to help the designer choose the best PR execution strategy for a given problem and evaluate whether a PR-style design is more beneficial than an ASIC-style design.

- We develop a soft-logic-realized framework for spatial and temporal sharing of the FPGA by multiple computer vision tasks. The framework includes the necessary architectural and runtime management strategies to support coarse and fine-grain temporal sharing. Notably, we investigate a specific fine-grain usage of PR referred to as real-time time-sharing. In real-time time-sharing, multiple vision tasks are reconfigured on the FPGA and each pipeline processes one frame in the time-scale of a camera frame.

- Using the framework, we design and implement application examples to (1) demonstrate the feasibility

of real-time time-sharing despite the non-trivial PR time on today's FPGAs and (2) quantify the area/device cost, power and energy benefits of PR-style designs relative to ASIC-style designs and software implementations. Notably, we show that a PR-style design can be more power/energy efficient than an ASIC-style design even when most of the execution time is spent on fabric reconfiguration in a case study.

- We also make projections on the benefits and costs of doing real-time time-sharing with faster PR. While faster PR results in greater area/device cost efficiency, the power/energy overhead of a PR-style design can outweigh the benefits of using a smaller FPGA.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides background information on the architecture of an FPGA and partial reconfiguration. Chapter 3 discusses the terminology used in this thesis and presents the insights behind why a PR-style design can be faster or smaller than an ASIC-style. Chapter 4 presents the PR execution and allocation strategies for developing efficient PR-style designs and explains our analytical model. Chapter 5 presents our soft-logic-realized framework for accelerating computer vision applications and details the architectural and runtime mechanisms to support real-time time-sharing. Chapter 6 presents our quantification on the area/device cost, power and energy benefits of PR-style designs relative to ASIC-style designs for applications built on top of our framework. Chapter 7 discusses the potential cost and benefits of having faster PR. Chapter 8 discusses related work. Finally, Chapter 9 presents concluding remarks and future directions.

Chapter 2

Background

This section first covers the basic architecture of an FPGA before discussing the motivations behind using an FPGA for computing. Finally, we explain the PR design flow in more details.

2.1 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) are semiconductor devices built off arrays of lookup table (LUT)-based blocks referred to as configurable logic blocks (CLBs) [91] or arithmetic logic modules (ALMs) [43] in Xilinx and Intel terminology, respectively (Figure 2.1). CLBs and ALMs consist of lookup tables (LUTs), registers, and multiplexers, and are integrated with other components on an FPGA, such as embedded memory blocks, embedded digital signal processing (DSP) blocks, via programmable interconnect. Embedded memory and DSP blocks are typically organized in columns and interleaved in a specific pattern depending on the target FPGA. An FPGA fabric is subdivided into multiple regions called clock regions or sectors (terminology specific to Intel Stratix 10 FPGAs) in Xilinx or Intel's terminology, respectively. Clock regions may contain a different amount and type of resources depending on the target FPGA. The distribution of resources (LUTs and hard blocks) on the fabric is not necessarily uniform. On the other hand, sectors are homogeneously-sized, that is, they contain the same amount and type of resources.

After the device has been manufactured, FPGA hardware resources can be reconfigured to implement different functions by modifying the content of the FPGA configuration memory made of volatile memory cells (mostly static random access memory(SRAM)-based). The configuration memory is organized into an array of configuration frames which are the smallest addressable segments of the FPGA configuration

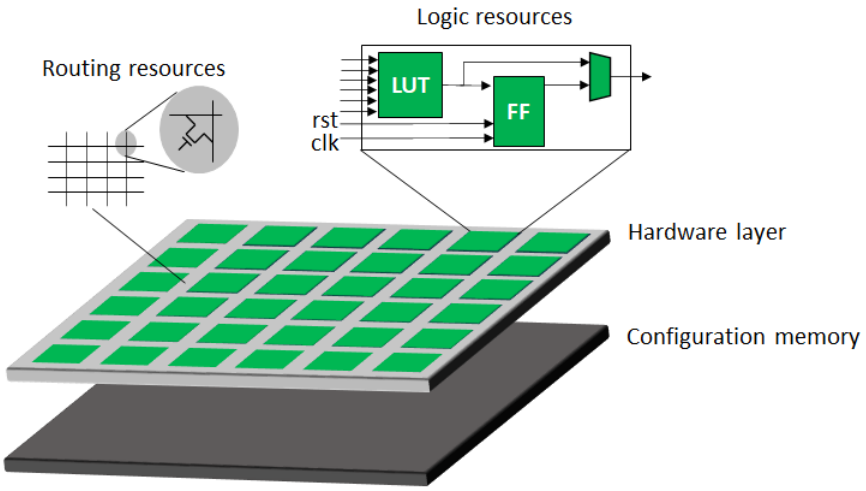

Figure 2.1: A typical FPGA architecture is comprised of two layers: a hardware logic and a configuration memory layer.

memory. Reconfigurable frames are built from a discrete number of configuration frames. A reconfigurable frame represents the smallest reconfigurable region within an FPGA. In Xilinx devices, a reconfigurable frame is one element (CLB, block RAM, DSP) wide by one clock region high [93]. The number of resources in these frames vary by device family [93]. Please refer to [93] for more detailed information, rules and restrictions. Data written to the configuration memory cells is contained in a bitstream file generated at design time. A bitstream contains all the necessary information to specify how logic resources available (CLBs, on-chip memory blocks, DSP blocks, routing resources) on the FPGA should be configured to execute a given functionality.

2.2 ASIC-Style Design for Computing

The features of post-manufacturing programmability differentiate FPGAs from Application Specific Integrated Circuits (ASICs), which are hardened for application specific functions. General-purpose processors rely on a fixed architecture support to deliver software-based programmability through a set of instructions. In other words, instructions are scheduled to execute on a fixed number of compute units and leverage a fixed caching architecture that can not be changed post-manufacturing. In contrast, FPGAs' reprogrammability is hardware-based and allows to create application-specific architectures by reconfiguring the available logic and routing resources. As transistors are directly used to accelerate application-specific computations, using

an FPGA may provide better silicon area utilization, potentially leading to improved performance and/or greater area, power and energy efficiency.

With the computing industry's need for more energy-efficiency than software can achieve and, at the same time, more flexibility than ASICs, there is a growing emphasis on deploying FPGAs for domain-specific acceleration [35,38,45]. FPGAs have been proven particularly beneficial for accelerating streaming computations when data is produced at a given rate by an off-board source and is directly streamed to the FPGA for processing (e.g., video analytics or network function processing). We are also starting to see FPGAs' post-manufacturing programmability being recognized as a deciding feature in selecting FPGAs over ASICs, for instance, in the data centers by the industry's largest players [9,12,68].

2.3 Partial Reconfiguration

The discussion of PR in this section is based on the Xilinx PR flow [93].

Partial Reconfiguration (PR) allows some regions of the FPGA fabric, referred to as PR regions or reconfigurable partitions (RPs), to be reprogrammed at runtime [93]. At design time, the FPGA needs to be partitioned in (1) a non reconfigurable region that typically includes the I/O infrastructure, and (2) one or multiple PR regions that can be reprogrammed individually at runtime. Each PR region can be reprogrammed with partial bitstreams built for this region at design time to accelerate specific functions.

Static Region and Reconfigurable Partitions. The cartoon in Figure 2.2(a) depicts an FPGA fabric organized into a top-level static region enclosing a runtime reconfigurable region subdivided as two reconfigurable partitions. In Xilinx environment, a reconfigurable partition appears in the top-level design as a "black-box" submodule with know I/O ports but opaque internals. In Figure 2.2(a), the two reconfigurable partitions are shown to have the same port list, simply A and B in this toy example. A reconfigurable partition can have an arbitrary rectilinear outline and can cross clock regions. Resources enclosed within the outline (e.g., LUTs, BRAM blocks) are part of the reconfigurable partition and can be reprogrammed at runtime.

Build Flow. At design time, the locations of the PR region I/O pins, the port nets, and the physical boundary of the static and PR regions are fixed. The net for a port (whether input or output) terminates at a reserved interface point called partition pin. Partition pins are automatically created and placed by the tool when defining a reconfigurable partition. It is also possible for the designer to change the location of the partition pins. These virtual I/O are established within interconnect tiles as the anchor points that remain

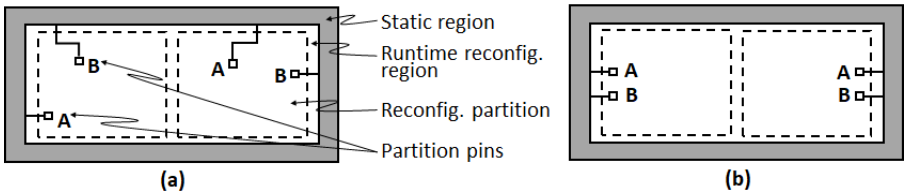

Figure 2.2: An FPGA fabric organized into a top-level static region enclosing a runtime reconfigurable region subdivided as two reconfigurable partitions. The partition pins in (a) have been arbitrarily placed; the partition pins in (b) have been placed deliberately.

consistent from one module to the next. No physical resources such as LUTs or flip-flops are required to establish these anchor points, and no additional delay is incurred at these points [93]. In Figure 2.2(a), the partition pins A and B are shown as to have been placed arbitrarily by the tool. The figure also shows the placed-and-routed nets that connect the partition pins out to the static region. Figure 2.2(b) shows another version where the partition pins have been deliberately placed during floorplanning.

Bitstream Versions. In addition to defining a layout for the fabric at design time i.e. determining the number, shape and size of reconfigurable partitions), modules are synthesized and placed & routed for the reconfigurable partitions. Any module compiled at design time can be separately mapped in a specific reconfigurable partition if the module satisfies two requirements (1) the module has matching I/O interfaces with the reconfigurable partition (same I/O ports) and (2) the module consumes less resources than available in the reconfigurable partition. When a module can be placed in multiple RPs, the module needs to be separately placed-and-routed for the different RPs to produce non-interchangeable, partition-specific versions of bitstreams.

Reconfiguring at Runtime. At runtime, the reconfiguration of a reconfigurable partition can be initiated from outside the FPGA (e.g., an external CPU), by logic on the fabric, or by an embedded processor. The set of partial bistreams compiled at design time constitutes a module library that can be stored in on-chip or off-chip memory at system boot-up (e.g., from BRAM, DRAM or flash). To reconfigure an RP, the module is paused (all incoming and outgoing traffic is stopped); the incoming partial bitstream is loaded from the storage medium and passes through a PR interface to reprogram the PR region; finally, the new module is started. A PR interface is essentially a gateway to the configuration memory cells. Different PR interfaces are available on Xilinx Zynq FPGAs such as the processor configuration access port (PCAP) or the internal configuration access port (ICAP) depending on whether the reconfiguration process is triggered from a hard

embedded processor or from logic on the fabric, respectively. On Intel Stratix 10 FPGAs, reconfigurations occur through the secure device manager (SDM). Note that, in this work, our PR-style designs are deployed on Zynq SoC FPGAs with embedded ARM cores. PR is managed by software running on the ARM core and bitstreams are held initially in flash, and loaded into DRAM for use at system boot-up.

While one reconfigurable partition is undergoing reconfiguration, the logic on the rest of the fabric is not affected except the portions that interact directly with the reconfigurable partition's input/output ports. The disruption during PR must be accounted for explicitly by the enclosing design with the help of auxiliary status signals that indicate the readiness of the PR module. The minimum time to reconfigure a reconfigurable partition is on the order of milliseconds. The total time is a function of the size of the loaded bitstream. *For standard PR (uncompressed bitstreams), the bitstream size is a function of the reconfigurable partition size regardless of the actual degree of resource utilization within.*

In summary, building an efficient PR-style design requires to find (1) a good partitioning on the FPGA at design time, that is, decide on the number, the size and the connectivity between PR regions. Note that it is not necessary that all the reconfigurable partitions be the same size. For example, if the module workload mix is known ahead of time, one could improve mappability by creating asymmetrically resourced reconfigurable partitions tuned to the module workload at design time. For example, one would want to allocate reconfigurable partitions to be large enough for the largest required module or combination of modules. In other words, a PR-style design can consist of few large PR regions, many small PR regions, or a mix of small and large PR regions depending on the use-case. (2) a good allocation and execution strategy to allocate modules to PR regions and an execution time-slot, respectively. The module-to-RP and time-slot allocation can be managed by an external processor, an embedded processor, or logic on the fabric. The allocation strategy can be decided offline or online depending on the use-case.

Chapter 3

The incentives for using PR

This chapter first presents the terminology used in the remainder of the thesis. We then further elaborate on the concept of slack which we find to be the most important feature to have for an application to benefit from PR. Finally, we introduce a simplified analytical model and the area-time volume representation of an FPGA to explain the intuitions behind how PR can reduce slack and improve a design's performance or area. The next chapter continues with a more complete examination of the model.

3.1 Terminology

Application and Task Model. We refer to an application as a set of independent tasks. A task is modeled as a directed graph where nodes are processing stages, and edges represent stage dependencies. Each processing stage is accelerated on the FPGA by a micro or a macro hardware module that includes the necessary FPGA logic, hardened compute blocks and memory resources. Micro hardware modules accelerate processing stages that typically have low or medium compute complexity including linear algebra computations on (1) single data elements (e.g., scaling, threshold operations), (2) a small neighborhood of data elements (e.g., 2D stencil-based operations such as 2D convolutions) and (3) a large region of data elements (e.g., matrix-matrix multiplications). Macro modules are built with many micro modules and accelerate multiple processing stages at a time or even an entire task (e.g., stereo vision or optical flow computation). More examples of micro and macro modules are given in Section 5.3.4 of Chapter 5. In the remainder of the thesis, we do not make a distinction between micro or macro hardware modules, and generally refer to a micro or a macro module as a hardware module. A module can be reconfigured in a PR

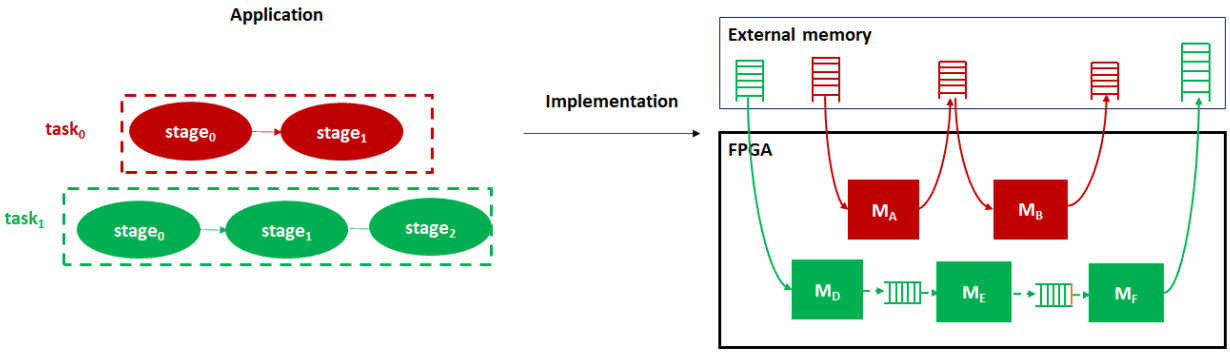

Figure 3.1: Example of an application with two independent tasks task_0 and task_1 accelerated on an FPGA. task_0 has two stages accelerated by dependent modules that share data through external memory .task_1 has three stages accelerated by dependent modules that share data through on-chip connections.

region as long as it has the same set of I/O ports and uses less resources than available in the PR region.

Modules have multiple Pareto-Optimal implementation variants with different performance-area trade-offs, e.g., the larger the variant, the faster it is. Figure 3.1 shows an example of an application with two independent tasks task_0 and task_1 accelerated on the FPGA. task_0 has two processing stages and task_1 has three processing stages. Dependent modules in task_0 share data through external memory while dependent modules in task_1 share data through on-chip memory connections.

ASIC-Style and PR-Style Designs. When we say that an application is accelerated on the FPGA, we mean that all tasks supported by the application are accelerated on the FPGA. We refer to the FPGA implementation of the application as an FPGA design. When we say that an application is mapped statically, we mean that the design does not use PR, and we refer to it as an ASIC-style design. On the other hand, when using PR to implement the application, the design is referred to as a PR-style design. In a PR-style design, the fabric is divided into multiple PR regions or reconfigurable partitions (RP). The time to reconfigure a PR region is referred to as reconfiguration time or PR time.

3.2 Reducing Slack for Design Optimization

3.2.1 Slack: Inefficiency in ASIC-Style Designs

In our work, we found that applications that most benefit from PR are the ones for which the ASIC-style implementation has slack. Slack occurs when logic resources occupied by the ASIC-style design are not

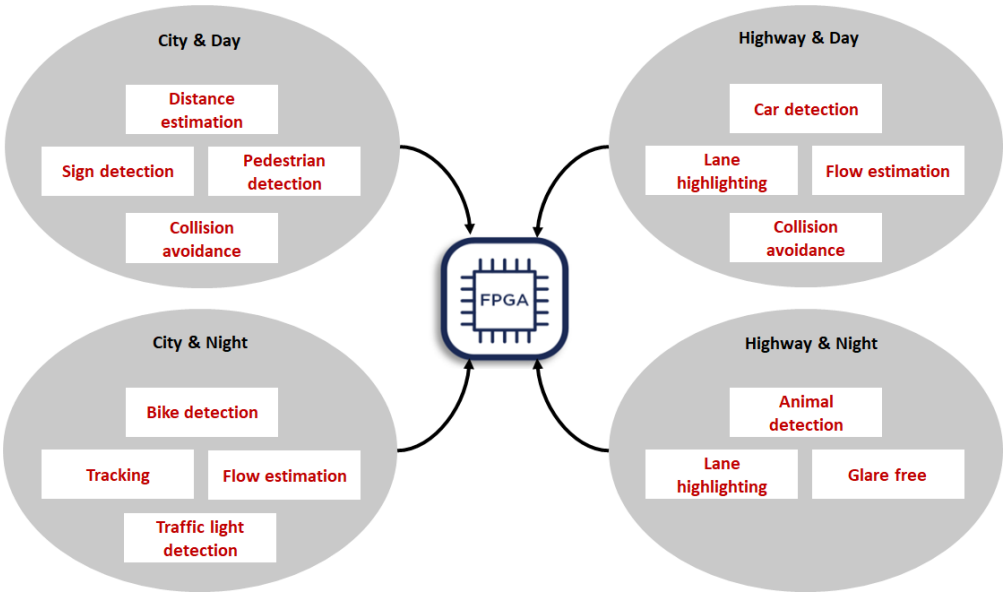

Figure 3.2: Example of an interactive application with slack deployed on an automotive system. In an ASIC-style design, some of the occupied resources are not active all the time since only a subset of tasks (tasks are shown in red) is requested by an user at a time depending on the external environment (city or highway & day or night).

actively utilized 100% of the time. Slack stems from hard-to-avoid sources of inefficiencies (e.g., operation dependencies, pipeline imbalancing) and can therefore be considered as inefficiency in a design. An example of an application with slack is shown in Figure 3.2. In this interactive application deployed on an automotive system, many tasks need to be accelerated on the FPGA but not of all them are needed at the same time. The subset of tasks needed at a given time is requested by the user depending on the environment (city or highway & day or night). Therefore, when mapping this application on the FPGA in an ASIC-style fashion, not all logic resources occupied by the design are actively utilized all the time. Instead of mapping this application statically on a possibly large FPGA, tasks could be temporally shared on a smaller FPGA with PR resulting in a reduction of device cost, and potentially, power and energy. In this work, we mostly focus on using PR for area, device cost, power or energy reduction. Another option is to use PR to execute more tasks than could be statically mapped on a given FPGA size (by swapping inactive tasks with active ones).

In the rest of this section, we use another simplified and idealized example to develop the intuitions behind how a PR-style design can either be (1) smaller while running at the same performance or (2) faster using the same area compared to an ASIC-style design with slack. Chapter 4 will present a more complete

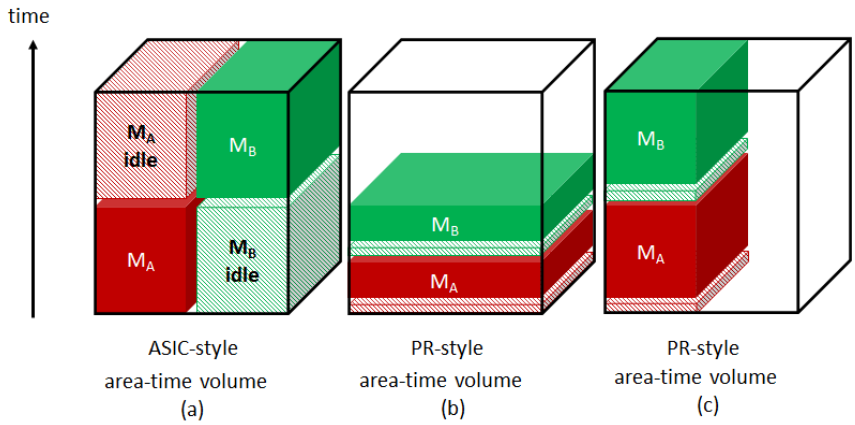

Figure 3.3: In an ASIC-style design, logic resources that are inactive must still occupy the fabric. In a PR-style design, slack can be reduced with better area-time scheduling.

model.

3.2.2 Partial Reconfiguration to Improve Performance or Area

Simplified Execution Model. We consider an application with a single task that has two dependent stages, stage_A and stage_B. Each stage is accelerated by a module for which multiple implementation variants exist. Each implementation variant is characterized by the latency function $Lat_i()$. $Lat_i(a)$ is the latency achieved by the module variant for stage_i using a logic resources. For a given stage_i, larger variants have lower latency i.e., $Lat_i(a)<Lat_i(b)$ if $a > b$. stage_B can start only after stage_A is finished. Each stage runs once per execution of the application. The latency of the application is the sum of the two dependent stages' latencies.

ASIC-Style Design. Consider two common design objectives: (1) minimize latency given an area budget, or (2) minimize area given a latency upper bound. For simplicity, assume $Lat_A(a)=Lat_B(a)$ for any a. In that case, to achieve optimality in either objective, the total logic resources, A_{total}, must be equally divided between stage_A and stage_B's modules ($A_A=A_B=0.5A_{\text{total}}$). The latency of the application is $2Lat_{A/B}(0.5A_{\text{total}})$. Solving either optimization scenarios repeatedly for different latency or area targets will produce a set of ASIC-style implementations that trade off latency against logic resources. Starting from this, we ask the question: can a PR-style design improve over the Pareto front of an ASIC-style design?

PR-Style Design. The above scenario for the ASIC-style design is shown in Figure 3.3.a. In this area-time volume representation of the FPGA, the fabric area is 100% occupied by the modules for stage_A and stage_B.

However, due to the dependency between the two modules, only one of the two modules is active at a time. In other words, the ASIC-style design has slack since some resources available to the design are not active all the time. In the area-time volume view, we see there are slack volumes when a module is idle i.e. when resources are occupied but not used.

In contrast to an ASIC-style design where resource allocation cannot change over time, it is possible to reduce slack with better area-time scheduling in a PR-style design. Therefore, a PR-style design may be able to achieve a smaller area-time volume by being faster, by using fewer resources, or both. For instance, to minimize latency given the same area budget, we can allocate the entirety of A_{total} to a module for stage_A first and then to stage_B (Figure 3.3.b). By doing so, the PR-style design's latency is reduced as both modules now run faster using all the resources available. On the other hand, a PR-style design can maintain the same latency using half the resources by allocating $0.5A_{\text{total}}$ to a module for stage_A first and then to stage_B (Figure 3.3.c). With slack reduced, both PR-style designs fit into smaller area-time volumes than the ASIC-style design. Notice in Figure 3.3.b and Figure 3.3.c, a small amount of slack appears when switching between modules to reflect the non-zero delay to perform PR. The full model in Chapter 4 will account for the effect of PR time.

Other Forms of Slack. In the area-time volume representation, slack represents inefficiency. In ASIC-style designs, slack stemming from module dependencies cannot be eliminated without changing the initial algorithm or implementation; it is a consequence of statically allocating resources. We find that reducing slack is the most important opportunity for improvement available to PR-style designs. Slack can arise in other forms in ASIC-style designs.

In our simplified example, we assume that module variants exist for any amount of resources. In practice, module variants for a stage only exist at certain performance/resource combinations. The modules selected to fit an area budget in an optimal ASIC-style design may not sum up perfectly to use all the resources. Further, when the modules of stage_A and stage_B are executed in a pipelined fashion to improve the throughput of many independent executions, it may not be possible to always find equal throughput variants for the two stages; in the resulting unbalanced pipelines, a too-fast stage has to stop or slow down to wait for the other stage. A more subtle example of slack exists when implementing a generic engine capable of accelerating different algorithms or neural networks. This generalized engine consists of a superset of features to accommodate all possibilities but only a subset of features is needed at a time (e.g., NPU [27], DPU [95]). A PR-style design could potentially reduce this type of slack (only the variant needed at a time occupies the fabric).

Chapter 4

Analytical Approach to PR

This chapter discusses different PR execution strategies to design area-efficient PR-style designs and presents our complete analytical model briefly introduced in the previous chapter.

4.1 Overview

Optimization Goals. To derive our performance model, we consider the problem of maximizing an application's performance given an area budget.

- *minimize the application's latency given an area budget A.* We label this problem as **min L given A**.
- *maximize the application's throughput given an area budget A.* We label this problem as **max T given A**.

Execution Model. Table 4.1 lists all the parameters used in our model. In this chapter, an application consists of a single task with N dependent stages; each stage is accelerated by a module. Dependent modules share data either through external or on-chip memory depending on data size. Though our discussion focuses on tasks with dependent stages, our model also applies if stages are independent. A single start-to-finish execution of a module is referred to as a run. If a task requires multiple independent runs, modules can execute concurrently. Figure 4.1 illustrates this execution model. The example application consists of three dependent stages stage_A, stage_B, and stage_C accelerated by three modules. In this application, each module needs to complete three runs R_0, R_1, and R_2. Modules execute concurrently to complete the runs as quickly as possible, subject to the dependency constraints.

Table 4.1: Model parameters.

A	Area budget
N	Number of stages in a task
P	Performance
L	Latency
T	Throughput
B	Batch size
R	Execution run
M	Module
a_i	Area of module variant accelerating stage_i
$Lat_i()$	Latency of module variant accelerating stage_i
$Tput_i()$	Throughput of module variant accelerating stage_i
$Time_{PR}()$	Time to reconfigure a PR region
F	Scaling factor
BW_i	Memory bandwidth requirement of the module variant that accelerates stage_i
BW_{total}	Total memory bandwidth available

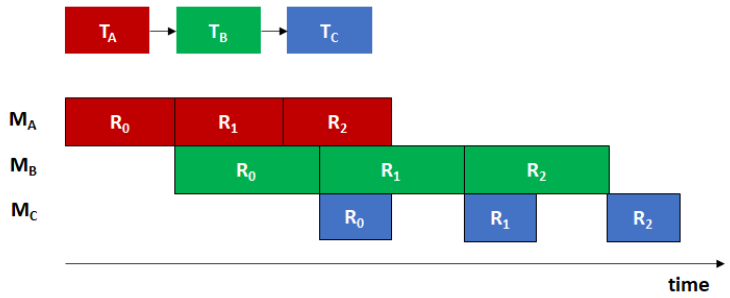

Figure 4.1: Example timeline of an application with three dependent stages accelerated by modules M_A, M_B, and M_C.

We consider two performance metrics, latency and throughput. Latency is defined as the start-to-finish time required for all modules accelerating a task to complete one run (including I/O time for data read and write and compute time). Throughput is defined as the number of runs completed per unit time in steady-state.

Performance-Area Trade-offs. For each module, a finite set of implementation variants exists. A variant accelerating stage_i is characterized by its area a_i, its latency $Lat_i(a_i)$, and its throughput $Tput_i(a_i)$ as functions of area. We assume that Lat_* and $Tput_*$ are monotonically increasing functions but make no further assumption on their shape, e.g., performance can scale sub-linearly or linearly with area.

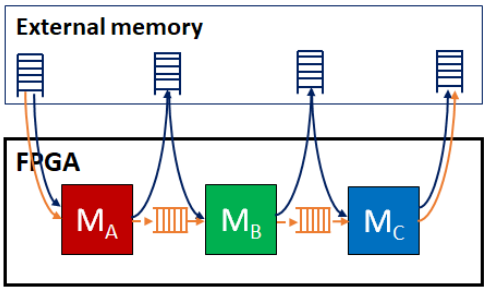

Figure 4.2: In an ASIC-style design, dependent modules share data through either external (blue) or on-chip memory (orange) depending on data size.

PR-Style Design Considerations. We define $Time_{PR}(a)$ as the time to reconfigure a PR region of size a, and assume that PR time is proportional to the PR region size.

4.2 ASIC-Style

We first derive the equations for the ASIC-style design that are applicable whether dependent modules share data through external or on-chip memory (Figure 4.2). In both cases, the number of buffers required to hold intermediate data is $N + 1$. In all equations, we define I as the set of subscripts for stages in the task.

Min L Given A. Let $Lat_{\text{Asic}}(A)$ be the latency of the ASIC-style design given A resources.

$$Lat_{\text{Asic}}(A) = \sum_{i \in I} Lat_i(a_i), \sum_{i \in I} a_i \leq A \tag{4.1}$$

Max T Given A. Let $Tput_{\text{Asic}}(A)$ be the throughput of the ASIC-style design given A resources.

$$Tput_{\text{Asic}}(A) = \min(\{\, Tput_i(a_i) \mid i \in I\}), \sum_{i \in I} a_i \leq A \tag{4.2}$$

4.3 PR-Style

4.3.1 Ignoring PR Time: PR-Style Performance Bounds

Ignoring PR time, we first derive the lower and upper bounds on the latency and throughput, respectively, achievable by any PR-style design presented in the next subsections. The simplest and most efficient execution strategy is to schedule stages serially on one PR region. Each module runs once before the PR region is reconfigured with the next module. In the best-case scenario, the PR region is of size A and the highest performance variant using A resources exists for all modules.

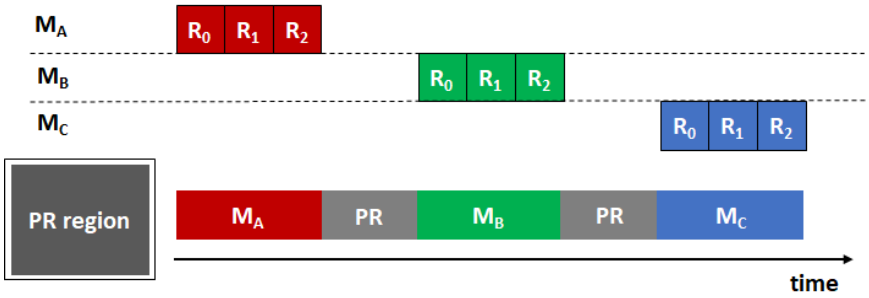

Figure 4.3: Example of serialized execution in a PR-style design with one PR region when batching ($B = 3$).

Min L Given A. Let $Lat_{\mathsf{PR},1,\mathsf{min}}(A)$ be the lower bound on latency for the PR-style design with one PR region.

$$Lat_{\mathsf{PR},1,\mathsf{min}}(A) = \sum_{i \in I} Lat_i(A) \tag{4.3}$$

Max T Given A. Let $Tput_{\mathsf{PR},1,\mathsf{max}}(A)$ be the upper bound on throughput for the PR-style design with one PR region.

$$Tput_{\mathsf{PR},1,\mathsf{max}}(A) = \frac{1}{\sum_{i \in I} \frac{1}{Tput_i(A)}} \tag{4.4}$$

4.3.2 Including PR Time: Serialized Execution on one PR Region

When accounting for PR time and scheduling stages serially on one PR region, each module runs once before the PR region is reconfigured with the next module. Given N stages, the PR region is reconfigured N times. Compute and reconfigurations are serialized.

Min L Given A. Let $Lat_{\mathsf{PR},1}(A)$ be the latency of the PR-style design with one PR region.

$$Lat_{\mathsf{PR},1}(A) = \sum_{i \in I} Lat_i(a_i) + N \times Time_{PR}(A) \tag{4.5}$$

Scheduling stages serially on one PR region of the largest size may not result in the design's minimum latency. Though using larger variants leads to a decrease in compute time, it also has the effect of increasing PR time, which may offset the speedup benefit of larger variants. In the next subsection, we discuss a scheduling alternative where compute and reconfigurations are overlapped.

Max T Given A: Batching to Amortize PR Time. Let $Tput_{\mathsf{PR},1}(A)$ be the steady-state throughput of the PR-style design with one PR region.

$$Tput_{\mathsf{PR},1}(A) = \frac{1}{\sum_{i \in I} \frac{1}{Tput_i(a_i)} + N \times Time_{PR}(A)} \tag{4.6}$$

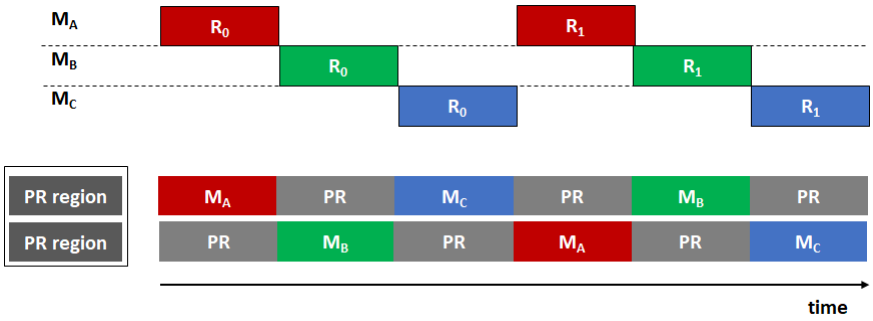

Figure 4.4: Interleaved execution on two PR regions. PR time can be hidden by overlapping compute and reconfiguration.

If PR time is non-trivial compared to compute time, we can amortize PR time by executing each module B times (i.e. batching B runs) before reconfiguring the PR region (Figure 4.3). Let $Tput_{\mathsf{PR},1}(A)$ be the steady-state throughput of the PR design with one PR region when batching runs.

$$Tput_{\mathsf{PR},1}(A) = \frac{B}{\sum_{i \in I} \frac{B}{Tput_i(a_i)} + N \times Time_{PR}(A)} \tag{4.7}$$

Batching allows us to reduce the ratio of total PR time to total compute time at a greater resource cost to buffer intermediate results. Given enough buffering capacity, PR time can be almost totally amortized for large enough B.

4.3.3 Including PR Time: Special Cases

Min L Given A: Interleaved Execution on Two PR regions. When optimizing for latency, interleaving stage execution on multiple PR regions allows us to overlap reconfigurations and compute to hide PR time, which may result in better latency than serializing stage execution on one PR region. Figure 4.4 shows an example of interleaved execution for $k = 2$. In this example, $Time_{PR}(A/2) = Lat_i(a_i), \forall i \in I$. By overlapping compute and reconfigurations, PR time is completely hidden. Having $k > 2$ may be beneficial provided that multiple PR regions can be reconfigured simultaneously. Simultaneous reconfiguration of multiple PR regions is not supported from a user standpoint using current FPGA tools and PR flow. In this work, we only consider the case where $k = 2$, and define $Lat_{\mathsf{PR},2}(A)$ as the latency of the PR-style design with two PR regions.

$$Lat_{\mathsf{PR},2}(A) = \sum_{i \in I} \max(Time_{PR}(A/2), Lat_i(a_i)) \tag{4.8}$$

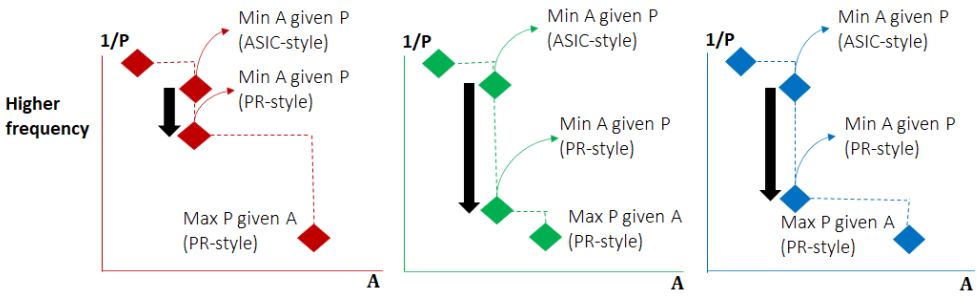

Figure 4.5: The strategy for choosing module variants depends on the design objective: (1) maximize performance (P) given an area (A) budget or (2) minimize area given a performance bound. Example for the three-stage task accelerated by modules M_A (red), M_B (green), and M_C (blue) shown in Figure 4.2.

Max T Given A: Serialized Execution on k PR regions. When optimizing for throughput, it is generally preferable to choose the smallest k to reduce a design's complexity in terms of buffering management since each PR region requires its own intermediate buffer. A k-PR region solution should be considered when appropriately large module variants are not available for all modules in a single PR region solution.

When having multiple PR regions executing in parallel (similar to k-way SIMD), stage execution can be serialized on each PR region of size A/k. On each PR region, each module runs once or multiple times before the PR region is reconfigured. Let $Tput_{\mathsf{PR,1}}(A/k)$ be the steady-state throughput of a single PR region of size A/k and $Tput_{\mathsf{PR,k}}(A)$ be the steady-state throughput of the PR-style design with k PR regions. Assuming that k reconfigurations can occur simultaneously,

$$Tput_{\mathsf{PR,k}}(A) = k \times Tput_{\mathsf{PR,1}}(A/k) \tag{4.9}$$

As explained previously, only one reconfiguration can happen at a time using current tools. The above throughput can still be achieved by offsetting the start of compute on each PR region by a sufficient number of PR times to ensure that two PR regions are not reconfigured simultaneously.

4.3.4 PR Strategies to Minimize Area Given a Performance Bound

Serialized Execution on one Small PR Region. To derive the model, this chapter focuses on the problem of maximizing performance given an area budget. Another important design goal that we investigate in this work is to minimize area given a performance bound. This latter objective is most relevant for applications that are primarily concerned with efficiency rather than maximum performance (e.g., video

analytics, image processing applications). Depending on the design objective targeted, the strategy for picking module variants differs. In the case of maximizing performance given an area budget, the best strategy is to schedule tasks serially on one large PR region and pick the largest module variants available. In the case of minimizing area given a performance bound, the best strategy is to schedule tasks serially on a small PR region and pick the *smallest module variants available*. Figure 4.5 illustrates these two strategies using the example application shown in Figure 4.2.

Slack From Higher Clock Frequency Than Needed. When minimizing area given a performance bound, the best-effort ASIC-style design uses the smallest module variants operating at the required frequency to achieve the desired performance. The ASIC-style design's operating frequency may be less than the maximum achievable frequency. If the ASIC-style ran faster than needed (i.e. if it was clocked at the maximum achievable frequency), the design would have slack since occupied resources would not be active all the time. This form of slack can be leveraged by a PR-style design that should use the smallest module variants operating at a higher frequency than the ASIC-style design to compensate for PR time. If the PR-style design does not run at the desired performance using the smallest variants operating at the maximum clock frequency, the variants' size should be increased incrementally until the target performance is achieved. Note that the same latency and throughput equations presented in the previous sections of this chapter can be used for the problem of minimizing area given a performance bound. The techniques discussed to amortize or hide reconfiguration time also remain relevant.

Example Applications. Some examples of real-world applications in which the objective is to meet a performance target subject to area, power or energy constraints include video analytics, interactive and robotic applications. In these applications, data is produced by an input source running at a fixed rate, and the processing pipeline runs at a rate equal or greater than the input source. For instance, in a video analytics application, the input source is a camera, and the processing pipeline can operate at a rate equal or greater than the camera's rate. In a robotic application, a robot needs to process events in a given time frame, and not necessarily as fast as possible, i.e. there is a latency upper bound to process an event. In these examples, the objective is not to maximize performance but rather to minimize metrics such as area, cost, power or energy while meeting latency or throughput requirements. Chapter 5 discusses in greater details the practical challenges and benefits of using PR for these types of applications.

4.3.5 Memory Requirements in PR-style designs

In this section, we discuss the buffering and memory bandwidth requirements of a PR-style design. Compared to an ASIC-style design, a PR-style design requires additional buffering capacity for batching and additional external memory bandwidth when faster module variants are used. A module variant is faster if it uses more resources and/or operates at a higher clock frequency. For the **Max T Given A** problem, we also model the impact of limited memory bandwidth on throughput.

Buffering Requirement. In a PR-style design, each PR region requires two intermediate buffers to hold its intermediate input and output data. The intermediate buffers can be stored in on-chip or off-chip memory depending on the data size. The on-chip buffering option is preferred to minimize the latency and power/energy for data movement. In practice, when batching to amortize reconfiguration time, the buffering capacity required by a PR-style design exceeds the amount of on-chip memory available on current FPGAs (few MBs on large FPGAs). The amount of data to buffer can range from tens to hundreds of MBs depending on the use-case. We quantify this surplus in the next section. Chapter 7 discusses the additional power/energy overhead in a PR-style design due to additional data movement to/from on-board external memory.

If the intermediate buffers are stored in on-chip memory, additional architecture support is needed so that the output of the upstream module stored on chip is used as the input to the next module. One possible solution is to design an intermediate on-chip memory controller to connect the PR region to the intermediate buffers instead of having static, direct connections between the PR region and the buffers. The on-chip memory controller fetches the data from the appropriate intermediate buffer to send to the PR region, and writes the output from the PR region to the appropriate buffer.

Max T Given A: Memory Bandwidth Requirement. When maximizing throughput given an area budget, the best strategy is to serialize module execution on one PR region. An upper bound on the memory bandwidth required by the PR-style design can be determined by considering the read and write bandwidth required by the fastest variant in the design i.e. the variant with the highest throughput.

When the memory bandwidth required by the variant is greater than the total memory bandwidth available in the system, the variant throughput is going to be degraded by some factor proportional to the memory bandwidth required. We introduce a scaling factor F to model the impact of limited memory bandwidth on a variant's throughput. F is equal to the ratio of memory bandwidth required by the variant to the memory bandwidth available in the system if the bandwidth required by the variant is greater than

Table 4.2: Resource utilization of the two PR-style designs $\mathbf{P_1}$ and $\mathbf{P_2}$ post place & route on the Ultra96 v2 board at 150 MHz. In both designs, most resources are spent for compute. In $\mathbf{P_2}$, the PR regions are almost equally-sized.

		$\mathbf{P_1}$	**(1 PR region)**		$\mathbf{P_2}$	**(2 PR regions)**	
	I/O infrastructure	**PR region**	**Total**	**I/O infrastructure**	**PR region 0**	**PR region 1**	**Total**
LUT	3366 (4.8%)	61,920 (87.8%)	65,286 (92.5%)	5231 (7.4%)	28,800 (40.8%)	30,240 (42.9 %)	64,271 (91%)
BRAM36Kb	0	198 (91.7%)	198 (91.7%)	0	108 (50%)	108 (50%)	216 (100%)
DSP	0	288 (80%)	288 (80%)	0	144 (40%)	216 (60%)	360 (100%)
PR time (ms)	N/A	12	N/A	N/A	6	6	N/A

the bandwidth available. Otherwise, F is equal to 1. Let $Tput_{i,peak}(a_i)$ be the peak throughput of the module variant that accelerates stage_i, BW_i the bandwidth requirement of the variant, and BW_{total} the total bandwidth available in the system.

$$Tput_i(a_i) = F \times Tput_{i,peak}(a_i), F = \begin{cases} BW_i/BW_{total}, & \text{if } BW_i > BW_{total} \\ 1, & \text{otherwise} \end{cases} \tag{4.10}$$

We can replace this expression of $Tput_i(a_i)$ in the throughput equations presented in this section.

4.4 Evaluation

4.4.1 Experimental Setup

We develop three compute-bound applications representative of real-world applications with cost constraints [36,54,79]. For all studies, we use a low-end FPGA board (Ultra96 v2) with a XC7ZU3EG Zynq part that has 70,560 LUTs, 216 BRAMs and 360 DSPs. These studies serve as concrete examples of ASIC-style designs with slack (due to module dependencies or modules having mismatched throughput). Each application consists of three dependent stages, with some stages being more compute intensive than others, which perform common vision processing such as detection or classification. Dependent modules share data through external memory since the amount of on-chip memory on the Ultra96 is not sufficient to hold the inter-module buffers in on-chip memory. Note that having more stages per application would favor PR-style designs, since the length of the dependency chain would increase. In other words, we choose to focus on more challenging design scenarios (shorter pipelines).

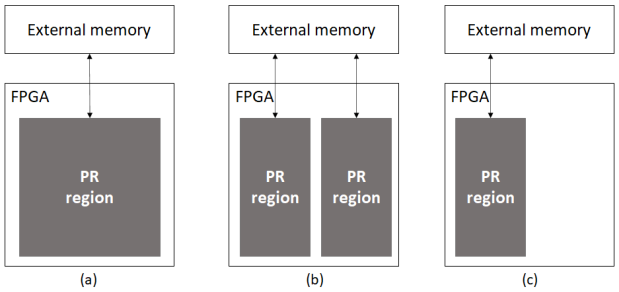

Figure 4.6: Three PR-style designs are considered in these studies: (a) $\mathbf{P_1}$ with a single large PR region on which stages are scheduled sequentially, (b) $\mathbf{P_2}$ with two almost equally-sized PR regions on which stages are executed in an interleaved fashion, and (c) $\mathbf{P_{1,s}}$ with a single smaller PR region (one PR region of $\mathbf{P_2}$) on which stages are scheduled sequentially.

Design Scenario. In the studies, we solve the **max T given A** and **min L given A** problems from Section 4, and also consider the problem of minimizing area given a latency upper bound, which we refer to as **given L min A**. Using our model, we search the design space to find the best-achievable ASIC-style and PR-style designs for a given problem. The best-achievable design consists of the set of module variants resulting in the design's maximum throughput, minimum latency or minimum area possible given the module variants available. We use Vivado 2019.1 to build our designs [94].

PR-Style Designs. Figure 4.6 shows the three PR-style designs we consider in these studies: (1) $\mathbf{P_1}$ with a single large PR region on which stages are scheduled sequentially, (2) $\mathbf{P_2}$ with two almost equally-sized PR regions on which stages are executed in an interleaved fashion, and (3) $\mathbf{P_{1,s}}$ with a single smaller PR region (one PR region of $\mathbf{P_2}$) on which stages are scheduled sequentially. Table 4.2 reports the resource utilization of $\mathbf{P_1}$ and $\mathbf{P_2}$ (the PR region of $\mathbf{P_{1,s}}$ has the same size as PR region 1 of $\mathbf{P_2}$) on the Ultra96 v2 board at 150 MHz. In both designs, most resources on the Ultra96 v2 are used for compute. The time to reconfigure a PR region through the processor configuration access port (PCAP) when partial bitstreams are stored in external DDR is 12 ms (partial bitstreams of 5.5 MB for $\mathbf{P_1}$) and 6 ms (partial bitstreams of 2.8 MB for $\mathbf{P_2}$). We use one ARM core to manage the operation of the fabric at runtime (i.e. reconfiguration of the PR regions and module execution). PR bitstreams are stored into on-board external DDR.

When optimizing for latency, we report the latency of $\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$ whenever possible. We refer to latency (or frame latency) as the time to process one input frame by the application, i.e. the time it takes for each module to run once. When optimizing for throughput, we report the throughput of $\mathbf{P_1}$ for different batch sizes B. In the context of our studies, the input to an application is a frame. When $B > 1$, the module

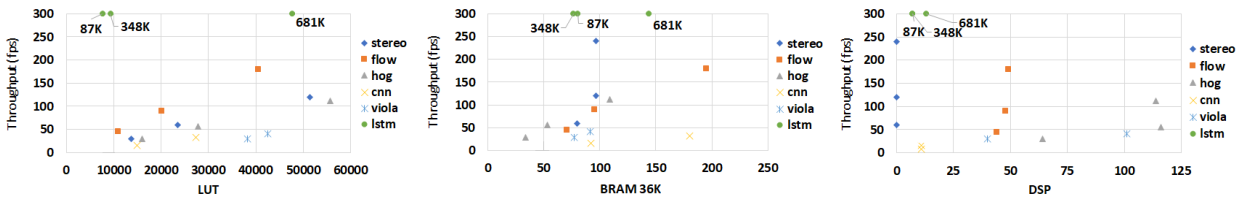

Figure 4.7: Module variant throughput vs area (LUT, BRAM36k and DSP) post place & route on the Ultra96 v2 board at 150 MHz for the six modules used in the studies. Each module has up to three variants.

processes B frames before the PR region is reconfigured.

Performance Density. In addition to latency and throughput, we also compare the performance density of ASIC-style and PR-style designs. Performance density is defined as the number of frames processed per unit time per unit area. This metric quantifies how efficiently a design utilizes available resources. The higher the performance density, the more area-efficient the design is (less slack in the area-time volume). Since there is no simple definition for area on an FPGA, we consider the resources used by the bottleneck resource as a proxy for area. For instance, if BRAM is the bottleneck as it is the case in our studies, performance density is computed as the number of frames processed per unit time per BRAM. For latency, we divide 1/latency by the number of BRAM used in the design. For throughput, we simply divide throughput by the number of BRAM used in the design.

Module Characterization. In the studies, we use six modules: hog [46], cnn [33], lstm [69], viola [98], flow [17], and stereo (developed in-house). Each module has up to three implementation variants generated with Vivado HLS 2019.1 [92] (Figure 4.7). The variants are provided by the module developer or obtained by changing parameters in the HLS source code, such as the number of compute engines, the data precision, and the on-chip buffering size. The modules' interfaces are modified to conform to our PR region interfaces. In our studies, all PR regions have the same interfaces, namely, one AXI memory-mapped, one AXI-lite, a clock, a reset, and an interrupt. All data transfers, including data sharing between modules in the ASIC-style design, happen through external DRAM.

Modules operate on 256×256 frames, except for the lstm module which operates on 32×32 frames. Modules process one frame at a time. Therefore, frame latency is the inverse of throughput, and includes both compute and data movement time. Data movement accounts for no more than 15% of the end-to-end latency. For all variants, module throughput scales mostly linearly with its resources. The bottleneck resource for all modules is either LUTs or BRAM on the Ultra96 v2.

Table 4.3: Resource utilization, throughput and frame latency of the variants used in $\mathbf{P_1}$.

	hog	**cnn**	**lstm**	**stereo**	**flow**	**viola**
LUT	55,635	27,573	47,745	51477	40,509	42,283
BRAM36Kb	109	180	144	96.5	195	91.5
DSP	114	11	13	0	49	101
Throughput (fps)	116	32	2.1k	240	180	41.3
Frame latency (ms)	8.6	31.2	0.48	4.2	5.6	24.2

Table 4.4: Resource utilization and frame latency of the variants used in $\mathbf{P_2}$.

	hog	**cnn**	**lstm**	**stereo**	**flow**
LUT	27,879	15,009	7461	23,551	20,106
BRAM36Kb	53.5	92	80.5	96.5	95.5
DSP	114	11	13	0	48
Frame latency (ms)	17.9	62.5	0.87	8.3	11.1

4.4.2 Model Validation: Case Study Results

In this section, we illustrate how to use our model and validate its effectiveness in three case studies. We show that (1) our first-order model allows to accurately estimate a design's throughput and latency. (2) Our analysis helps determine the most suited PR execution strategy for a problem. Notably, when optimizing for latency, it is important to evaluate both PR execution strategies (serialized execution on one PR region and interleaved execution on multiple PR regions) to find the best one for a given problem. (3) PR-style designs improve performance and performance density upon ASIC-style designs with slack. (4) Given an area budget, if the ASIC-style design is too big to fit, using PR can help make the design fit and run at useful performance.

Study 1: Activity Recognition. The first case study performs activity recognition and is based on [36]. Three dependent stages are accelerated by a hog, a cnn and a lstm modules. This study explores the **max T given A** and **min L given A** problems. In this study, we explain how to use our model for quick design space exploration. The same methodology is used for the two other studies.

Max T Given A. Table 4.5 shows the resource utilization and the throughput of the ASIC-style design and the module variants used. The ASIC-style design's throughput is equal to 16 fps and is limited by

Table 4.5: Resource utilization, average memory bandwidth, and throughput of the ASIC-style design and the module variants used post place & route on the Ultra96 v2 board at 150 MHz for the activity recognition study.

	Module variants			**ASIC-style**		
	hog	**cnn**	**lstm**	**I/O Infrastructure**	**Modules**	**Total**
LUT	15,495 (22%)	14,614 (20.7%)	7715 (10.9%)	6082 (8.6%)	37,824 (53.6%)	43,906 (62.2%)
BRAM36Kb	34 (15.7%)	92 (42.6%)	80.5 (37.3%)	0	206.5 (95.6%)	206.5 (95.6%)
DSP	64 (17.8%)	10 (2.8%)	7 (1.9%)	0	81 (23%)	81 (23%)
Memory bandwidth (MB/s)	23.6	42.7	3.3	N/A	N/A	69.6
Throughput (fps)	30	16	271	N/A	N/A	16

the throughput of the slowest module (cnn). The hog and lstm variants are roughly 2× and one order of magnitude faster than the cnn variant, respectively. The amount of computation per frame for the lstm variant is much less than the two other modules. Therefore, the ASIC-style design has slack, and there is opportunity for PR to improve.

Based on our analysis and on module variants available, batched execution on a single PR region solution ($\mathbf{P_1}$) should provide best performance. Figure 4.8 shows the estimated and measured throughput, and the intermediate buffering capacity required for $\mathbf{P_1}$ vs. batch size B. We use equation 4.7, measured throughput variants (Table 4.3) and PR time (Table 4.2) to compute these estimations. We observe that (1) as predicted by the model, when B increases, PR time gets amortized, but with diminishing return when $B \geq 32$. (2) For all B, the estimated and measured throughput match within 2.35%. (3) At $B = 64$, the throughput of the PR-style design is 24.7 fps, which represents a 54.4% improvement over the ASIC-style design. (4) Intermediate buffering capacity linearly increases with B, and is equal to 50.3 MB for $B = 64$. The intermediate buffers are stored in on-board external memory (on the Ultra96, 2 GB of external DDR is available). The external memory bandwidth (read and write) requirement for $\mathbf{P_1}$ is 91.2 MB/s due to the hog module. This represents a 31% increase over the ASIC-style design which needs on average 69.6 MB/s (Table 4.5).

The ASIC-style design uses 206.5 BRAMs (95.6% of BRAM resources) and has a performance density of 0.077 fps per BRAM. $\mathbf{P_1}$ uses 198 BRAMs (91.7% of BRAM resources available) and has a performance density of 0.12 fps per BRAM, which represents a 55.8% improvement over the ASIC-style design.

Min L Given A. When optimizing for latency, the ASIC-style design has slack since modules are dependent (one frame processed at a time), and therefore, we expect PR to be beneficial. Figure 4.9.activity shows the

Table 4.6: Resource utilization and latency of the ASIC-style design and module variants used post place & route on the Ultra96 v2 board at 150 MHz for the depth and motion estimation study.

	Module variants			**ASIC-style**		
	hog	**stereo**	**flow**	**I/O Infrastructure**	**Modules**	**Total**
LUT	27,244 (38.6%)	13,767 (19.5%)	10,943 (15.5%)	3366 (4.8%)	51,924 (73.6%)	55,320 (78.4%)
BRAM36Kb	52.5 (24.3%)	79.5 (36.8%)	70.5 (32.6%)	0	202.5 (93.8%)	202.5 (93.8%)
DSP	114 (31.7%)	0	44 (12.2%)	0	158 (43.9%)	158 (43.9%)
Frame latency (ms)	17.8	16.7	22.2	N/A	N/A	56.7

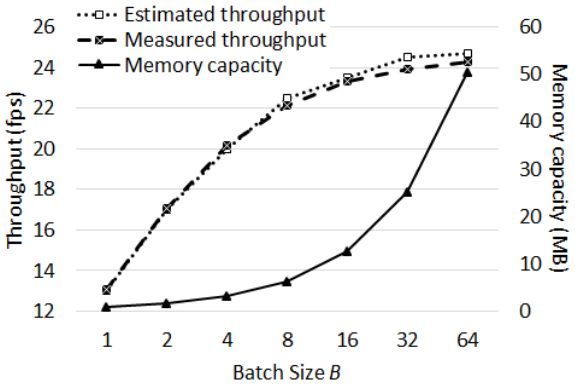

Figure 4.8: Throughput of $\mathbf{P_1}$ vs. B for the first case study.

frame latency of the latency-optimized ASIC-style design ($\mathbf{A_s}$), and the three PR-style designs ($\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$). We estimate the latency of $\mathbf{A_s}$ using equation 4.1 and measured module latencies (Table 4.5). The ASIC-style design has an estimated latency of 99.5 ms, which exactly matches our measurement.

We estimate the latencies of the PR-style designs using equations 4.5 and 4.8, measured latencies from Tables 4.3 and 4.4, and PR time from Table 4.2. The estimated latencies for $\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$ are 76.6 ms, 102 ms, and 92.4 ms, respectively. The measured latencies for $\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$ are 76.8 ms, 102.2 ms, and 92.6 ms, respectively. We observe that (1) estimated and measured latencies match within 0.26%, and (2) among the three PR-style designs, $\mathbf{P_1}$ has the smallest latency, as predicted by the model (22.8% improvement over the ASIC-style design). Note that PR time accounts for a non-negligible fraction of the frame latency of $\mathbf{P_1}$ (46.9%). However, $\mathbf{P_1}$ still outperforms $\mathbf{P_2}$, illustrating that the ratio of PR time to compute time should not be considered alone when optimizing for latency.

Considering performance density, $\mathbf{A_s}$ uses 206.5 BRAMs and has a performance density of 0.049 per-

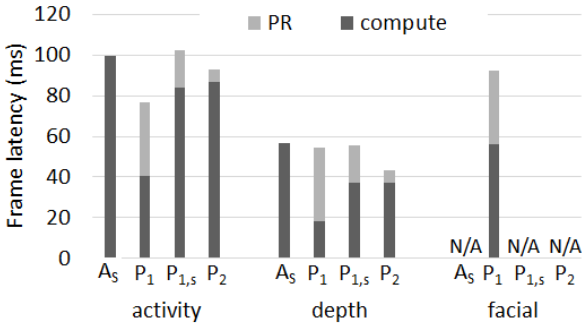

Figure 4.9: Frame latency of the ASIC-style design ($\mathbf{A_s}$) and the PR-style designs $\mathbf{P_1}$, $\mathbf{P_{1,s}}$ and $\mathbf{P_2}$ for the three studies. In the facial recognition study, the resources on the Ultra96 v2 are insufficient to map $\mathbf{A_s}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$.

seconds per BRAM. $\mathbf{P_1}$ uses 198 BRAM and has a performance density of 0.066 per-seconds per BRAM (34.7% improvement over ASIC-style).

Study 2: Depth and Motion Estimation. The second case study performs depth and motion estimation, and is based on [79]. Three dependent stages are accelerated by a hog, a stereo, and a flow module, respectively. This study explores the **min L given A** problem.

Figure 4.9.depth shows the frame latency of the latency-optimized ASIC-style design ($\mathbf{A_s}$), and the three PR-style designs ($\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$). We estimate the latency of $\mathbf{A_s}$ using equation 4.1 and module latencies from Table 4.6. The estimated latency of $\mathbf{A_s}$ is 56.7 ms (matches the measured latency). Using the same procedure described in the first case study, we obtain latency estimations for $\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$ of 54.4 ms, 55.3 ms, and 43.3 ms, respectively. The measured latencies for $\mathbf{A_s}$, $\mathbf{P_1}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$, are 56.7 ms, 54.4 ms, 55.3 ms, and 43.3 ms, respectively. We observe that (1) estimated and measured latencies match within 0.18%, and (2) among all PR-style designs, $\mathbf{P_2}$ has the lowest latency, as predicted by the model (23.6% improvement over the ASIC-style design), reinforcing the fact that using the largest variants available may not achieve minimum latency.

Considering performance density, $\mathbf{A_s}$ uses 202.5 BRAMs (93.8% of BRAM resources available) and has a performance density of 0.087 per-seconds per BRAM. $\mathbf{P_2}$ uses 216 BRAMs, and has a performance density of 0.11 per-seconds per BRAM (26.4% improvement over the ASIC-style design). Note that $\mathbf{P_{1,s}}$ uses only 108 BRAMs while achieving a 2.46% latency improvement compared to $\mathbf{A_s}$. $\mathbf{P_1}$ uses 2× more BRAM but only improves latency by 1.8% compared to $\mathbf{P_{1,s}}$. $\mathbf{P_{1,s}}$ has a performance density of 0.165 fps per BRAM (92.2% improvement over the ASIC-style design). In a design scenario where area is to be minimized given

a latency upper bound of 60 ms, $\mathbf{P_{1,s}}$ would be the best design choice.

Study 3: Facial Emotion Recognition. The final study performs facial emotion recognition, and is based on [54]. Three dependent stages are accelerated by a viola, a cnn and an lstm module, respectively. This study explores the **min L given A** and **given L min A** problems.

Min L Given A. The resources on the Ultra96 v2 are insufficient to map $\mathbf{A_s}$, $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$. In the case of $\mathbf{A_s}$, 243 BRAMs are needed to map the smallest variants for the viola, the cnn and lstm modules (only 216 BRAMs are available on the Ultra 96 v2 board). In the cases of $\mathbf{P_{1,s}}$, and $\mathbf{P_2}$, it is not possible to map the smallest variant available for the viola module since this variant consumes more LUT resources (35K LUTs) than available in the PR region of $\mathbf{P_{1,s}}$ and $\mathbf{P_2}$ (30k LUTs available). Figure 4.9.facial shows the frame latency of $\mathbf{P_1}$. Using the same procedure as in the first case study, we estimate the frame latency of $\mathbf{P_1}$ to be 92.2 ms. The measured latency is 92.1 ms (0.11% error). $\mathbf{P_1}$ uses 198 BRAMs and has a performance density of 0.055 per-seconds per BRAM. In summary, when the ASIC-style design is too big to fit, PR can make the design fit and achieve useful performance (less than 100 ms).

Given L Min A. Given a latency upper bound of 100 ms, we want to estimate the minimum area needed by an ASIC-style design to achieve this requirement. On a larger FPGA board (Ultrascale+ 102), the ASIC-style design consisting of the smallest module variants available uses $65,987$ LUTs, 249.5 BRAMs, and 56 DSPs, and achieves a latency of 100.2 ms post place & route at 150 MHz. The performance density of the ASIC-style design is 0.04 per-seconds per BRAM. Considering the PR-style design from **min L given A**, $\mathbf{P_1}$ improves latency by 8% and performance density by 27.3% compared to the ASIC-style design.

4.5 Summary

This chapter presents a set of PR execution strategies to build efficient PR-style designs that can (1) be faster given an area budget or (2) smaller given a performance bound than ASIC-style designs with slack. We discuss our first-order model to quickly and accurately estimate the relative merits of ASIC-style and PR-style designs in the early stage of design development. The model considers the trade-offs between PR region size, PR time and module performance. We also account for the impact of memory bandwidth requirements on module performance. Though limited, this choice of execution model and performance metrics allows us to cover a non-trivial range of design scenarios and applications (e.g., video analytics/image processing pipelines, feed-forward neural networks).

We validate our first-order model in three study applications that serve as practical examples of ASIC-

style designs with slack. Notably, we show that (1) our first-order model allows to accurately estimate a design's throughput and latency. (2) Our analysis helps determine the most suited PR execution strategy for a problem. (3) PR-style designs improve performance and performance density upon ASIC-style designs with slack. (4) Given an area budget, if the ASIC-style design is too big to fit, using PR can help make the design fit and run at useful performance. The model relies on the existence of a module library consisting of Pareto-optimal module variants used to build the ASIC-style and PR-style designs. The accuracy of the model depends on (1) how well the library has been characterized in terms of area, latency, throughput, and memory bandwidth requirement and (2) the ability to place and route modules at the required clock frequency, which can be challenging depending on the problem. The model could be improved to account for this clock frequency uncertainty, for instance, by defining different levels of confidence based on the design's complexity. In Chapter 9, we further elaborate on the limitations of the work presented in this chapter.

Chapter 5

Design with PR for Domain-Specific Acceleration

While the previous chapters offer a theoretical foundation to understand when, why and how PR can be beneficial compared to an ASIC-style approach for applications with a single task, this chapter discusses in greater details the practical challenges of using PR to accelerate applications with multiple tasks. Specifically, this chapter describes our investigation on using PR for computer vision applications where multiple tasks can be spatially shared and real-time time-shared on the fabric.

The rest of this chapter is organized as follows. Section 5.1 motivates the use of PR for accelerating vision applications and discusses the challenges when doing real-time time-sharing. In this usage mode, multiple vision pipelines can round-robin execute within the time-scale of a camera frame. Section 5.2 reviews the operation of a streaming vision pipeline on an FPGA. Section 5.3 presents the design of our framework. Section 5.4 presents a detailed evaluation of our framework where we show that multiple tasks can spatially share and real-time time-share the FPGA and achieve real-time performance (30+ fps).

5.1 Motivations and Challenges

In this section, we first explain why computer vision applications are good candidates for a PR approach. We then discuss the practical challenges and potential benefits of using PR at a fine-grain for computer vision acceleration. Notably, we investigate the feasibility of doing real-time time-sharing on today's FPGAs. In

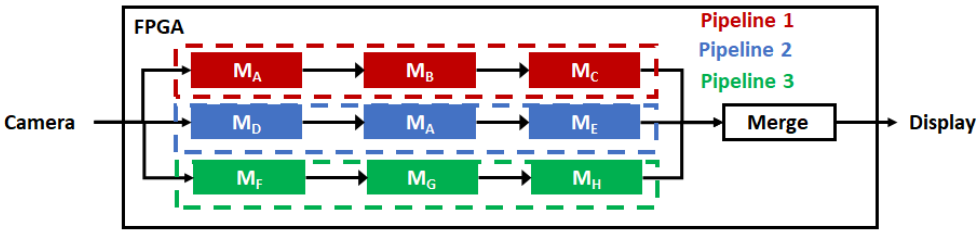

Figure 5.1: Example of a computer vision application with three independent tasks accelerated by streaming pipelines running simultaneously. Each pipeline needs to run at the same rate as the input camera.

this usage mode, multiple vision pipelines can round-robin execute within the time-scale of a camera frame.

Why PR for Computer Vision. Due to their flexibility, power and energy efficiency, FPGAs have been increasingly used to accelerate vision applications [9,12,35,86,96]. For the most part, vision applications are mapped statically when accelerated on an FPGA. However, an ASIC-style approach might fall short for a rapidly emerging class of vision applications [60,78] that are highly concerned about efficiency in terms of cost, power and energy, and not only raw performance (e.g., automotive, robotic or video analytics applications). These emerging application need to support many different tasks with real-time requirements, and have stringent area, cost, power or energy constraints. An example of such application was presented in Chapter 3 (Figure 3.2). The dynamic adaptation requirement of this interactive vision system leads to a potentially large number of tasks to execute at runtime. For all tasks to run at the desired performance, an ASIC-style design may require more resources than available on the target FPGA. Using a larger FPGA is not an option for these applications with very limited power or energy budget and/or cost constraint (e.g., robotic or smart SSD applications [49]). For such use-cases, a PR approach can provide a solution if (1) the ASIC-style has slack, and (2) the amount of slack is sufficient for a PR-style design to operate at the desired performance. Another potential advantage of using PR for these applications is to reduce the volume of data sent to a remote server or to the cloud. By accelerating as many tasks as possible on the FPGA, closer to the sensors at the Edge, the requirements in terms of bandwidth can be reduced and response time can be improved.

Slack in Computer Vision Applications. For most real-time vision applications that we target in this chapter, the ASIC-style design has slack since it can be clocked at a frequency lower than the maximum achievable frequency to meet the desired performance. If clocked at the maximum frequency, the ASIC-style design would run faster than needed, and therefore, occupied resources would not be active all the time. Figure 5.1 presents an example application to illustrate this point. In this application, three pipelines are

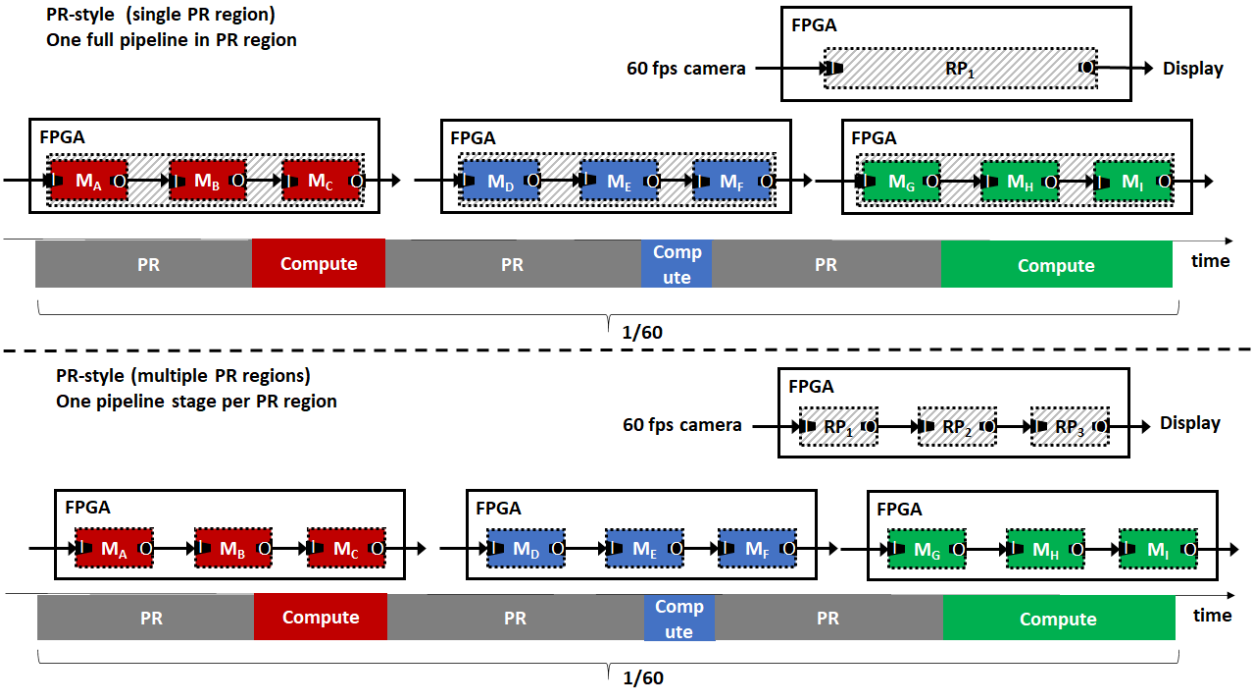

Figure 5.2: A single large PR region solution (top) or a solution with multiple smaller PR regions (bottom) would fail in many use-cases when doing real-time time-sharing on today's FPGAs.

mapped statically on the FPGA. We focus our explanation on the operation of a single pipeline since the two other pipelines operate similarly. A pipeline needs to process full HD frames (1920-by-1080 pixels per frame) produced by a HDMI camera at 60 fps, that is, a frame needs to be processed by the pipeline in less than 16.7 ms. Assuming that each module accepts and outputs one pixel per clock, the simplest option is to choose an operating frequency of 148.5 MHz such that the pipeline runs at 60 fps i.e., the time to process a frame by the pipeline is 16.7 ms in steady-state. Another option is to clock the ASIC-style design at the maximum achievable clock frequency. For this example, let us assume a maximum frequency of 300 MHz which is readily achievable on current FPGA platforms for typical vision processing modules. When clocked at 300 MHz, the pipeline would be idle half of the time since it needs to wait for the camera to produce data. Also, the ASIC-style design clocked at 300 MHz would consume more power than the ASIC-style design clocked at 148.5 MHz. Therefore, in this example, the best option is to clock the ASIC-style design at 148.5 MHz even though the maximum achievable frequency is 300 MHz. In other words, computing at a faster rate than the rate at which the input data is produced does not provide any benefit in this example.

On the other hand, in a PR-style design, clocking the design at the maximum possible frequency is

beneficial to compensate for the reconfiguration time. In the PR-style design clocked at 300 MHz, the pipeline would process a frame in 6.9 ms, and would remain inactive during 9.8 ms until a new frame is produced by the camera. During this inactive time, the FPGA can be reconfigured with another pipeline that can start execution after its reconfiguration. Hypothetically, if both pipelines can be reprogrammed on the FPGA and can process a frame in less than 16.7 ms, this process can be repeated infinitely. We refer to this specific fine-grain usage of PR as real-time time-sharing. By doing real-time time-sharing, instead of using a larger FPGA on which the three pipelines are statically mapped, one can use a smaller FPGA potentially resulting in device cost, power and energy savings.

Note the difference between using PR at a fine-grain and real-time time-sharing. Real-time time-sharing is a specific fine-grain usage of PR in which *multiple pipelines are temporally shared on the FPGA and round-robin executed within the time-scale of a camera frame.* In other words, all pipelines need to process the same input frame from the camera (no frame skipping).

Real-Time Time-Sharing Challenges. To realize real-time time-sharing in practice, the biggest challenges are (1) to develop efficient PR execution strategies for maximizing the number of real-time time-shared pipelines and ensuring that all pipelines run at the desired performance (30+ fps), which is not trivial given the significant reconfiguration time on current FPGAs, and (2) to design architectural mechanisms to prevent camera data loss during pipeline reconfigurations. Considering the application example in Figure 5.1, a single large PR region solution (PR time is greater than 10 ms with a bitstream size larger than 1 MB) in which one full pipeline is reconfigured at a time would fail in many use-cases due to the large PR time (Figure 5.2). A PR solution with multiple smaller PR regions connected to each other (Figure 5.2) would also fall short since PR reconfigurations are serialized on current FPGAs (i.e. one reconfiguration happens at a time). All PR regions need to be reconfigured before the pipeline can start execution resulting in a non-trivial reconfiguration time in many scenario.

In the next section, we present the design of our framework leveraging PR for temporal and spatial sharing of multiple vision pipelines on an FPGA. The framework includes the necessary architectural and runtime mechanisms to tackle the aforementioned challenges and to support coarse-grain temporal sharing, real-time time-sharing and spatial sharing of the FPGA fabric. This framework aims at facilitating the implementation of PR-style designs used for our evaluation of PR costs and benefits in Chapters 6 and 7.

5.2 Background

This section uses the simple streaming vision pipelines depicted in Figure 5.1 to explain the operation of standard streaming vision pipelines necessary to comprehend real-time time-sharing.

We assume that the streaming vision pipeline is driven by a camera and outputs to a display, and that pixels are continuously streamed into the pipeline. The camera streams pixels into the first stage of the pipeline at a steady rate. $T_{\mathsf{frame,camera}}$ is the time between the first pixel and last pixel of a frame produced by a camera; the frame rate is $\frac{1}{T_{\mathsf{frame,camera}}}$. In a simple pipeline, all pipeline stages consume and produce pixels at the same steady rate as the camera, logically computing an output frame from each input frame. The stages may need to buffer multiple lines of the frame but never a complete frame. There is a delay between when the first pixel of a frame enters a stage (or a pipeline) and when the first pixel of the same frame exits a stage (or a pipeline); this time is $T_{\mathsf{fill,stage}}$. After the first pixel exits, the last pixel exits T_{frame} later. In steady-state with continuous streaming inputs, a complete frame would exit every T_{frame}.

For example, considering a camera that outputs full-HD frames at 60 fps, $T_{\mathsf{frame,camera}}$=16.7 milliseconds. When stages accept one pixel per clock, the pipeline needs to operate at a minimum frequency of 148.5 MHz to run at 60 fps. If a stage requires buffering of 10 lines of the frame, T_{fill} of that stage will be at least 0.13 milliseconds ($10 \text{ lines} \times \frac{1920 \text{ pixels}}{\text{line}} \times \frac{1}{148.5 \text{ MHz}}$).

Under basic operation, any given stage just needs to keep up with the pixel rate from the camera. However, a stage running by itself could be clocked faster resulting in shorter $T_{\mathsf{frame,stage}}$ and $T_{\mathsf{fill,stage}}$. For example, this is applicable when the streaming input and output of a stage are sourced from and sinked into DRAM instead of camera and display.

5.3 Spatial and Temporal Sharing of the FPGA by Multiple Computer Vision Pipelines

This section first gives an overview of our framework. Sections 5.3.2, 5.3.3, 5.3.4, and 5.3.5 describe into greater details the design of the important components in the framework for spatial and coarse-grain temporal sharing of the FPGA. Section 5.3.6 discusses the operation of real-time time-sharing and presents the architectural and runtime mechanisms needed to support this mode of operation.

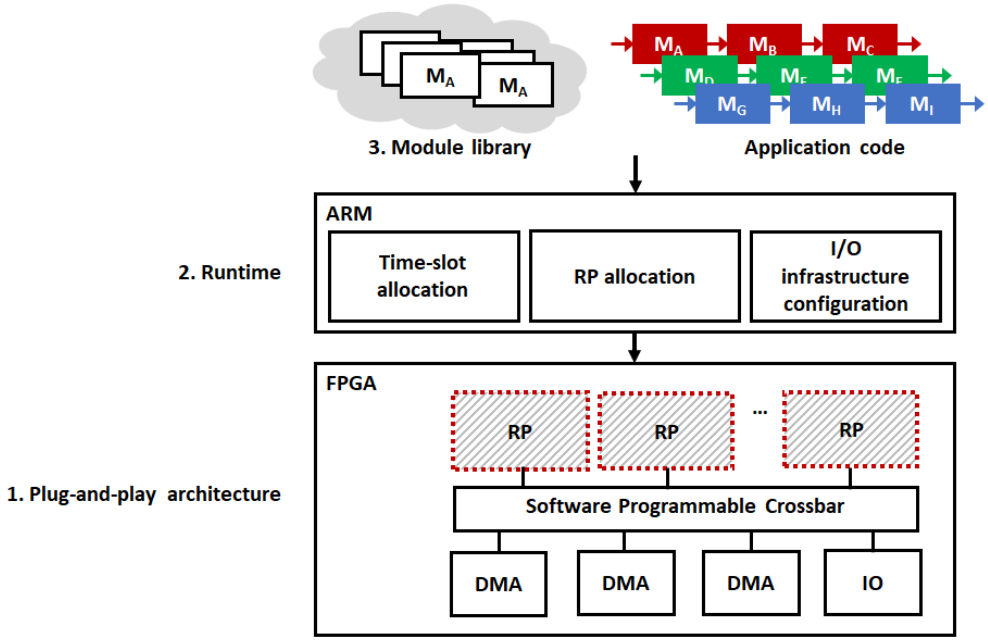

Figure 5.3: The framework is built on top of (1) a plug-and-play architecture divided into multiple reconfigurable partitions (RPs), (2) a runtime system which manages the reprogramming of the FPGA with the pipelines specified in the application code, and (3) a module library which contains pre-compiled vision modules used to reprogram the RPs.

5.3.1 Overview

In the framework, multiple computer vision pipelines can spatially and temporally share the FPGA. By spatial sharing, we mean that multiple pipelines can be mapped and executed simultaneously on the fabric. When more pipelines need to be accelerated than what the device can fit, the different pipelines can be time-shared on the FPGA; the FPGA can be reprogrammed at runtime with different sets of pipelines at a coarse or fine grain. We refer to repurposing as a coarse-grain usage of PR, i.e. the time interval between reconfigurations is within minute to hour range (similar to a "role-and-shell" approach). Real-time time-sharing is the specific fine-grain usage of PR that we investigate with a time interval between reconfigurations within the millisecond range.

Figure 5.3 shows the three main components of the framework:

- a plug-and-play architecture divided into multiple PR regions. The static I/O infrastructure consists of a software programmable crossbar that provides connectivity between PR regions and the rest of the system (camera, display, external memory, embedded CPU).

```
//create task objects
task_t task0;
task_t task1;
task_t task2;
//pedestrian detection
task0.name   = "pedestrian";
task0.input0 = &in_pedestrian;
task0.output = &out_pedestrian;
//car detection
task1.name   = "car";
task1.input0 = &in_car;
task1.output = &out_car;
//stereo
task2.name   = "stereo";
task2.input0 = &in_left;
task2.input1 = &in_right;
task2.output = &out_stereo;
//start the tasks
start_task(&task0);
start_task(&task1);
start_task(&task2);
```

Figure 5.4: Sample of the application code used to repurpose the FPGA with three tasks.

- a runtime system which manages the reprogramming of the FPGA with an application (set of pipelines) specified in an application code. In our implementation, the runtime system and the application code both run on an embedded CPU.
- a module library which contains pre-built vision modules used to reprogram the PR regions.

5.3.2 Programming Model

In the framework, the set of tasks to be reprogrammed and executed on the FPGA is specified in an application code. This section explains how to create a set of tasks in the application code using the proposed APIs and objects.

The process of creating and starting a task follows three steps: (1) create a task object, (2) specify the different fields for the task object (e.g. address of input/output buffers in external memory, image size), (3) start the task. This process is repeated for each task to be accelerated on the FPGA. Figure 5.4 shows an example of an application code to specify three vision tasks. In this example, each task takes as input one or two images stored in external memory and produces an output that is stored in external memory.

5.3.3 Plug-And-Play Architecture

As shown in Figure 5.3, the framework is built on top of a plug-and-play architecture which connects the PR regions and the external I/O (i.e. cameras, display and on-board external memory) via a static I/O infrastructure. The static I/O infrastructure is composed of a software programmable crossbar and direct memory access (DMA) engines. A PR region can communicate with another PR region and with external I/O through the software programmable crossbar. DMA engines allow data transfers between external on-board memory and the modules reprogrammed in the PR regions.

At design time, the plug-and-play architecture can be parameterized for different PR-style designs and modes of operation (i.e. repurposing or real-time time-sharing), that is, the number of PR regions, their size and I/O interfaces can be customized for a given design. The number of crossbar endpoints and the number of DMA engines are also parameters that can be changed at design time.

PR Region: Size and I/O Interfaces. When doing repurposing, PR regions can be large since PR time can be amortized over long enough time interval between reconfigurations. In this case, the scheduling entity can be an entire task or pipeline. When doing real-time time-sharing, using a large PR region approach is not a viable solution. Instead, we reconfigure each processing stage of a task in a small PR region. Section 5.3.6 discusses real-time time-sharing in more details.

In our framework, PR regions can have either streaming or memory-mapped interfaces. In the same design, all PR regions have the same I/O interfaces for data transfers. PR regions with streaming I/O interfaces can have up to two input interfaces and one output interface. PR regions with memory-mapped I/O interfaces can have up to four memory-mapped I/O interfaces. PR regions have one control interface used to send commands or parameters (e.g., start, filter size) to the module and to get the module status (e.g., done, idle).

Software Programmable Crossbar. The crossbar can be programmed at runtime in software (not reprogrammed in a PR sense) to provide general connectivity between the PR regions, the external I/O and the DMA engines. The software programmable crossbar allows multiple PR regions to communicate with each other, which is important to connect dependent hardware modules. Section 5.3.6 gives a more complete explanation on the benefits of the software programmable crossbar when doing real-time time-sharing.

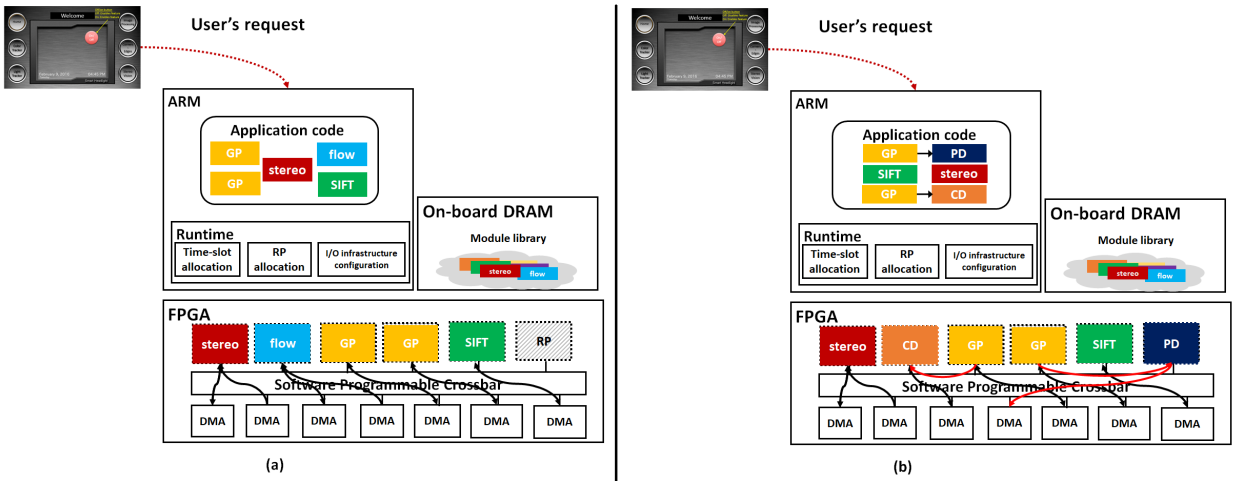

Figure 5.5: Example of an interactive application built on top of the framework. At user's request, a set of tasks specified in the application code is repurposed on the FPGA. The general connectivity of the software programmable crossbar allows to create complex tasks by module composition.

5.3.4 Module Library

The module library contains PR bitstreams used to reprogram the PR regions. Modules from the library have the same I/O interfaces as the PR regions. The library consists of macro and micro modules (see Chapter 3 Section 3.1 for the definition of a macro and a micro module). Each module has a set of parameters (e.g., image resolution, kernel weights, kernel size, downsampling factor) that can be configured in the application code. A macro module essentially accelerates a task or a full pipeline, and consists of many micro modules that accelerate stages of a task. An example of a macro module to accelerate a CNN for car detection is shown in Figure 5.11). The "car detection" macro module consists of a pipeline of micro modules that operate on single data elements (e.g., relu), small neighborhood of data elements (e.g., pooling), and large neighborhood of data elements (e.g., 3D convolutions casted as matrix-matrix multiplications). Micro or macro modules can be composed to design a wide range of vision pipelines. More complex tasks can be created when composing multiple macro modules as illustrated in Figure 5.5.b. In this example, two complex tasks are shown; each task consists of two macro modules. The first task consists of a pyramid and a pedestrian detection stages accelerated by a "pyramid" (GP) and a "pedestrian detection" (PD) macro module, respectively. The second task has a pyramid and car detection stages accelerated by a "pyramid" (GP) and a "car detection" (CD) macro module, respectively.

5.3.5 Runtime System

The reprogramming of the FPGA is managed by the runtime system which has five main functions:

- parse the application code,
- allocate PR regions and a time-slot for each task specified in the application code, and reprogram the PR regions with the appropriate modules from the module library,
- program the software programmable crossbar to effect the required connectivity,
- configure the DMA engines to map the correct input/output buffers to the tasks,
- configure the modules with user-specified parameters and start the modules.

Figures 5.5 (a) and (b) illustrate the operation of the framework at runtime. In this example, the framework is implemented on a System-on-Chip (SoC) platform with an embedded CPU and an FPGA. The interactive application to accelerate on the FPGA is specified in the application code that runs on the embedded CPU. Two sets of tasks are specified in the application code. At user's requests (via touchscreen), one set of tasks is reprogrammed and executed on the FPGA. The plug-and-play architecture has six PR regions connected to external memory via the software programmable crossbar that has seven DMAs. The module library is stored into on-board external memory. The runtime system is executed on the embedded CPU.

At the first user's request, the runtime system parses the application code and reprograms the FPGA with the first set of tasks with one stereo, one flow, two Gaussian pyramid (GP), and one SIFT (Figure 5.5 (a)). At the second request, the runtime system parses the application code and reprograms the FPGA with the second set of tasks with two complex tasks (Gaussian pyramid + pedestrian detection and Gaussian pyramid + car detection)), one SIFT and one stereo (Figure 5.5 (b)). The runtime system (1) reprograms the car and pedestrian detection tasks on the FPGA (the four other tasks are already programmed on the device), (2) changes the crossbar links (highlighted in red) to chain the Gaussian pyramid to the car detection task, and to compose the second Gaussian pyramid with the pedestrian detection task, (3) configures the required DMA engines (note that one DMA engine is shared between the flow and the pedestrian detection) and (4) start the execution of the tasks.

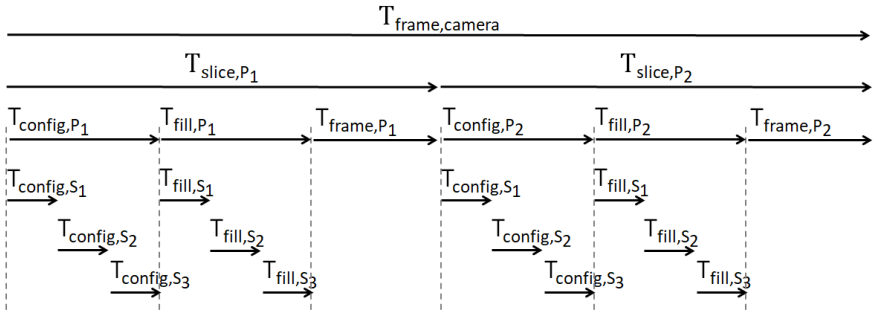

Figure 5.6: Time-sharing by two three-stage pipelines. A pipeline starts processing only after all stages have been configured.

5.3.6 Real-Time Time-Sharing Support

This section describes the operation of a basic real-time time-sharing system and its performance model. We then explain the technical challenges, the architectural and runtime mechanisms to support this mode of operation.

Real-Time Time-Sharing Operation. In real-time time-sharing, multiple pipelines round-robin execute in the time scale of a camera frame, i.e. all pipelines process the same camera input frame in the time scale of a camera frame. Since every input frame needs to be processed by every pipeline, initially we take $T_{\mathsf{frame,camera}}$ to be the basic scheduling quantum T_{round} for one round of round-robin execution. Each pipeline P_i is assigned a timeslice $T_{\mathsf{slice,P}_i}$.

When doing real-time time-sharing, each pipeline stage is assigned its own RP. This design choice is motivated by two reasons: (1) to reduce RP size, and therefore, reconfiguration time since a pipeline stage can be reconfigured in a smaller RP than a full pipeline, and (2) to reduce the number of reconfigurations required, and therefore, reconfiguration time since vision pipelines may share common stages (see for example Figure 5.16). The high cost of reconfiguring a RP can be avoided when an already configured processing stage can be retained and reused across pipelines.

Performance Model. During a pipeline's timeslice, the partitions needed by the pipeline are configured first, and then one camera frame is fully processed. If the total time to configure a pipeline P_i is $T_{\mathsf{config,P}_i}$,

$$T_{\mathsf{slice,P}_i} = T_{\mathsf{config,P}_i} + T_{\mathsf{fill,P}_i} + T_{\mathsf{frame,P}_i}$$

Note in the above, T_{fill} and T_{frame} are for when the pipelines are operating against DRAM.

For a valid real-time schedule,

$$\sum_{\text{all } \mathsf{P_i}} T_{\mathsf{slice},\mathsf{P}_i} \leq T_{\mathsf{round}} = T_{\mathsf{frame,camera}}$$

If the above condition is satisfied, all pipelines can keep up with the camera's frame rate.

Camera Input Buffering. Two main challenges need to be addressed when doing real-time time-sharing. We first discuss the input buffering issue before addressing the problem of high reconfiguration time. In real-time time-sharing, the same input frame from the camera needs to be presented to all pipelines during their respective time-slice (no frame skipping). Therefore, during pipeline reconfigurations, data produced by the camera needs to be double-buffered (1) to prevent camera data loss during pipeline reconfigurations, and (2) for every pipeline to process the frame stored in one buffer while the camera writes a new frame in the other buffer. Double-buffering also allows for decoupling the camera and pipeline rates. In an ASIC-style design, the camera and pipeline rates can be the same since the pipelines only need to stay in sync with the camera. When doing real-time time-sharing, the pipeline rate has to be greater than the camera rate to compensate for PR time (e.g., by clocking the PR-style design at a higher clock frequency than the ASIC-style design if possible). For pipelines in a PR-style design to run at a greater rate than in the ASIC-style design, there needs to be unexploited slack in the ASIC-style design (refer to Chapter 4 Section 4.3.4).

Given enough on-chip memory, one could double buffer the input frame from the camera in on-chip memory. However, the amount of data to buffer, which ranges from few KBs to MBs depending on the reconfiguration time and the camera frame rate and resolution, exceeds the on-chip memory available on low-end FPGAs (typically less than 4 MB on current low-end FPGAs). Therefore, in our implementations, we double buffer the input from camera into on-board external DRAM. For the frame rate and resolution targeted in this work, external memory bandwidth is not the bottleneck. To give an idea, assuming a camera running at 60 fps for full HD frames (YUYV422), the memory bandwidth requirement is approximately 249 MB/s to write the camera output to on-board external memory.

During each timeslice, the runtime framework drives the active pipeline with a pixel stream from DRAM at the maximum rate the pipeline can handle, or up to the DRAM bandwidth. The output of the pipeline is also double-buffered into DRAM so that multiple pipeline output streams can be merged for display by a function (e.g., XOR) or rendered simultaneously on a split-screen. Figure 5.6 illustrates an execution timeline when two three-stage pipelines are time-shared as described above.

Overcome Reconfiguration Time. This straightforward approach is not sufficient for achieving useful frame rates given the reconfiguration speed on today's FPGAs. The time to reconfigure a PR region ranges between a few to 10s of milliseconds on current FPGAs. Therefore, the time to reconfigure the PR regions

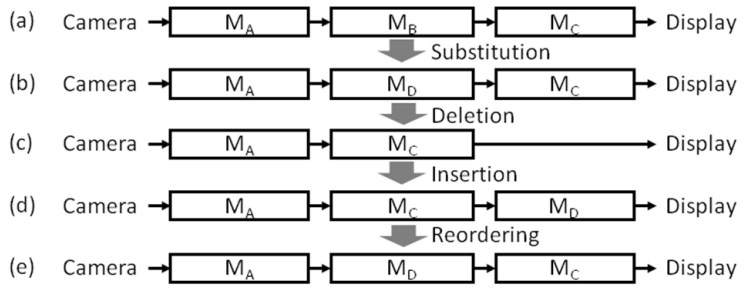

Figure 5.7: Using the software programmable crossbar, reconfiguration of RPs can be avoided by retaining and reusing already configured stages across pipelines. The main connectivity modes supported by the crossbar are 1. substitution 2. insertion/deletion 3. stage reordering.

for a pipeline is often comparable with the processing time. The time to reconfigure the PR regions is also significant relative to $T_{\mathsf{frame,camera}}$. If we use real-time time-sharing as described in previous section, the time to configure a pipeline alone will exceed $T_{\mathsf{frame,camera}}$ in most non-trivial scenarios. Next, we introduce additional techniques needed to achieve usable performance. These optimizations aim at (1) reducing the number of reconfigurations (configurable streaming interconnect), (2) hiding (staggered-start execution) or (3) amortizing (batching) the reconfiguration time when possible.

Configurable Streaming Interconnect. As explained above, in vision processing, even when pipelines have different functionalities, they may share common stages. The high cost of reconfiguring an RP can be avoided when an already configured processing stage can be retained and reused across pipelines.

Figure 5.7 identifies different ways for multiple pipelines to reuse common stage configurations. The simplest scenario is when switching from pipeline (a) to pipeline (b) where the two pipelines have the same topology and differ only in one stage. To switch from (a) to (b) (and vice versa), only the middle RP has to be reconfigured. When switching from (b) to (c), no RP reconfiguration is needed if there is a way to skip over the $\mathsf{M_D}$ stage of (b). Furthermore, by retaining $\mathsf{M_D}$ even though it is not used by (c), a switch from (c) back to (b) can also be done without RP reconfiguration. On the other hand, also with $\mathsf{M_D}$ in place, a switch from (c) to (d) does not require reconfiguration but requires a different streaming connectivity than going to (b). In fact, one can switch between (b), (c), (d) arbitrarily without RP reconfigurations but as combinations of deletion, insertion, or reordering by changing the connectivity between already configured RPs.

To support these different scenario, we provide a flexible software programmable crossbar to ensure

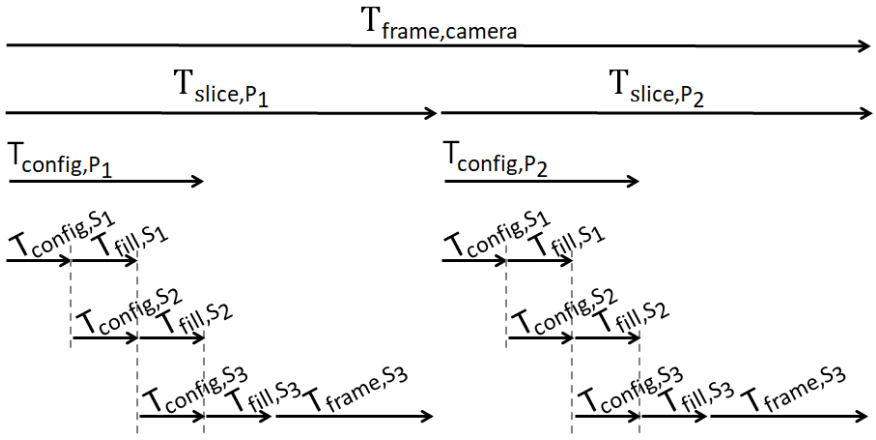

Figure 5.8: Time-sharing by two three-stage pipelines. A stage of a pipeline starts processing as early as possible, that is, when the stage is configured AND its upstream stage is producing output.

connectivity between RPs and other infrastructural elements. The crossbar connects all elements in the system—RPs, camera , display and streaming DRAM DMA engines—as inputs and outputs. The crossbar can be set by the controlling software between pipeline reconfigurations to establish the desired static streaming topology for the next timeslice. *Configuring the interconnect by software is 3-orders of magnitude faster than reconfiguring an RP.* The statically decided streaming connectivity topology never has two streaming sources going to the same destination so the crossbar can be simple and efficient with no need for flow control nor buffering. (To support forks and joins in the pipeline, some RPs have multiple input or multiple output interfaces while others have exactly one input and one output interface.) For simple streaming connections, the upstream and downstream stages are connected with a single-cycle buffered path. When a DRAM streaming connection is used to allow for buffering and stage decoupling, the source and sink RPs' streaming interfaces are redirected to/from the DMA engines of the infrastructure instead.

This flexible crossbar turned out to be a critical mechanism. The area expense of this interconnect infrastructure is well justified by the reconfiguration time savings. As we will see in the evaluation section, given the currently high cost of reconfiguration, practical time-sharing is only feasible if the number of reconfigured partitions between pipeline switches is kept to a minimum.

Staggered-Start: Overlap Reconfiguration and Compute. When doing real-time time-sharing as described above (Figure 5.6), we waited until all of RPs of a pipeline have been configured before starting processing. When all stages are ready, streaming processing can progress synchronously throughout the pipeline. However, given that reconfiguration time of a partition is significant relative to processing time, we

are motivated to overlap processing and reconfiguration by (1) reconfiguring the PR regions in order from first to last; and (2) streaming input into the earlier stages as soon as they are ready. Figure 5.8 illustrates the execution timeline for the same two pipelines used in Figure 5.6 but now starting a stage as soon as possible, in other words, when the stage is configured and its upstream stage is producing output.

In this staggered-start execution, it is possible for an upstream stage to start producing output before its downstream stage is ready. Thus, it becomes necessary to introduce buffering as a part of the streaming connection abstraction between a downstream stage being reconfigured and its upstream stage to support staggered start. The buffering capacity must be sufficient to capture all of the output of an upstream stage until the downstream stage is ready. Data is buffered into DRAM since the amount of data may exceed BRAM capacity. Hence, to buffer and delay the data stream until the downstream stage is ready, we need to use a decoupling DMA engine between each downstream stage being reconfigured and its upstream stage. In the worst case, we have found it necessary to support the option for streaming connections to be physically realized as a circular-buffer in DRAM. When the number of stages in a pipeline is large, the designer needs to provision sufficient *external memory bandwidth* and ensures that external memory bandwidth does not become the bottleneck.

With staggered start, $T_{\mathsf{slice,P}}$ of pipeline P is upper bounded by

$$T_{\mathsf{config,P}} + T_{\mathsf{fill,P}} + T_{\mathsf{frame,P}}$$

In the case when all the stages have comparable processing time $T_{\mathsf{slice,P}}$ is lower bounded by

$$T_{\mathsf{config,P}} + T_{\mathsf{fill,S_{last}}} + T_{\mathsf{frame,S_{last}}}$$

where $T_{\mathsf{fill,S_{last}}}$ and $T_{\mathsf{frame,S_{last}}}$ are T_{fill} and T_{frame} of the last pipeline stage only. When some stages have much longer configuration or processing time $T_{\mathsf{slice,P}}$ is more tightly lower bounded by

MAX over all stages S_i:

$$\left(\sum_{\mathsf{S}_j<\mathsf{S}_i} T_{\mathsf{config,S}_j}\right) + T_{\mathsf{fill,S}_i} + T_{\mathsf{frame,S}_i}$$

Amortization and Downsampling. In the basic scheme presented in previous section, a round of round-robin execution is completed for each quantum T_{round}=$T_{\mathsf{frame,camera}}$. This is not necessary. We can increase T_{round} to be a multiple $g\times T_{\mathsf{frame,camera}}$. In this case, we would double-buffer g frames at a time from camera into DRAM. During each pipeline's timeslice, the runtime framework drives the active pipeline with g consecutive active frames from the DRAM double-buffer. Thus the cost of reconfiguration is amortized over

a longer processing time. This option can be used with or without staggered start. In both cases, $T_{\mathsf{slice},\mathsf{P}_i}$ of pipeline P_i is now upper bounded by

$$T_{\mathsf{config},\mathsf{P}_i} + T_{\mathsf{fill},\mathsf{P}_i} + g\times T_{\mathsf{frame},\mathsf{P}_i}$$

For a valid real-time schedule,

$$\sum_{\text{all } \mathsf{P_i}} T_{\mathsf{slice},\mathsf{P}_i} \leq T_{\mathsf{round}} = g\times T_{\mathsf{frame,camera}}$$

The runtime framework can still produce a smooth video output when $g > 1$ because the output is also double-buffered. Beside the added storage cost, a major downside of increasing g is the very large increase in end-to-end latency through the runtime framework (which now includes the time to buffer multiple input and output frames).

Also note, increasing g improves scheduling by amortizing the reconfiguration cost. Therefore, it cannot help when the sum of the pipeline's processing time already exceeds $T_{\mathsf{frame,camera}}$. In this case, the only option is to downsample the video stream from camera into the pipeline. If the runtime framework selectively only passes every s frames of the camera input to the pipelines, the pipeline timeslices only need to fit within the new scheduling quantum of

$$T_{\mathsf{round}} = g \times s\times T_{\mathsf{frame,camera}}$$

5.4 Evaluation

This section presents our evaluation of the framework. We first evaluate the performance of vision tasks when (1) executed individually, (2) spatially sharing the fabric, and (3) temporally sharing the fabric at a coarse-grain (repurposing) i.e. the time interval between reconfigurations is within minute to hour range. We show that tasks executed individually within our framework (1) achieve comparable performance to tasks mapped statically on the FPGA, and (2) are up to two orders of magnitude faster than a CPU implementation. When up to six vision tasks are executed simultaneously within our framework deployed on a Zynq FPGA, we show that they can achieve the same performance as when executed individually (up to 169 MPixels/s). Second, we demonstrate that it is possible to time-share up to three pipelines on a Zynq 7000 Series FPGA and achieve 30+ fps. Chapter 7 further evaluates the benefits and costs of real-time time-sharing for different PR speeds in a simulation-based limit study.

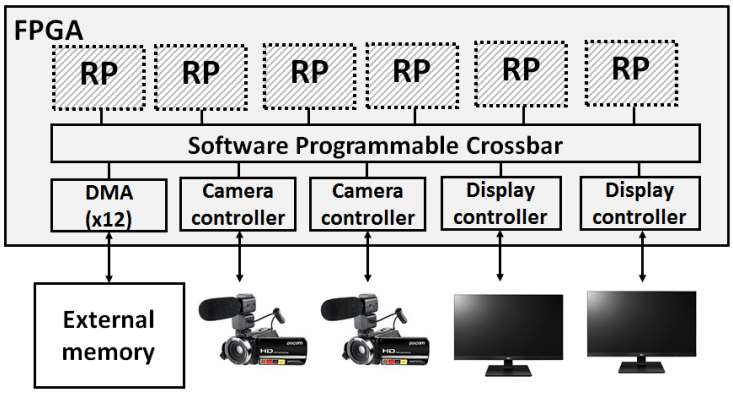

Figure 5.9: In our prototype framework for repurposing, the FPGA is connected to external memory (via 12 DMA engines), to two HDMI cameras and to two HDMI displays.

Table 5.1: Resource used by the plug-and-play architecture (static I/O infrastructure and six reprogrammable partitions) and percentage utilization (in parenthesis) on the Zynq 7045 FPGA. The static infrastructure comprise a 20-endpoint crossbar and 12 DMA engines. The six reprogrammable partitions contain roughly the same amount of logic and memory resources.

	I/O Infrastructure			**Compute (6 PR partitions)**
	Crossbar	DMA engines	Misc	
LUT	5580 (3%)	22,270 (10%)	2428 (1%)	173,400 (80%)
BRAM	0	36 (7%)	7 (1%)	500 (92%)
DSP	0	0	0	840 (93%)

5.4.1 No Real-Time Time-Sharing Setup

Implementation. We implement a prototype of the framework (Figure 5.9) for repurposing on the ZC706 board [91] using Vivado 2015.2 tool chain [89]. The plug-and-play architecture (also referred as PR-style design) has six reprogrammable partitions connected to on-board external memory, to two HDMI cameras and to two HDMI displays via the software programmable crossbar. Two ARM cores are available on the Zynq boards. We use one core that runs Linux to load input test images into external memory, and to save the output of vision tasks to disk. The other bare-metal core executes the runtime system and the application code.

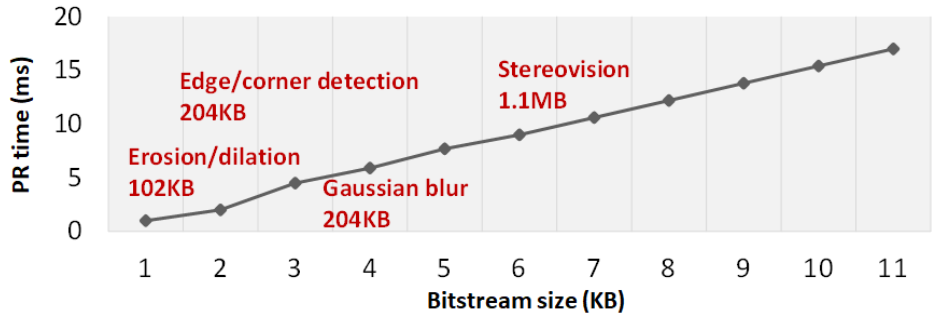

Figure 5.10: The PR time for a single partition is proportional to its size, and ranges from 2 ms to tens of ms on an FPGA from the Zynq 7000 Series family.

PR-Style Design Area. We first characterize the area and repurposing overhead of the PR-style design implemented. Table 5.1 reports the number of logic resources used by the static I/O infrastructure and the reprogrammable partitions on the Zynq 7045 FPGA. The six reprogrammable partitions consume more than 80% of the FPGA logic and memory resources and have similar size, i.e., they contain approximately the same number of logic and memory resources. The static I/O infrastructure comprise a 20-endpoint crossbar and 12 DMA engines. Each DMA engine allows for one concurrent read and write streaming accesses to external memory.

Repurposing Time Overhead. The time overhead for repurposing the FPGA with a single module is the sum of the times for:

- reprogramming a partition with the module. Figure 5.10 shows the average time for reprogramming the partitions through the processor configuration access port (PCAP) on the FPGA when scaling the number of partitions. We observe that the reprogramming overhead ranges between 17 to 102 ms depending on the size of the PR region.
- programming the crossbar links,
- configuring the DMA engines.

We ran hundreds of tests to ensure that the time for programming the crossbar links and for configuring the DMA engines in software is negligible compared to the time spent for reprogramming a partition. (The crossbar and DMA operations take in the order of few microseconds.) Therefore, the time to repurpose the FPGA with a single a module is, at first order, equal to the time to reprogram a partition.

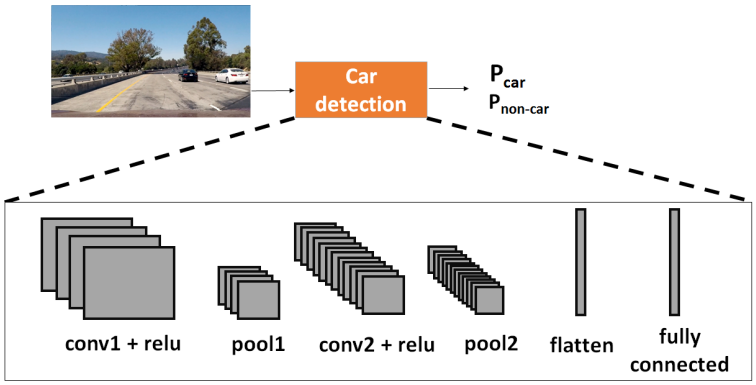

Figure 5.11: Architecture of the CNN used for inference for the car detection task.

Module Library. When repurposing, the scheduling entity is a task, that is, a PR region is reprogrammed with a macro module that accelerates an entire task or pipeline at a time. We implement six macro modules using Xilinx high-level synthesis tool (Vivado HLS 2015.2 [88]) to integrate into our module library. We performed hundreds of cycle-accurate simulations to ensure the functional correctness of each module. We have used these modules to build live demonstrations of this framework shown at multiple venues.

- *stereo*: we implement depth from stereo vision based on the block matching algorithm [71] using a 7×7 search window. In this example, we process eight lines and disparities in parallel.

- *flow*: we implement 1 iteration of Lucas-Kanade [57] to compute the dense optical flow information using a 5×5 search window. We use floating point type for the linear algebra operations (e.g., matrix-matrix multiplications, matrix inversion) performed by the algorithm.

- *SIFT* : we implement the Scale Invariant Feature Transform (SIFT) algorithm [56] to detect key points in an image.

- *pyramid*: we implement a pyramid of Gaussian based on [1]. The input image is first blurred with a Gaussian filter before being downsampled by a vertical and horizontal factor that can be changed at runtime.

- *car detection*: we implement a 5-layer convolutional neural network (CNN) inference for car detection based on a binary classifier. We use the Keras library [11] written in Python to train the network using the Udacity 2017 challenge training data set. The network architecture is depicted in Figure 5.11. This network processes 64 × 64 images.

Table 5.2: Clock and measured throughput of the *stereo*, *flow* and *face detection* modules when executed individually (1) within our framework and (2) within frameworks that use a static FPGA design flow.

	stereo		**flow**		**face detection**	
	Clock (MHz)	Throughput (MPixels/s)	Clock (MHz)	Throughput (MPixels/s)	Clock (MHz)	Throughput (MPixels/s)
Darkroom [37] (Zynq 7100)	169	148	169	165	N/A	N/A
Rigel [38] (Zynq 7100)	169	148	169	165	N/A	N/A
[74] (Zynq 7045)	N/A	N/A	N/A	N/A	N/A	2.5
Framework (Zynq 7045)	169	152	169	161	169	2.3

Table 5.3: Measured throughput of the *stereo*, *flow*, *SIFT*, *pyramid*, *car detection*, and *pedestrian detection* modules when executed individually within our framework vs. when executed on a i7-3770@3.40 GHz machine. Within our framework, all modules are clocked at 169 MHz.

	stereo	**flow**	**SIFT**	**pyramid**	**car detection**	**pedestrian detection**
Throughput on a i7-3370@3.40 GHz (in MPixels/s)	13	10	4	230	4	4
Throughput within the framework (in MPixels/s)	152	161	161	169	169	169

- *pedestrian detection*: we implement a 5-layer CNN inference architecture to detect pedestrians. The CNN inference architecture is based on the CNN architecture used for accelerating the *car detection* task. We use the Keras library written in Python to train the network using the training data set from [16].

We also use a *face detection* implementation from [74] based on the Viola Jones feature detection algorithm [84]. We change the interfaces of the design so that it can be compiled and executed within our framework. This *face detection* module processes image sizes of up to 256×512.

5.4.2 No Real-Time Time-Sharing Results

In this section, we present an evaluation of the framework to demonstrate that modules executed individually within our framework (1) have performance comparable to modules mapped statically, and (2) are up to two orders of magnitude faster than CPU implementations. We also demonstrate that when up to six tasks are executed simultaneously on the FPGA, each task can achieve the same throughput as when executed individually (up to 169 MPixels/s).

Table 5.4: Resource utilization of the *stereo*, *flow*, *SIFT*, *pyramid*, *face detection*, *car detection*, and *pedestrian detection* modules executed individually within our framework. The number in parenthesis represents the percentage of resource utilization on the Zynq 7045 FPGA.

	stereo	**flow**	**SIFT**	**pyramid**	**face detection**	**car detection**	**pedestrian detection**
LUT	20396 (9%)	7509 (8%)	28397 (13%)	12234 (6%)	47134 (22%)	71543(33%)	71543(33%)
BRAM	61 (11%)	80 (15%)	37 (7%)	24 (4%)	125 (23%)	44.5(8%)	44.5(8%)
DSP	0 (0%)	39 (4%)	83 (9%)	55 (6%)	104 (12%)	126(14%)	126(14%)

Performance of Individual Modules. We first characterize the performance and the area of the seven modules presented in section 5.4.1 when reprogrammed and executed individually within the framework. We compare the performance of tasks executed within the framework with (1) tasks mapped statically and (2) CPU implementations.

Table 5.2 reports the throughput achieved by the *stereo*, *flow* and *face detection* modules when compiled and executed individually (1) within our framework, and (2) within frameworks that use a static FPGA design flow [37,38,74]. The FPGA that we use (Zynq 7045) has 21% less logic cells than the FPGA (Zynq 7100) used by Darkroom [37] and Rigel [38]. Still, we observe that the *stereo* and the *flow* modules placed and routed within our framework can be clocked at the same frequency as modules placed and routed with Darkroom and Rigel (169 MHz). Our implementation of the *stereo* and the *flow* modules achieves a throughput of 152 and 161 MPixels/s, respectively, which is comparable to the performance obtained for modules placed and routed within Darkroom and Rigel. Within our framework, the *face detection* module is clocked at 169 MHz and achieves a throughput of 2.3 MPixels/s which is comparable to the throughput reported in [74].

Table 5.3 reports the throughput of the *stereo*, *flow*, *SIFT*, *pyramid*, *car detection* and *pedestrian detection* modules when executed individually (1) within our framework and (2) on a i7-3770@3.40 GHz. The CPU implementations are written in Python and use APIs from the OpenCV [44] and the Keras library. Most tasks accelerated on the FPGA are one to two orders of magnitude faster than the CPU implementation.

Module Resource Utilization. Table 5.4 shows the resource utilization of the *stereo*, *flow*, *SIFT*, *pyramid*, *face detection*, *car detection* and *pedestrian detection* modules. Three modules (*face detection*, *car detection* and *pedestrian detection*) consume more FPGA resources than available in a single reprogrammable partition.

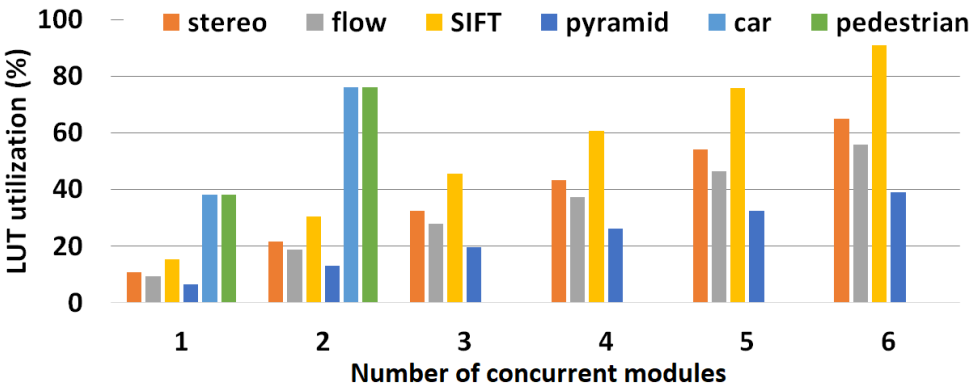

Figure 5.12: The FPGA can be reprogrammed with up to six *stereo*, *flow* or *pyramid*, with up to five *SIFT*, with up two *car detection* or with up two *pedestrian detection* modules at once, while utilizing less than 80% of the FPGA logic resources.

To repurpose the FPGA with these three modules, we use the amorphous technique (Chapter A.2) to fuse the resources from multiple reprogrammable partitions into one larger reprogrammable partition that can be repurposed with any of these modules.

Performance of Concurrent Modules. Next, we evaluate the performance of multiple modules sharing the fabric, that is, multiple modules are mapped and are executed simultaneously on the fabric. We show that module performance degrades when increasing the number of modules mapped simultaneously. This performance degradation is mainly due to the limited amount of on-board external memory bandwidth available for streaming data to multiple modules and writing the results back. We use all modules presented in section 5.4.1 except for the *face detection* module since its performance is not on par with the performance of other modules.

We first consider reprogramming the FPGA with identical and independent modules that are executed simultaneously. To evaluate the number of identical and independent modules that can be reprogrammed on the FPGA, we report the logic utilization when scaling the number of identical modules reprogrammed on the FPGA in Figure 5.12. We aim at using less than 80% of logic resources available in the reprogrammable partitions to ease the placement and routing process. To achieve a logic utilization of less than 80%, the FPGA can be reprogrammed with up to six *stereo*, *flow*, and *pyramid* modules, with up to five *SIFT* modules, with up to two *car detection*, and with up to two *pedestrian detection* modules.

Figure 5.13 shows the average throughput of a vision module when executed with up to five other identical and independent modules. As the baseline, we report the throughput of the module when executed

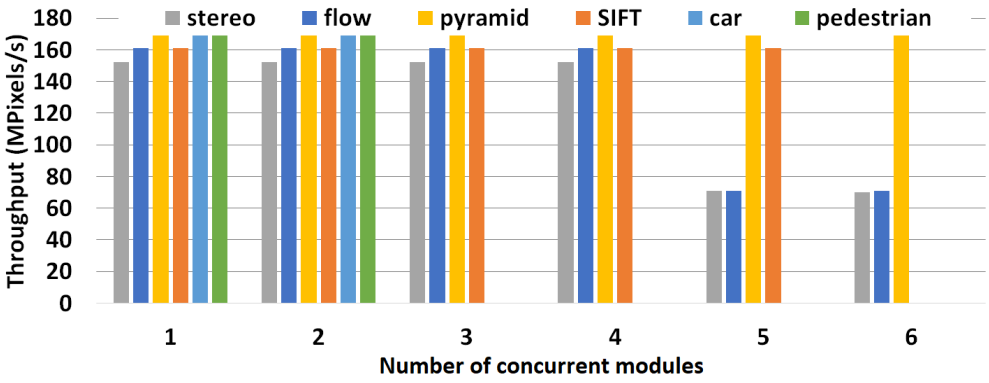

Figure 5.13: Average throughput of a module when running concurrently with up to five other identical and independent modules. When up to four *stereo*, four *flow*, six *pyramid*, five *SIFT*, two *car detection* or two *pedestrian detection* execute concurrently, the throughput of a *stereo*, *flow*, *pyramid*, *SIFT*, *car detection* or *pedestrian detection* is the same as when executed by itself within the framework.

individually within the framework. For the cases of *SIFT* and *pyramid*, the performance of each module when executed concurrently with four other *SIFT* or five other *pyramid*, respectively, is the same as when executed individually (161 MPixels/s and 169 MPixels/s, respectively). When executing two *car detection*, the performance of a single *car detection* is the same as when executed individually (169 MPixels/s). When executing two *pedestrian detection*, the performance of a single *pedestrian detection* is the same as when executed individually (169 MPixels/s). In the case of the *stereo* and the *flow* modules, the performance of a *stereo* and a *flow* module when executed concurrently with less than three other *stereo* or three other *flow*, respectively, is the same as when executed by itself within the framework (152 MPixels/s and 161 MPixels/s, respectively). When more than five *stereo* and *flow* modules are running simultaneously, the performance of a single module degrades by approximately 54% (71 MPixels/s).

Random Module Combinations. To demonstrate that the performance of multiple concurrent modules is limited by the amount of memory bandwidth available, we generate tens of random module combinations and repurpose the FPGA with one random module combination at a time. In each module combination, we choose randomly up to six modules among the *stereo*, *flow*, *car detection* and *pedestrian detection* modules, that require two concurrent external memory accesses (one read and one write), and the *SIFT* and *pyramid* modules, that require three concurrent external memory accesses (two reads and one write).

Figure 5.14 reports the average throughput (in MPixels/s) of a vision module executed in a random module combination vs. the total number of concurrent accesses to external memory required by all modules

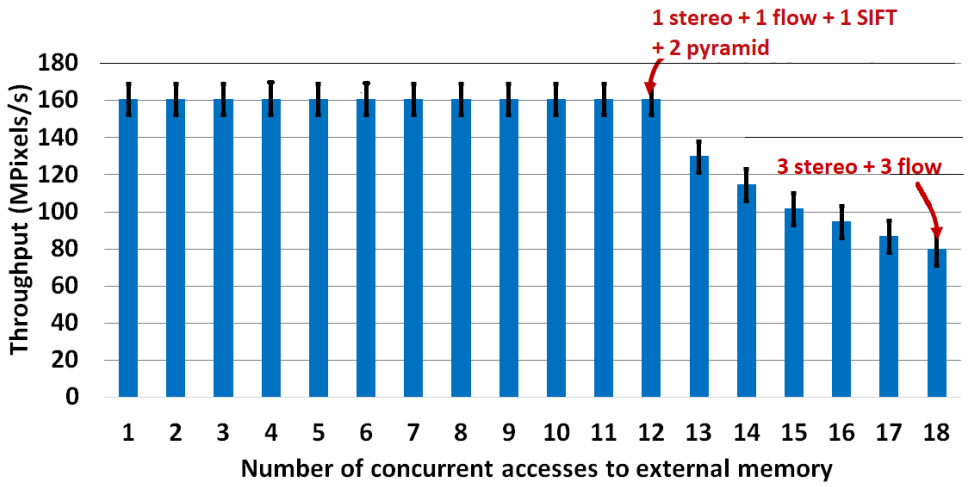

Figure 5.14: Average throughput of a single module executed within a random module combination vs. number of concurrent accesses to external memory required by the set of modules in a random module combination.

in a combination. We annotate Figure 5.14 with some random module combinations. We observe that the average performance of a single vision module when the number of concurrent accesses to external memory is strictly less than 13 is 161 MPixels/s with a population standard deviation of 6.3. With more than 13 concurrent accesses to external memory, the performance of a single module degrades by up to 25% on average with a population standard deviation of 6.8. For more than 13 concurrent accesses, up to 6.1 GB/s of memory bandwidth are needed to stream data continuously from (to) external memory to (from) multiple modules.

Module Composition Performance. In this last set of results, we discuss the performance of complex tasks formed by composing macro modules (refer to Section 5.3 for examples of complex tasks). We show that the performance of a single complex task executed in the framework is up to two orders of magnitude faster than a CPU implementation.

Figure 5.15 reports the average runtime to process a single image by two complex tasks composed of two modules for different input image sizes when one task is (1) executed individually within our framework and (2) executed individually on a i7-3770@3.40 GHz. Both tasks blur an image before doing CNN inferencing. The first task is composed of a *pyramid* and a *car detection* module. The second task has a *pyramid* and a *pedestrian detection* module. On the CPU, the time for processing an image by the *pyramid + car detection*

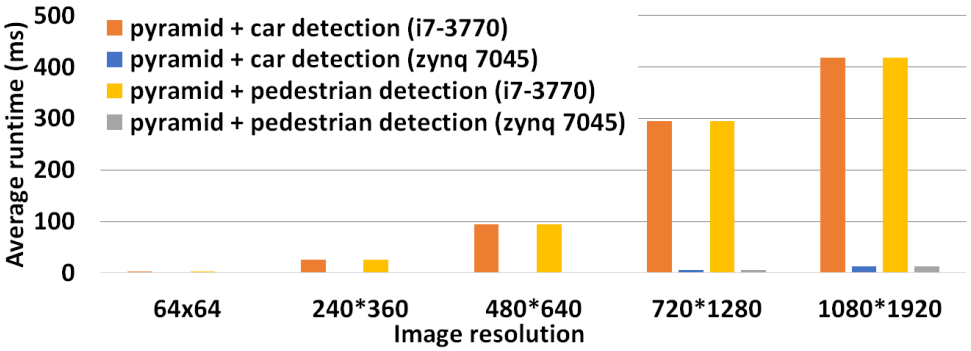

Figure 5.15: Average runtime to process one image by the *pyramid + car detection* and *pyramid + pedestrian detection* tasks for different image sizes when executed (1) within the framework and (2) on a i7-3370@3.40 GHz machine.

task is 2.2, 25, 95, 294 and 417 ms for 64×64, 240×360, 480×640, 720×1280 and 1080×1920 images, respectively. On the FPGA, the time for processing an image by the *pyramid + car detection* task is 0.032, 0.5, 2, 5.6 and 12.5 ms for 64×64, 240×360, 480×640, 720×1280 and 1080×1920 images, respectively. The FPGA implementation of the *pyramid + car detection* task is up to two orders of magnitude faster than the CPU implementation. Accelerating the *pyramid + pedestrian detection* task on the FPGA yields the same performance benefits as when accelerating the *pyramid + car detection* task.

5.4.3 Real-Time Time-Sharing Setup

The previous sections presented a characterization and an evaluation of the framework used for spatial and temporal sharing of the fabric at a coarse-grain. This section discusses the experimental setup to evaluate the performance of real-time time-shared tasks in our framework. Specifically, we characterize the resource utilization of the plug-and-play architecture before diving into the details of the management software executing on the embedded CPU. We also describe the implementation of the modules used in this case-study and their memory bandwidth requirements. The next section presents our real-time time-sharing results.

Implementation. We implement a prototype of the framework for real-time time-sharing. The prototype system is built on a Xilinx ZC706 development board with a Xilinx XCZ7045 Zynq SoC FPGA. (Table 5.5 gives the specifications of this board.) The FPGA is connected to a camera (VITA 2000-sensor) that supports up to full HD resolution at 60 fps (1080p@60fps). The video output of the time-shared pipelines is displayed

Table 5.5: Xilinx ZC706 board specification

FPGA	Xilinx XCZ7045
Hard CPU Cores	2 x ARM A9
LUT	218,600
BRAM36Kb	545
DSP	900
DRAM Bandwidth	12.8 GB/s (Fabric only)

on a split-screen.

Static I/O Infrastructure. The backbone of the plug-and-play architecture is the software programmable crossbar discussed in Sections 5.3.3 and 5.3.6. This interconnect infrastructure provides streaming connections between ten RPs for vision processing stages, the camera controller input, HDMI controller output, as well as five DMA engines for streaming to and from DRAM buffers. The interconnect is based on a custom crossbar implementation but the interface follows AXI4-Stream standard. Once configured, the interconnect infrastructure is capable of streaming frames at 1080p@60fps between a fixed pair of source and sink RPs. Two RPs can also be connected by a DRAM streaming connection that incorporates a circular-buffer FIFO in DRAM. Except for the camera and display controllers, the entire system–static I/O infrastructure and reconfigurable partitions—are by default clocked at 200 MHz. The camera and display controllers are clocked at 148.5 MHz.

Management Software. At runtime, a manager running on the embedded ARM processor core manages the creation, execution and time-sharing of vision pipelines. The specification of each pipeline (such as number of stages, module running in each stage and connectivity between stages) is registered with the runtime manager. To switch execution to a new pipeline, the runtime manager assigns a stage to an RP if the RP already has the required module. The RPs for the remaining stages are reconfigured through the PCAP interface with bitstreams loaded from on-board DRAM. Once the partitions are reconfigured, the runtime manager configures the modules (e.g., setting image size, kernel size), DMA engines, and interconnect to effect the required connectivity before starting pipeline execution. The built-in camera and display controllers are initialized once when the FPGA is first started.

For time-sharing, the runtime manager will cycle through all of the registered pipelines once for every g frames of video. The runtime manager will poll the active pipeline for completion before initiating a switch to the next pipeline. This runtime manager does not do scheduling or enforce maximum time quantum. If

Table 5.6: Logic resource used by the static I/O infrastructure and the reconfigurable partitions on the Xilinx XCZ7045.

	static I/O infrastructure			**Reconfigurable**
	Crossbar	DMA engines	Misc	
LUT	4940 (2%)	10725 (5%)	30578 (14%)	122400 (56%)
BRAM36Kb	0	15 (3%)	23.5 (4%)	360 (66%)
DSP	0	0	0	300 (33%)

total time to cycle through all of the pipelines exceeds the time quantum of $g \times s \times T_{\mathsf{frame,clock}}$, the processing falls out of sync to produce glitching output.

Vision Modules. When doing real-time time-sharing, the scheduling entity is a pipeline stage, that is, a module reprogrammed in a PR region accelerates a pipeline stage. We use Xilinx Vivado HLS to develop the vision modules used in this study. We also make use of the HLS video library that offers a subset of HLS-synthesizable OpenCV functions. These HLS-based modules can be incorporated into our runtime framework since our interconnect supports AXI4-streaming interface.

Logic Resource Utilization. Table 5.6 breaks down the fabric resource utilization between the static I/O infrastructure and reconfigurable partitions. The I/O infrastructure requires a non-trivial amount of resources. The crossbar is only a small fraction of the total infrastructure. On the other hand, the DMA engines to stream data through DRAM is quite expensive. On a large FPGA like the Xilinx XCZ7045, ample resources remain to be divided as ten independent reconfigurable RPs. We aimed for a total fabric utilization of roughly 70% to ease the placement and routing process.

DRAM Bandwidth. The peak DRAM bandwidth available on the Xilinx ZC706 development board is 12.8 GB/s (DRAM on the PS side). This bandwidth is shared by all of the DRAM streaming connections through AXI HP ports. To support 1080p@60fps, each DRAM streaming connection requires a total of 497 MB/sec of memory bandwidth (read and write). DRAM streaming connections include the double-buffers for the camera input and display output, and the decoupling buffers needed to support staggered start execution (Section 5.3.6).

We created a microbenchmark to measure the total DRAM bandwidth actually utilized when increasing the number of active thru-DRAM streaming connections. On the Xilinx ZC706 development board, up to five concurrent thru-DRAM streaming connections can be supported for 1080p@60fps. Since two thru-DRAM

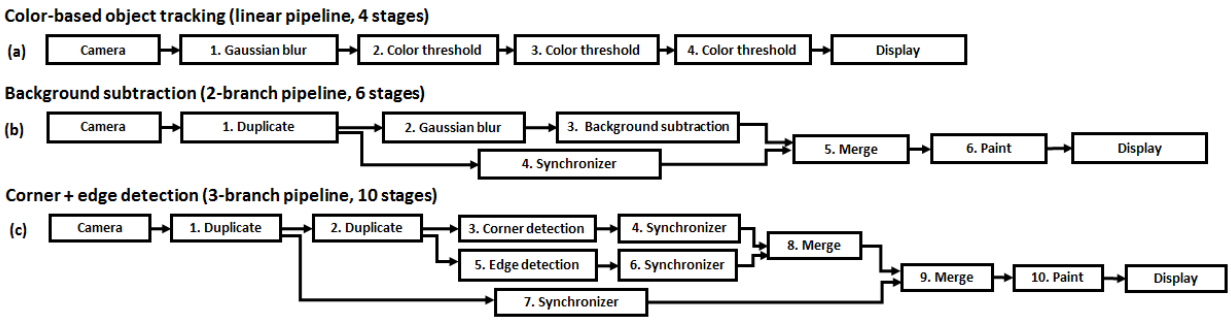

Figure 5.16: Logical view of three pipeline examples: (a) color-based object tracking where objects of up to three different colors are tracked, (b) background subtraction, (c) corner and edge detection.

streaming connections are taken up by camera and display for double-buffering, we are left with only three usable thru-DRAM streaming connections for decoupling the staggered start of RPs (Section 5.3.6). This restricts the applicability of the staggered start optimization in the prototype.

5.4.4 Real-Time Time-Sharing Results

This section presents our real-time time-sharing results on a case study of implemented designs. This evaluation aims to show that useful real-time performance (30+ fps) can be achieved when time-sharing multiple streaming vision pipelines on today's FPGAs. We first discuss the opportunities of using PR for accelerating computer vision pipelines, and start to quantify the benefits of real-time time-sharing in terms of area over an ASIC-style approach. We then present our results when measuring the achieved throughput of time-shared pipelines under different operating conditions with the camera running at 720p@60 fps and 1080p@60 fps. Chapter 7 further elaborates on the costs and benefits of real-time time-sharing for different PR region sizes, compute times, and PR speeds in a simulation-based study.

ASIC-Style Design Limitation. The dynamic adaptation requirement of interactive real-time vision systems leads to a potentially large number of pipelines to execute at runtime. These pipelines can have a variable topology and number of stages. Figure 5.16 shows the logical view of three pipeline examples that have different number of stages and topology. A non-linear pipeline topology allows to overlay different masks, computed on each branch, on the original camera frame. Each pipeline branch can execute one or a combination of vision modules listed in Table 5.7 leading to many potential different pipelines.

Traditionally, one way to implement such a system is to map all pipelines simultaneously and statically on the FPGA (ASIC-style approach). Table 5.8 presents the logic resources used by seven of the most

Table 5.7: Vision modules used in our evaluation.

Edge detection	computes a binary mask of vertical and horizontal edges using a Sobel filter [44]
Color-based object tracking	tracks objects based on their color
Template tracking	tracks a given template by computing sum-of-absolute differences and by thresholding
Corner detection	computes a binary mask of corners using a Harris corner detector [44]
Blob detection	detects blobs by using morphological operations and thresholding
Gaussian blur	blurs an image by using a Gaussian filter
Background subtraction	removes frame background by thresholding

Table 5.8: Logic resource used by seven 3-branch pipelines. When individually mapped in an ASIC-style design, each pipeline runs at 250 MHz.

	edge + corner	edge+template	blob+color	edge+color	corner+template	background + corner	background + edge
LUT	13147	13098	14142	13601	14085	14797	13810
FF	12455	11635	11423	11234	12146	13222	12711
BRAM36Kb	5	5	3.5	3.5	5	3.5	3.5

resource-expensive 3-branch pipelines, when each pipeline is mapped individually and directly on the FPGA (ASIC-style approach). These numbers give an idea of the potential cost of mapping a large number of pipelines statically on an FPGA. For instance, mapping all seven pipelines would consume more than 90k LUTs. If all possible linear and non-linear pipelines were to be mapped statically and simultaneously, they would possibly not fit on the FPGA. Note that mapping those pipelines individually and directly to the FPGA results in the best possible performance. When mapped individually on the FPGA, each of these pipelines can make timing at 250 MHz. We expect performance to degrade with an increasing number of parallel pipelines to map statically.

PR Performance Results. PR presents a viable alternative to overcome the resource limitation and the inflexibility of an ASIC-style design. When using PR, we expect the performance to degrade compared to the ASIC-style design due to the RP I/O placement port constraint that can add wire delay. For real-time time-sharing, the performance needs to be sufficient for correct pipeline operation for 720p@60fps and 1080p@60fps input video. Also, there should be enough slack to interleave pipelines at the time scale of a camera frame.

To assert the performance of a PR system, we use the system described in the previous section. The ten RPs are differently sized to support repurposing and real-time time-sharing. The four largest RPs (bitstream size of 1.1 MB) are used for repurposing while the six smallest RPs (bitstream size of 300 KB) are used for real-time time-sharing. We generate partial bitstreams for the seven modules such that all modules can be hosted in any of the smallest RP. We are able to generate partial bitstreams at 200 MHz in the PR-style design. Despite the expected performance degradation, pipelines can operate correctly for 720p@60fps and 1080p@60fps input video when using PR. An operating speed of 200 MHz also allows to time-share pipelines at the time scale of a camera frame.

Reconfiguration Overhead. Before presenting the performance of time-shared pipelines, we need to simplify the set of pipelines that we use for the time-sharing evaluation. To do so, we perform a first set of experiments to assert that the cost of switching from one pipeline to another is dominated by the PR time. The time cost for switching pipelines is the sum of the time (1) to reconfigure the PR regions, (2) to configure the interconnect, the DMA engines and the modules and (3) to start the pipelines. In these experiments, the pipelines occupy up to ten stages and have up to three branches.

We generate randomly tens of pipeline pairs (with different topology, different stages, and different number of stages), and measure the time to interleave two pipelines in a pair. We find that the time overhead that most matters is the time spent in RP reconfiguration. RP reconfiguration dominates the cost of a pipeline switch by three orders of magnitude. The time cost of reconfiguring the interconnect for topology change, and other configuration and startup cost, are within the range of 50s to 100s of microseconds depending on the number of RPs and interconnect links to reconfigure. Switching between pipelines with different topology does not impact time-shared performance. If switching from one pipeline to another does not change the state of any RP, the switch is almost free. (This is the case, for instance, when the set of stages used by one pipeline is a subset of the stages used by the other pipeline.)

Performance of Time-Shared Pipelines. For this evaluation, we only consider linear pipelines since pipeline topology does not impact performance as established previously. For these experiments, pipelines have up to six stages, and two interleaved pipelines differ by one, two, three, four, five and six RPs. (We only reconfigure the six smallest RPs while the four largest RPs remain unchanged. The time spent in reconfiguring RPs is proportional to the number of RPs to reconfigure since the six RPs have the same size.) When time-sharing, we execute pipelines (1) two at a time, and (2) three at a time.

Figure 5.17 first summarizes the achieved throughput in fps when the PR-style design is driven with a

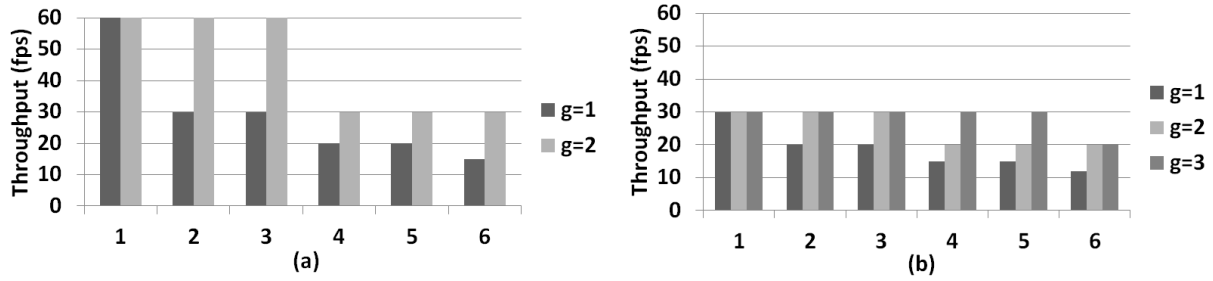

Figure 5.17: Throughput in fps for each time-shared pipeline for 720p@60fps input video when we execute (a) two pipelines or (b) three pipelines at a time. We reconfigure between one to six RPs per pipeline switch. Each time-shared pipeline processes g consecutive frames before reconfiguration.

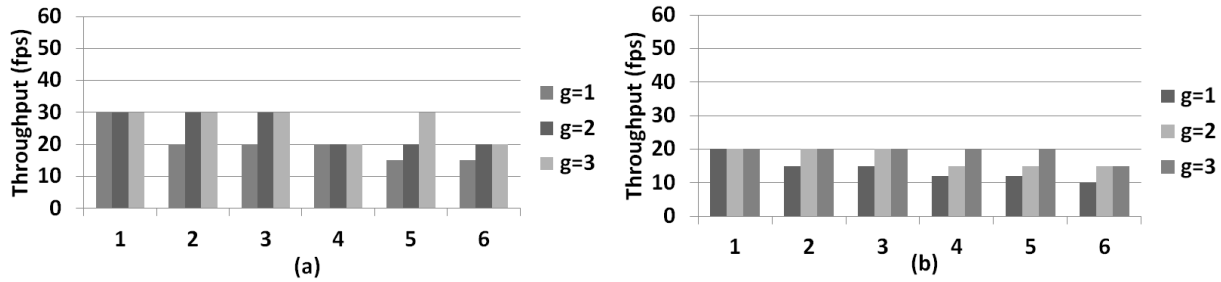

Figure 5.18: Throughput in fps for each time-shared pipeline for 1080p@60fps input video when we execute (a) two pipelines or (b) three pipelines at a time. We reconfigure between one to six RPs per pipeline switch. Each time-shared pipeline processes g consecutive frames before reconfiguration.

720p@60fps video stream, and when we execute (a) two pipelines at a time (b) three pipelines at a time by time-sharing. In Figure 5.17.a and Figure 5.17.b, there are six sets of bars corresponding to cases where we reconfigure between one to six RPs to switch from one pipeline to another. For each case, bars for different g are shown. g corresponds to the batching size, that is, the number of frames processed by each module before the PR region is reconfigured. Figure 5.17.a shows that two pipelines can be time-shared at 60 fps when one RP is reconfigured per pipeline transition, without batching i.e. $g = 1$. Factoring in reconfiguration time for more than one RP, the time-shared execution of the two pipelines can only keep up when the input is downsampled by $s = 2$, i.e., each pipeline runs at 30 fps (video output at 30 fps is still visually smooth), by $s = 3$, i.e., each pipeline runs at 20 fps, or by $s = 4$, i.e., each pipeline runs at 15 fps. Running the runtime framework at $g = 2$ (two consecutive frames are processed) can restore the frame rate of each pipeline to 60 fps with up to three RP reconfigurations. Factoring in reconfiguration time for more than three RPs, the

time-shared execution of the two pipelines reconfiguration can only keep up when the input is downsampled by $s = 2$, i.e., each pipeline runs at 30 fps.

When time-sharing by three pipelines (Figure 5.17.b), the interleaved execution only keeps up for $g = 1$ when the input is downsampled by $s = 2$ (one RP reconfiguration), i.e., 30 fps, or by $s \geq 3$ (more than one RP reconfiguration), i.e., $\leq 30\ fps$. Increasing g to 3 in this case allows s to be reduced to 2 (except for the last case when six RPs are reconfigured).

Figure 5.18 similarly summarizes the achieved performance measured in frames-per-second when the runtime framework is driven with a 1080p@60fps video stream, and when we execute (a) two pipelines or (b) three pipelines at a time by time-sharing. For 1080p processing, the higher processing time required by two pipelines, without considering reconfiguration time, already would not have fit into the $g = 1$ scheduling quantum of 16.7 ms. In this case, increasing g does not help. Thus, time-shared execution of two pipelines (Figure 5.18.a) requires downsampling by $s \geq 2$, i.e., $\leq 30\ fps$. Time-shared execution of three pipelines (Figure 5.18.b) would require further downsampling to $s \geq 3$, i.e., $\leq 20\ fps$.

5.5 Summary

This chapter discusses the motivations and practical challenges of using PR for computer vision acceleration. Computer vision applications are good candidates for a PR approach since (1) they benefit from FPGA acceleration, (2) they are concerned with efficiency in terms of cost, power and/or energy rather than performance only, and (3) they have slack and can tolerate current PR overhead. When an ASIC-style solution falls short, PR presents a viable alternative to overcome the inflexibility and the resource limitation of an ASIC-style design.

We investigate a very fine-grain PR usage referred to as real-time time-sharing in which multiple pipelines are round-robin executed in the time scale of a camera frame. We discuss the challenges, such as asynchronous module execution, rate mismatch and wasteful reconfigurations, that arise in this new mode of operation. We design and implement a framework including the required architectural/runtime mechanisms to support spatial sharing, coarse-grain temporal sharing (i.e. repurposing) and real-time time-sharing. Real-time time-sharing is a very aggressive usage of PR compared to repurposing, and therefore, requires the development of techniques to mitigate the impact of reconfiguration time on a design's performance.

In our evaluation, we first leverage our framework to implement PR-style designs for repurposing (the scheduling entity is a task). We demonstrate that tasks executed individually within our framework (1)

achieve comparable performance to tasks mapped statically on the FPGA, and (2) are up to two orders of magnitude faster than a CPU implementation. When up to six vision tasks are executed simultaneously within our framework deployed on a Zynq FPGA, we show that they can achieve the same performance as when executed individually (up to 169 MPixels/s). Second, we use our framework to implement PR-style designs for real-time time-sharing (the scheduling entity is a pipeline stage). We demonstrate the feasibility of realizing real-time time-sharing on current FPGAs (up to three time-shared vision pipelines can run at 30+ fps on a Zynq 7000 Series FPGA).

Chapter 6

Quantifying PR Benefits on Case Studies

This chapter presents our quantification of PR benefits for two case studies of implemented designs.

6.1 Overview

Using the framework presented in Chapter 5, we design and implement two application case studies to evaluate the area/device cost, power and energy benefits of PR-style designs relative to ASIC-style designs and software implementations. Both applications have slack since only a subset of tasks is needed at a given time. Also, both applications are deployed on systems with stringent area/device cost, power or energy constraints, and therefore, can potentially benefit from the additional efficiency offered by a PR approach. In the interactive application deployed on an automotive system (Figure 6.1), the tasks needed at a given time are requested by the user depending on the environment (city or highway & day or night). In the navigation application deployed on a robotic system (Figure 6.2), the task needed depends on the number of objects present in the scene and their color. If these applications were mapped statically on an FPGA, resources occupied by the design would not be active all the time.

In our power/energy evaluation, we evaluate the impact of reconfiguration frequency on the PR power/energy overhead. Notably, we investigate whether a PR-style design is still more power/energy efficient than an ASIC-style design when the reconfiguration time accounts for almost half of the total execution time. In

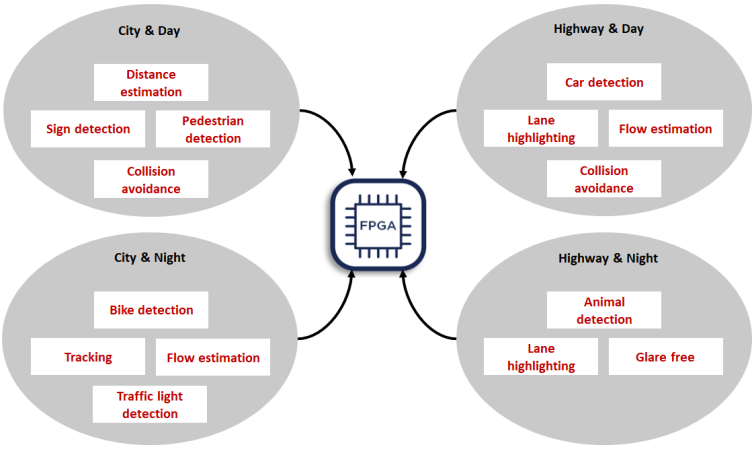

Figure 6.1: Example of an interactive application deployed on an automotive Headlight system [78]. The tasks needed (in red) are requested by the user based on the environment (city or highway & day or night) which *changes infrequently*. Each task is accelerated by one module.

the interactive application, reconfigurations happen infrequently due to the environment not changing frequently.Hence, the reconfiguration time is negligible compared to the compute time. On the other hand, in the navigation application, the task needed may change very frequently (e.g., every tens of milliseconds). Therefore, the reconfiguration time is significant relative to the compute time.

6.2 Implementation

We deploy our framework on the Zynq 706 and the Zynq 702 boards [91] to accelerate the interactive and the navigation application, respectively. A PR-style design provides most benefits if the FPGA has just enough resources to fit the set of concurrent tasks using the highest amount of resources in an application. The Zynq 706 board has an XC7Z045 FPGA which is large enough to fit the set of tasks using the highest amount of resources in the interactive application. The Zynq 702 board has an XC7Z020 FPGA (smaller than the XC7Z045 FPGA) which is large enough to fit the module using the highest amount of resources in the navigation application.

Two ARM CPUs are available on each board. We use one CPU that runs Linux to load the input test images (full HD) into on-board DRAM, and to save the output of vision applications. The other bare-metal CPU executes the runtime system and the application code which specifies the sets of tasks to accelerate in an application. In both applications, the scheduling entity is a task which is accelerated by one module from

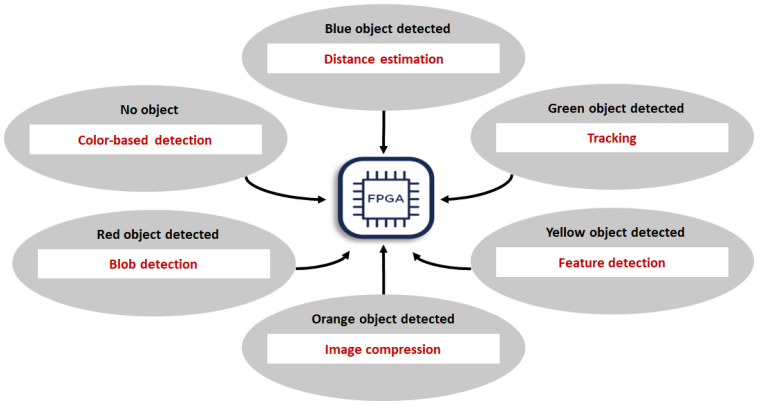

Figure 6.2: Example of a navigation application deployed on a robotic system. The task needed (in red) depends on the objects present in the scene, and may *change very frequently*. Each task is accelerated by one module.

the module library. The module library (the PR bitstreams) is loaded into on-board DRAM at system boot up. The PR regions are reconfigured through the processor configuration access port (PCAP).

6.3 PR-Style Design Characterization

Area Results. To quantify the area benefits of a PR-style design over an ASIC-style design, we first characterize the resource utilization of the PR-style design. Table 6.1 reports the resource utilization of the plug-and-play architecture on the (1) the Zynq 706 board with an XC7Z045 FPGA and (2) the Zynq 702 board with an XC7Z020 FPGA. On the XC7Z045 FPGA, the I/O infrastructure comprises a software configurable crossbar with 16 endpoints and eight DMA engines. We implement four reconfigurable partitions which occupy more than 80% of the FPGA resources. Three reconfigurable partitions have similar size and contain approximately the same amount of logic and memory resources; the fourth partition has approximately three times more resources than the other partitions to fit the largest module needed in the interactive application. When the set of tasks using the highest amount of resources is mapped on the FPGA, the PR-style design (including the I/O infrastructure) consumes 74%, 58%, and 18% of LUT, BRAM, and DSP resources, respectively.

On the smaller XC7Z020 FPGA, the I/O infrastructure comprises two DMA engines. We implement a single reconfigurable partition that is large enough to fit the largest module needed in the navigation application. When the largest module is mapped on the FPGA, the PR-style design (including the I/O

Table 6.1: Resource utilization of the plug-and-play architecture on (1) the Zynq 706 and (2) the Zynq 702 board. The percentage of resource utilization is given in parenthesis. In both cases, most of the FPGA resources are used for compute.

	Zynq 706				**Zynq 702**			
	I/O infrastructure			**Available for compute**	**I/O infrastructure**			**Available for compute**
	Crossbar	**DMA engines**	**Misc**	**Four PR regions**	**Crossbar**	**DMA engines**	**Misc**	**One PR region**
LUT	5580 (3%)	18,270 (9%)	2428 (1%)	185,400 (85%)	0	7388 (14%)	1035 (2%)	26,000 (49%)
BRAM36Kb	0	36 (7%)	7 (1%)	500 (92%)	0	14 (10%)	6 (4%)	80 (57%)
DSP	0	0	0	840 (93%)	0	0	0	120 (55%)

infrastructure) consumes 62%, 33%, and 38% of LUT, BRAM, and DSP resources, respectively.

In both cases, most of the FPGA resources are used for compute. The I/O infrastructure uses less than 16% of logic resources. Note that a similar amount of logic resources would be needed to build the I/O infrastructure in an ASIC-style design.

Individual Module Performance. To accelerate the tasks required in our two applications, we implement the 12 modules shown in Figures 6.1 and 6.2 using Vivado HLS [90]. A significant amount of effort was put in implementing and ensuring the functional correctness of these modules. More information about the modules can be found in [78] and [44].

The 11 modules used in the interactive application are all clocked at 169 MHz in the PR-style design mapped on the Zynq 706 board. The 11 modules achieve a throughput that ranges between 152 and 167 MPixels/s, which is sufficient to achieve 60 fps for full HD frames. Similarly, the six modules used in the navigation application are all clocked at 169 MHz in the PR-style design mapped on the Zynq 702 board.

In our frameworks, the *stereo* and the *flow* modules achieve a throughput of 152 and 161 MPixels/s, respectively. Our CPU implementations of the *stereo* and the *flow*, based on OpenCV and running on a i7-3770@3.40 GHz, achieve a throughput of 13 and 10 MPixels/s, respectively. In [37] and [38], the authors report a throughput of 148 and 165 MPixels/s for their static FPGA designs of the *stereo* and the *flow* modules, respectively, which is comparable to the performance we get.

Repurposing Overhead. The overhead for repurposing the FPGA with a single module is the sum of the time for reprogramming a partition with the module, and configuring the crossbar links and the DMA engines. In practice, the time for configuring the crossbar links and the DMA engines (in the order of

Table 6.2: Resource utilization of six modules used in the interactive application after place & route on the XC7Z045 FPGA. The percentage of resource utilization is given in parenthesis. The six modules are not evenly sized.

	lane highlighting	**sign detection**	**car detection**	**pedestrian detection**	**animal detection**	**bike detection**
LUT	1185 (1%)	16,576 (8%)	84,258 (39%)	80,756 (39%)	81,432 (37 %)	37,122 (17%)
BRAM36Kb	16 (3%)	50 (9%)	104 (19%)	99 (18%)	99 (18%)	59 (11%)
DSP	0	10 (1%)	126 (14%)	156 (17%)	234 (26 %)	70 (8%)

Table 6.3: Resource utilization of six modules used in the navigation application after place & route on the XC7Z020 FPGA. The percentage of resource utilization is given in parenthesis. The six modules are almost evenly sized (in terms of LUT).

	stereo	**flow**	**color-based detection**	**SIFT**	**blob detection**	**Gaussian pyramid**
LUT	18,396 (34.6%)	17,509 (33%)	16,501 (31%)	23,513 (44%)	20,855 (39%)	19,135 (36%)
BRAM36Kb	61 (44%)	60 (43%)	25 (18%)	26.5 (19%)	33 (24%)	47 (34%)
DSP	0	39 (18%)	0	83 (38%)	41 (18%)	55 (25 %)

hundreds of microseconds) is negligible compared to the time spent for reprogramming a partition. At first order, the time overhead for repurposing is equal to the time for reprogramming a partition which is proportional to it size.

In the PR-style design mapped on the Zynq 706 board, the time to reprogram (1) the three smaller partitions is approximately 22.2, 25 and 32.3 ms and (2) the largest partition is 73.4 ms. In the PR-style design mapped on the Zynq 702 board, the time to reprogram a partition is 19.7 ms. Despite the non-trivial time to reconfigure a partition, both applications meet their performance requirements (more details in next section).

In Sections 6.4 and 6.5, we quantify the benefits of PR-style designs mapped on smaller FPGAs over ASIC-style designs mapped on larger FPGAs for the interactive and the navigation application examples. We show that a PR-style design reduces logic resource utilization by 2.5× and 3.2×, device cost by approximately

10× and 4×, and power/energy consumption by 28% and 30% (with impact on cooling cost) compared to an ASIC-style design for the interactive and the navigation application, respectively.

6.4 Case Study 1: Interactive Application

Overview. In this application, the user can pick one of the four sets of tasks shown in Figure 6.1. In each set, up to four tasks are requested simultaneously; each set is accelerated by up to four modules. In the framework, when the FPGA is spatially shared by up to four modules, each module achieves 60 fps for full HD images. In this application, the FPGA is reconfigured with a different set of tasks within minute to hour range depending on the user's selection. *The total reconfiguration time is negligible over the total execution time.* Therefore, the reconfiguration time, power and energy overheads will not be considered for this evaluation.

Area Model. Before presenting our results, we offer a simple model to reason about the potential area benefits of a PR-style design over an ASIC-style design. We consider an application with a total number N_{modules} of modules to accelerate on the FPGA. For this simple model only, we assume that one module is needed at a time. We do not take into account the area utilization of the I/O infrastructure. In the ASIC-style design, all modules are mapped simultaneously on the FPGA; we use the smallest FPGA on which the ASIC-style design fits. For the PR-style design, we use the smallest FPGA that fits the largest module requiring the highest amount of resources in the application.

We define $R_{\text{S/D}}$ as the ratio of the amount of resources available on the FPGA used for the ASIC-style design to the amount of resources available on the FPGA used for the PR-style design. A larger ratio represents a greater saving. In the best-case scenario, all modules are evenly sized (i.e. they consume approximately the same amount of resources). In this case, $R_{\text{S/D-bestcase}} = N_{\text{modules}}$. In the worst-case scenario, modules are not evenly sized; the largest module is significantly larger than all other modules combined. In this case, $R_{\text{S/D-worstcase}} = 1$. Finally,

$$1 < R_{\text{S/D}} \leq N_{\text{modules}}$$

In the interactive application, $R_{\text{S/D}}$ is close to the worst-case scenario since modules are not evenly sized (Table 6.2). In the navigation application, even though modules are evenly sized (Table 6.3), $R_{\text{S/D}}$ is not close to the best-case scenario due to the quantization of FPGA sizes. In practice, using the smallest possible FPGAs to map the ASIC-style and the PR-style designs may not be feasible.

Table 6.4: Resource utilization of the ASIC-style design after place & route on the XC7VH870T FPGA and on the XC7Z035 FPGA for the interactive and navigation application, respectively. The resource utilization percentage is given in parenthesis. In both cases, the ASIC-style design consumes more resources than available on the FPGA used for the PR-style design.

	interactive			**navigation**		
	I/O infrastructure only	**Modules only**	**Total**	**I/O infrastructure only**	**Modules only**	**Total**
LUT	36,278 (7%)	449,026 (82%)	485,304 (89%)	8423 (5%)	115,899 (67.5%)	126,322 (72.5%)
BRAM36Kb	43 (3%)	835 (59%)	878 (62%)	20 (2%)	252.5 (28%)	272.5 (30%)
DSP	0 (0%)	770 (29%)	770 (29%)	0	218 (43.6%)	218 (43.6%)

ASIC-Style Design Area. Since no FPGA is large enough in the Zynq family to map the ASIC-style design, we use the XC7VH870T FPGA from the Virtex 7 family which is the smallest FPGA on which the ASIC-style design fits. The XC7VH870T FPGA uses the same process node (28 nm) as the XC7Z045 FPGA. The ASIC-style design maps 13 modules to accelerate all tasks shown in Figure 6.1, and has a software configurable crossbar with 25 endpoints, eight DMAs and additional control logic for enabling/disabling module combinations. We clock the design at 169MHz (same frequency as in the PR-style design). Table 6.4 shows the resource and percentage utilization of the ASIC-style design after place & route on the XC7VH870T FPGA. The XC7VH870T FPGA has 2.5× more LUT and BRAM resources, and 2.8× more DSP blocks than the XC7Z045 FPGA. The XC7VH870T FPGA is approximately 10× more expensive than the XC7Z045 FPGA.

Power Results. We estimate the power consumed by the ASIC-style and by the PR-style designs when one set of tasks is active at a time (Figure 6.1). We use the Xilinx Power Estimation tool [87] for estimating the power consumption of both designs based on their resource utilization after place & route assuming (1) nominal voltage, (2) same switching activity and frequency, and (3) clock gating the inactive part of the ASIC-style design. Maximum effort for clock gating is applied so that amount of non-leakage power consumed by the inactive part of the design is minimized. (We perform many power measurements on the actual Zynq 706 board when the design is running in steady-state, calibrate the model with our measurements, and find that our power estimations match with the power measurements within 3%.)

Figure 6.3 reports our estimations of the total power consumed by (1) the ASIC-style design and (2) the PR-style design for each set of tasks with corresponding error bars. The power consumption is broken down

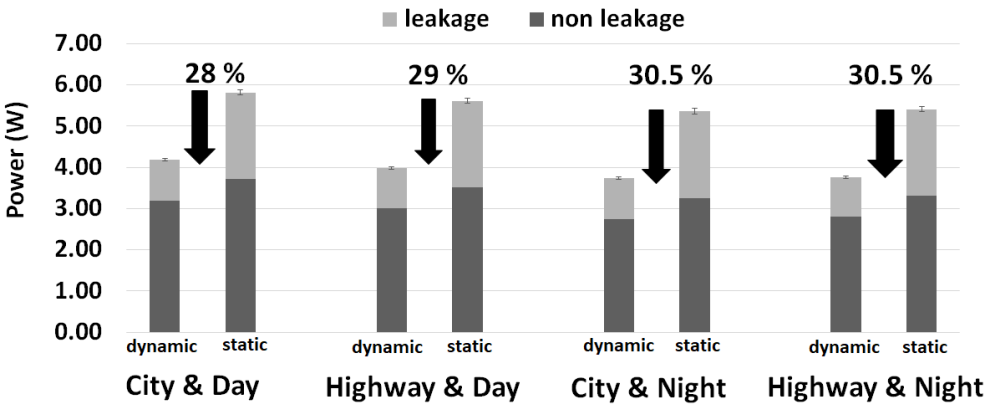

Figure 6.3: Power consumed by the (1) PR-style design labeled dynamic and (2) ASIC-style design labeled static in four environments shown in Figure 6.1. The power savings mainly result from a reduction of leakage power.

into the leakage and non-leakage power. The ASIC-style design consumes 28%, 29%, 30.5%, and 30.5% more total power than the PR-style design for the *City & Day*, *Highway & Day*, *City & Night*, and *Highway & Night* environment, respectively. The ASIC-style design consumes more power/energy than the PR-style design for two reasons: (1) the larger FPGA dissipates more leakage power than the smaller FPGA and (2) the ASIC-style design is larger than the PR-style design and uses more clocking resources, resulting in extra non-leakage power consumed. On average, the ASIC-style design dissipates 50% more leakage power and consumes 14.5% more non-leakage power than the PR-style design.

6.5 Case Study 2: Navigation Application

Overview. We build a navigation application on top of the framework running on the Zynq 702 board (Figure 6.2). Figure 6.4 illustrates the execution of this application during a fixed time interval for the ASIC-style and the PR-style designs. In the reference ASIC-style design (Figure 6.4 (a)), the system monitors the scene for changes. When an event happens (i.e. detection of a colored object), the system processes this event (by executing one of the five possible tasks) and then returns to its monitoring state. In the PR-style design (Figure 6.4 (b)), when an event happens, an idle phase starts. The active module is turned off before starting the reconfiguration of the partition with one of the five possible modules. The idle phase ends when the partition is reconfigured. In this application, an event needs to be processed in less than 34 ms. Though the ASIC-style design can keep monitoring the scene for changes while processing an event, the PR-style

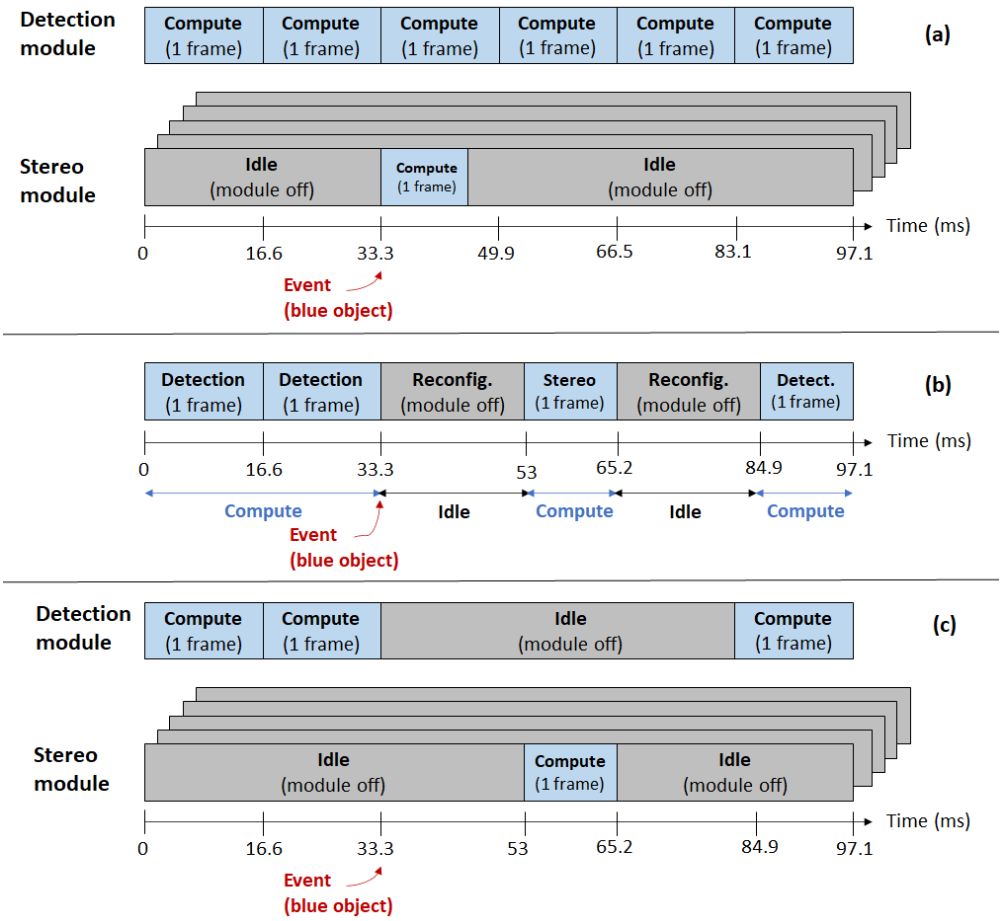

Figure 6.4: Example execution timeline of the navigation application for (a) the ASIC-style design (reference case), (b) the PR-style design, and (c) the ASIC-style design (equalized case).

design meets the application requirement (the time to reconfigure a partition is 19.7 ms and the average time to process an event is 12.2 ms).

Depending on the number of objects to detect in the scene, *the FPGA may be reconfigured very frequently.* The total reconfiguration time may not be negligible compared to the total execution time. In the worst-case scenario, the ratio of reconfiguration to compute is 1:1. In this evaluation, we take into account the total energy spent for reconfiguration by the PR-style design as a function of the number of reconfigurations per second. For a fair comparison, we also consider the conservative case where the compute time is identical in both the ASIC-style and the PR-style designs (Figure 6.4 (c)).

ASIC-Style Design Area. We use the XC7Z035 FPGA to map the ASIC-style design since it is the smallest FPGA from the same family as the XC7Z020 FPGA on which the ASIC-style design fits. The ASIC-style

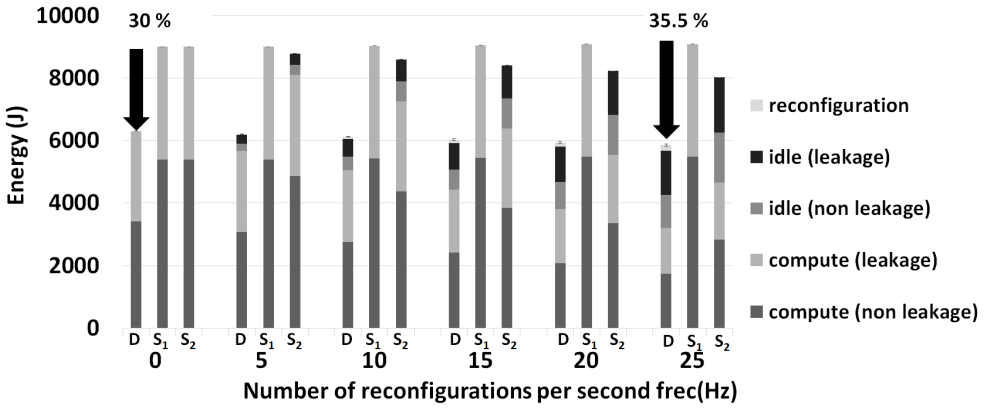

Figure 6.5: Total energy consumed by the PR-style design (D) and the ASIC-style design (in the reference case (S_1), in the equalized case (S_2)) vs f_{rec} ($t_{\text{interval}} = 3600$ s).

design maps six modules used in this application, has two DMA engines and a software configurable crossbar with eight endpoints to connect the six modules and the DMA engines. We clock the design at 169MHz (same frequency as in the PR-style design). Table 6.4 shows the resource and percentage utilization of the ASIC-style design after place & route on the XC7Z035 FPGA. The XC7Z035 FPGA has 3.2×, 3.6× and 4× more LUT, BRAM and DSP resources, respectively, than the XC7Z020 FPGA. The XC7Z035 FPGA is approximately 4× more expensive than the XC7Z020 FPGA.

Energy Model. We develop an energy model to breakdown the total energy into relevant components for the ASIC-style and the PR-style designs. The model considers the energy expended during a fixed time interval of length t_{interval}. In this time interval, we enforce that the ASIC-style and the PR-style designs process the same number of events.

The total energy consumed by the reference ASIC-style design (i.e. the system is never idle) $E_{\text{total, static reference}}$ during t_{interval} is equal to the total energy spent for compute $E_{\text{compute, static reference}}$.

$$E_{\text{total, static reference}} = E_{\text{compute, static reference}}$$

The total energy consumed by the equalized ASIC-style design (i.e. the system can be idle) $E_{\text{total, static equalized}}$ during t_{interval} has two contributions: (1) the total energy spent for compute $E_{\text{compute, static equalized}}$ and (2) the total energy spent when the design is idle $E_{\text{idle, static equalized}}$.

$$E_{\text{total, static equalized}} = E_{\text{compute, static equalized}} + E_{\text{idle, static equalized}}$$

The total energy consumed by the PR-style design $E_{\text{total, dynamic}}$ during t_{interval} has three contributions:

(1) the total energy spent for compute $E_{\text{compute, dynamic}}$, (2) the total energy spent when the design is idle $E_{\text{idle, dynamic}}$, and (3) the total energy spent for reconfiguration E_{reconfig}.

$$E_{\text{total, dynamic}} = E_{\text{compute, dynamic}} + E_{\text{idle, dynamic}} + E_{\text{reconfig}}$$

E_{reconfig} depends on the number of reconfigurations during the time interval considered and is rewritten as

$$E_{\text{reconfig}} = E_{\text{rec,partition}} \times f_{\text{rec}} \times t_{\text{interval}}$$

$E_{\text{rec,partition}}$ is the energy for reconfiguring a single partition. $f_{\text{rec}} \times t_{\text{interval}}$ is the number of partition reconfigurations during the time interval considered. f_{rec} is the number of reconfigurations per second in Hz.

Energy Results. We use the same methodology as in the first study for the power/energy estimations. Figure 6.5 reports the total energy consumed in Joules (J) during a fixed time interval of length $t_{\text{interval}} = 3600$ s for the (1) PR-style design $E_{\text{total, dynamic}}$, (2) ASIC-style design (reference case) $E_{\text{total, static reference}}$, and (3) ASIC-style design (equalized case) $E_{\text{total, static equalized}}$ for different number of reconfigurations per second f_{rec} (in Hz). We report the total energy spent (leakage and non-leakage) (1) for compute and (2) when the design is idle. We also report the total reconfiguration energy E_{reconfig}. When no reconfiguration happens, $E_{\text{total, static reference}}$ and $E_{\text{total, static equalized}}$ are greater than $E_{\text{total, dynamic}}$ since (1) the ASIC-style design dissipates more leakage power than the PR-style design due to the use of a larger FPGA and (2) the ASIC-style design uses more clocking resources than the PR-style design resulting in a higher non-leakage power consumption.

As f_{rec} increases, we observe that both $E_{\text{total, static reference}}$ and $E_{\text{total, static equalized}}$ remain greater than $E_{\text{total, dynamic}}$ even when the compute to reconfiguration ratio is almost 1:1 (i.e. for $f_{\text{rec}} = 25$ Hz, $t_{\text{compute}} = 1827$ s and $t_{\text{rec}} = 1773$ s). Table 6.5 breaks down the total energy consumed by the ASIC-Style and PR-style designs for $f_{\text{rec}} = 25$ Hz. We observe that E_{reconfig} is very small compared to $E_{\text{compute, dynamic}}$ and $E_{\text{idle, dynamic}}$ even for a large number of reconfigurations (the total number of reconfigurations is 90,000). The PR-style design consumes 30% and 35.5% less total energy than the ASIC-style reference design in the best-case scenario ($f_{\text{rec}} = 0$ Hz) and in the worst-case scenario ($f_{\text{rec}} = 25$ Hz), respectively (the modules used for event processing consume slightly more power/energy than the detection module). The PR-style design consumes 30% and 26% less total energy than the ASIC-style equalized design in the best-case scenario ($f_{\text{rec}} = 0$ Hz) and in the worst-case scenario ($f_{\text{rec}} = 25$ Hz), respectively.

Table 6.5: Energy breakdown when $t_{\text{interval}} = 3600$ s and $f_{\text{rec}} = 25$ Hz. When the ratio of reconfiguration to compute is almost 1:1, $E_{\text{total, dynamic}}$ is smaller than $E_{\text{total, static reference}}$ and $E_{\text{total, static equalized}}$ due to E_{reconfig} being very small.

		dynamic	static reference	static equalized
E_{compute} (J)	non-leakage	1736	5491	2741
	leakage	1462	3600	1827
	total	3198	9091	4568
E_{idle} (J)	non-leakage	1064	N/A	1596
	leakage	1418	N/A	1773
	total	2482	N/A	3369
E_{reconfig} (J)	non-leakage	177	N/A	N/A
	leakage	N/A	N/A	N/A
	total	177	N/A	N/A
E_{total} (J)	non-leakage	2977	5491	4337
	leakage	2880	3600	3600
	total	5857	9091	7937

6.6 Summary

Based on the framework presented in Chapter 5, we develop two PR-style designs for applications with slack and quantify their benefits in terms of area/device cost, power and energy relative to software implementations and ASIC-style designs. These applications benefit from dynamic FPGA mapping since (1) all tasks are not needed at the same time, and (2) reducing area, power or energy is as important as meeting performance requirement. In these two studies, we show that a PR-style design mapped on a smaller FPGA results in a reduction in logic resource utilization by 2.5× and 3.2×, device cost by approximately 10× and 4×, and power/energy consumption by 28% and 30% compared to an ASIC-style design mapped on a larger FPGA.

Note that in the second study (navigation application), the ASIC-style design can keep monitoring the environment for changes while processing an event. On the other hand, events may be missed in the PR-style design (the detection module is stopped when an event is processed). Therefore, PR can be only used if the use-case tolerates events' skipping. Also, for our power/energy comparison in the second study, we

could have included a third ASIC-style baseline design clocked at 60.8 MHz (instead of 169 MHz) such that events are processed in exactly 34 ms. By decreasing the clock, we expect the power/energy gap between the ASIC-style and the PR-style designs to reduce.

In the studies presented in this chapter, the clock frequency of the ASIC-style and PR-style designs are identical. Also, in both studies, the data source and sink of the ASIC-style and PR-style designs is the on-board external memory. In the next chapter, we consider the cases where (1) the PR-style design is clocked at a much higher clock frequency than the ASIC-style design to compensate for PR time resulting in a potentially greater amount of non-leakage power dissipated and (2) the input and output data in the ASIC-style design are not stored in external memory. Instead, they are directly streamed in and out of the modules. On the other hand, the module input and output of the PR-style design are stored in external memory. Therefore, the PR-style design incurs additional power and energy overhead for the external memory accesses.

Chapter 7

Projection: Overheads and Benefits of Real-Time Time-Sharing

Motivations. In Chapter 5, we mainly focus on the challenges for realizing real-time time-sharing in practice and demonstrate its feasibility on Zynq-7000 FPGAs. This chapter examines into further details the costs and benefits of real-time time-sharing on current and simulated FPGAs with higher PR speeds. Real-time time-sharing serves as a proxy for very aggressive, fine-grain PR usages that can be highly beneficial in terms of area/device cost but also incur much higher overhead in terms of *time and power/energy* compared to coarse-grain usages of PR. When doing real-time time-sharing, the power/energy overheads in PR-style designs stem from: (1) the very frequent reconfigurations for time-multiplexing, (2) the difference in clock frequency compared to an ASIC-style design to compensate for PR time (refer to Chapters 4 and 5 Sections 4.3.4 and 5.1, respectively, for a more detailed explanation), and (3) the additional data movement required compared to an ASIC-style design to fetch the PR bitstreams and to load/store intermediate data from/to off-chip memory. While we expect an improvement in area/device cost benefits with faster PR speed, in this chapter, we ask whether real-time time-sharing is a more power/energy efficient approach than mapping tasks statically on an FPGA when considering all power/energy overheads aforementioned. To address this question, we conduct a limit study to derive the number of tasks that can be successfully real-time time-shared on a given FPGA, and examine the impact of different PR region sizes, PR speeds and frame resolutions on the number of time-shared tasks. We then estimate the area, device cost, power and energy of PR-style and ASIC-style designs.

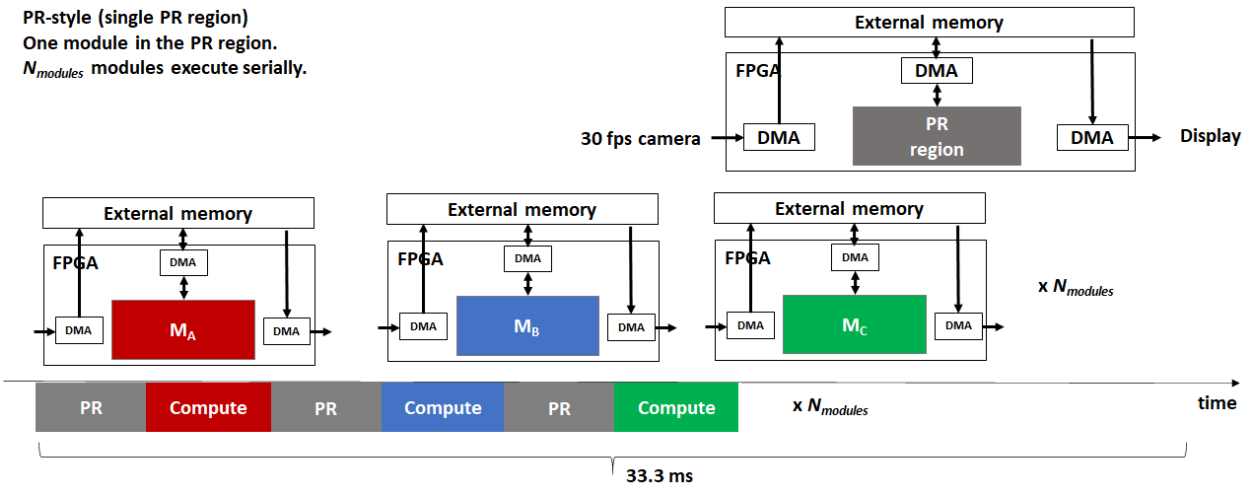

Figure 7.1: The PR-style design has a single PR region on which modules are reconfigured serially. When doing real-time time-sharing, there is additional data movement to load/store intermediate module data from/to external memory.

7.1 Overview

PR-Style Design. In this simulation-based study, we use the simplest designs and setup to understand the impact of PR region size, compute time, and PR speed on the costs and benefits of real-time time-sharing. The PR-style design has a single PR region which is reprogrammed with one module at a time (Figure 7.1). In contrast to Chapter 5 where a module accelerates a pipeline stage, in this study, a module accelerates an entire task or pipeline. Modules are reprogrammed serially on the PR region with no batching allowed.

We assume that a HDMI camera produces data at a fixed frame rate. In this study, we choose a camera rate of 30 fps (i.e. $t_{\text{camera,frame}} = 33.3$ ms) which is typical for many image processing or video analytics applications. Data produced by the camera is stored in on-board external memory. Each module reads an entire input frame from external memory, processes the full frame, and writes the entire output frame to external memory before the PR region is reconfigured with the next module. This reconfiguration-compute loop is repeated for every module. Similar to the setup presented in Chapter 5, we assume that the output of a module is sent to a display and that the PR bitstreams are stored in the on-board external memory. The static I/O infrastructure comprises three DMAs for which the area overhead is accounted for in our results. One DMA is needed to store the camera data to external memory during PR reconfigurations, another to send the output to the display, and a third to transfer the input and output data between the module and

Table 7.1: Specifications for the six PR regions used in this study.

PR region size	vvsmall	vsmall	small	medium	large	vlarge
LUT	6250	12,400	24,960	50,400	100,800	200,160
BRAM36Kb	24	48	96	168	336	636
DSP	48	96	192	336	1176	2040
PR bitstream size (MB)	0.69	1.4	2.8	5.8	9.8	17.9
t_{reconfig} for PR speed=0.45 GB/s	1.5	3.1	6.2	12.8	21.6	39.5
t_{reconfig} for PR speed=1 GB/s	0.69	1.4	2.8	5.8	9.8	17.9
t_{reconfig} for PR speed=2 GB/s	0.35	0.7	1.4	2.9	4.9	9
t_{reconfig} for PR speed=5 GB/s	0.14	0.28	0.56	1.16	1.96	3.58

the on-board external memory.

We make the following assumptions to further simplify the study: (1) all modules have the same size as the PR region, that is, modules use exactly the same amount of logic resources as available in the PR region. (2) All modules operate at the maximum possible frequency in the PR-style design to compensate for the reconfiguration time and to real-time time-share as many tasks as possible in 33.3 ms. (3) All modules consume and produce one pixel per clock in steady-state. The time to process a frame is identical for all modules even if differently-sized. In other words, larger modules are not faster than smaller ones.

Condition for Successful Real-Time Time-Sharing. For a number N_{modules} of modules to be successfully real-time time-shared on the FPGA, all modules are programmed on the FPGA once and process one input frame in less than $T_{\text{camera, frame}}$. We define t_{compute} as the time needed by one module to process one frame (t_{compute} is identical for all modules). t_{reconfig} is the time to reconfigure the PR region with one module. For successful real-time time-sharing, the following condition needs to be satisfied

$$T_{\text{camera, frame}} >= N_{\text{modules}} * (t_{\text{compute}} + t_{\text{reconfig}}) \tag{7.1}$$

Therefore, the maximum number of tasks that can be real-time time-shared during the time interval of $T_{\text{camera, frame}}$ is

$$N_{\text{modules}} = \lfloor T_{\text{camera, frame}} / (t_{\text{compute}} + t_{\text{reconfig}}) \rfloor \tag{7.2}$$

Table 7.2: Module compute time and external memory bandwidth requirements (read and write) in PR-style designs clocked at 300 MHz when doing real-time time-sharing at 30 fps for different frame resolutions.

Frame resolution	Number of pixels	t_{compute} (ms)	External memory bandwidth requirement (MB/s)
480p	450,450	1.5	54
720p	1,237,500	4.1	148.5
1080p	2,475,000	8.3	297

Study Parameters. We vary the size of the PR region and the PR speed to better understand the impact of t_{reconfig} on real-time time-sharing. We also consider different frame resolutions to evaluate the impact of t_{compute} on the number of real-time time-shared tasks. Table 7.1 shows the logic resources, the uncompressed PR bitstream size and t_{reconfig} for different PR speeds for the six PR region sizes used in this study. We consider four PR speeds: 0.45 GB/s, 1 GB/s, 2 GB/s and 5 GB/s. 0.45 GB/s corresponds to the PR speed achieved on current FPGAs (Zynq Ultrascale+ device family) while speeds above 1 GB/s are projected speeds (from conservative to very ambitious).

In this study, another parameter is the maximum clock frequency at which modules can be clocked at. On current FPGAs, computer vision processing modules can typically be clocked at a maximum frequency ranging between 200 to 400 MHz depending on the computation accelerated and on the target FPGA (about 200 MHz on 28 nm Zynq 7000 FPGAs and about 400 MHz on 20 nm Zynq Ultrascale+ FPGAs based on implemented vision modules). On larger recent FPGAs (e.g., Intel Stratix 10), the maximum clock frequency can even be higher. For this study, we assume that all modules can be clocked at a maximum frequency of 300 MHz which represents a middle ground for Zynq 7000 and Zynq Ultrascale+ devices (i.e. PR-style designs operate at 300 MHz). Table 7.2 reports the module compute time t_{compute} when modules are clocked at 300 MHz in the PR-style design and the external memory bandwidth requirement for read and write (assuming two bytes per pixel) to do real-time time-sharing at 30 fps for different frame resolutions.

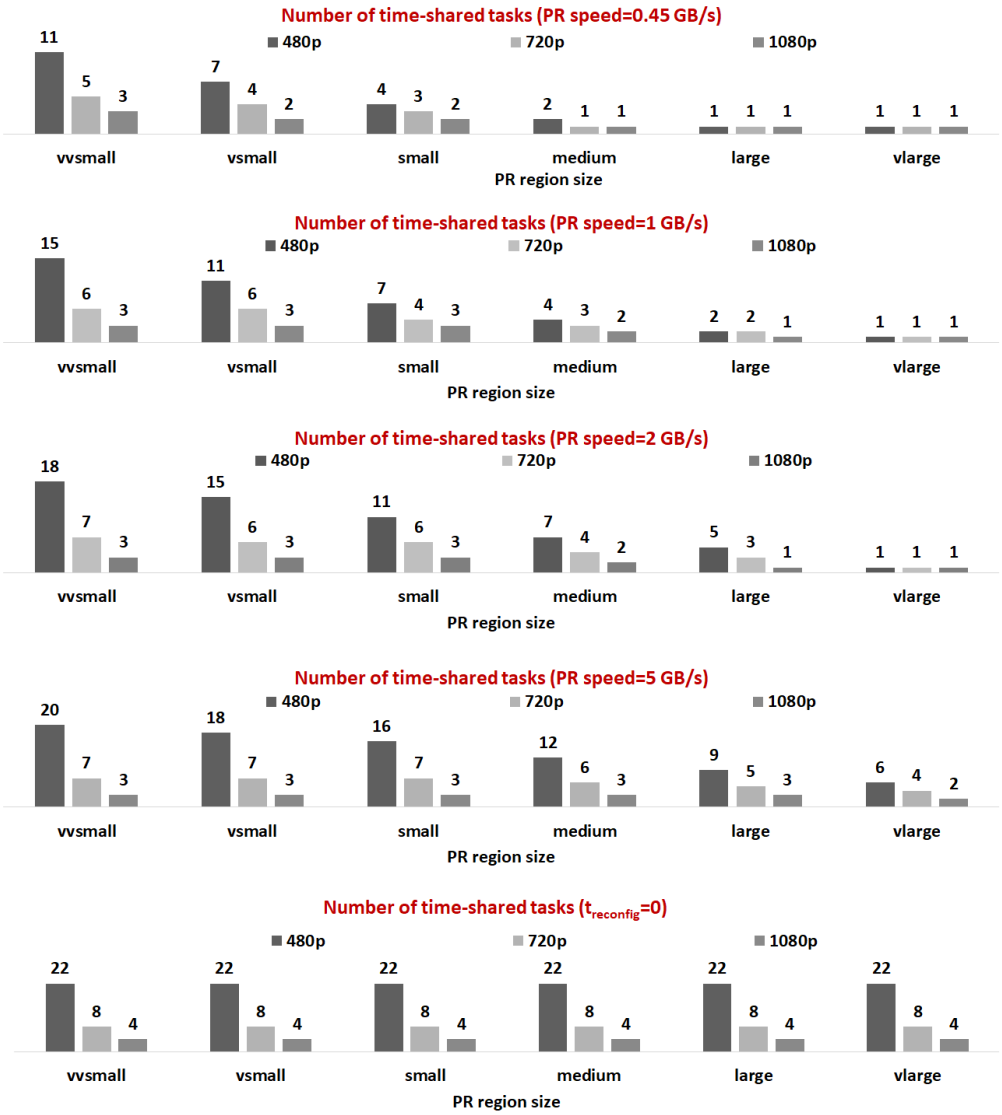

Figure 7.2: Number of real-time time-shared tasks at 30 fps for different PR region sizes, frame resolutions and PR speeds.

7.2 Real-Time Time-Shared Tasks Results

Figure 7.2 shows the number N_{modules} of tasks that can be successfully real-time time-shared for different PR region sizes, frame resolutions, and PR speeds. We also report an upper bound on the number of tasks that could be time-shared if PR speed is infinite i.e. $t_{\text{reconfig}} = 0$. As expected, we observe that (1) for a given frame resolution and PR speed, the bigger the PR region, the larger t_{reconfig}, and therefore, the less the number of tasks to time-share, (2) with faster PR speeds, t_{reconfig} decreases, and therefore, more tasks

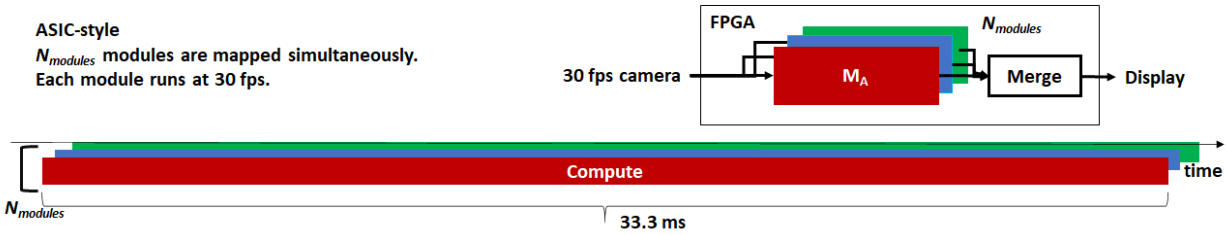

Figure 7.3: In the ASIC-style design, all modules are mapped and executed simultaneously. Compared to the PR-style design, there is no data movement between external memory and the modules: data produced by the camera is directly streamed to the modules for processing; module outputs are directly sent to the display.

can be time-shared. At 5 GB/s, the number of tasks is within 72% of the maximum task number (when $t_{\text{reconfig}} = 0$) for PR regions of size less than small. (3) For a given PR region size and PR speed, the greater the frame resolution (i.e. larger t_{compute}), the less the number of time-shared tasks. Note that with current PR speed (0.45 GB/s), it is possible to time-share up to 11 tasks accelerated by smaller modules in the time scale of 33.3 ms. For example, vision computations such as stereo vision or optical flow can be accelerated by hardware modules using less logic resources than available in a PR region of size small.

ASIC-Style Design. Next, we evaluate the area, device cost, power and energy of the PR-style and the best-effort ASIC-style designs. Figure 7.3 shows the ASIC-style design used in this study. In the best-effort ASIC-style design, each task is accelerated by a module variant that (1) uses the same amount of logic resources as in the PR region and (2) runs at 30 fps. For a given number of real-time time-shared tasks on the FPGA, all modules are mapped and execute concurrently on the FPGA in the ASIC-style design. For example, if 10 small modules can be real-time time-shared in the PR-style design, the logic resource utilization of the ASIC-style design is equal to 10× the size of a small module since all modules are mapped simultaneously.

For a fair power/energy comparison, the ASIC-style design uses the minimum clock frequency for the module to keep up with the camera's rate (30 fps). Each module accepts and outputs one pixel per clock cycle. Therefore, for a specific frame resolution, we can derive the clock frequency of the ASIC-style design by multiplying the frame resolution by the camera frame rate. The minimum frequency for an ASIC-style to run at 30 fps is 13.5 MHz, 37.5 MHz, and 74.3 MHz for 480p, 720p, and 1080p resolutions, respectively, assuming an HDMI camera, i.e. we account for the blanking intervals.

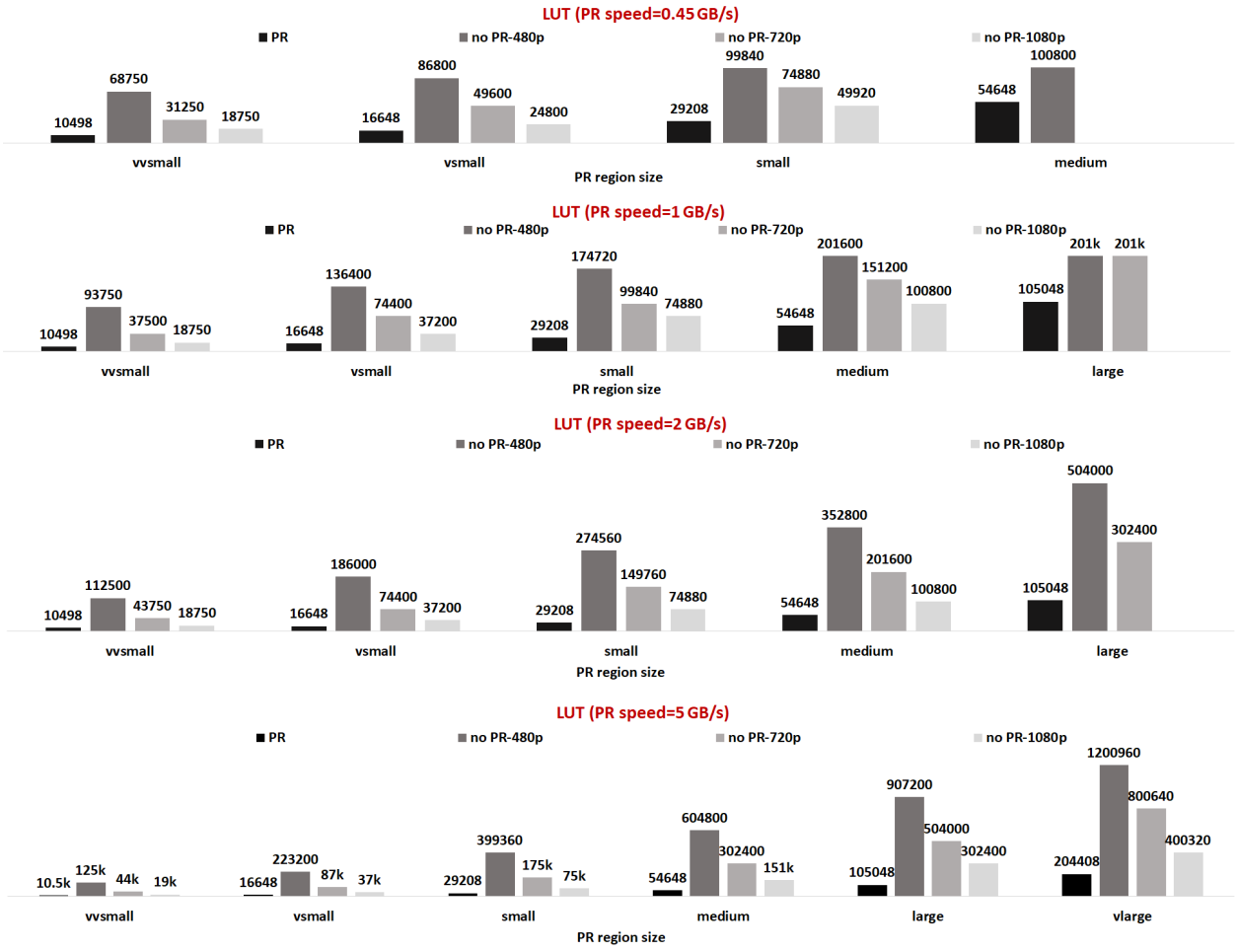

Figure 7.4: LUT utilization in ASIC-style (no PR) and PR-style designs for different PR region sizes, frame resolutions and PR speeds.

Notice that (1) the ASIC-style design has slack since it does not need to be clocked at the maximum possible frequency of 300 MHz to run at 30 fps and (2) there is no data movement between external memory and the fabric in the ASIC-style design in contrast to the PR-style design. In the ASIC-style design, data produced by the input camera is directly streamed to the module. The output from a module is directly streamed to the display.

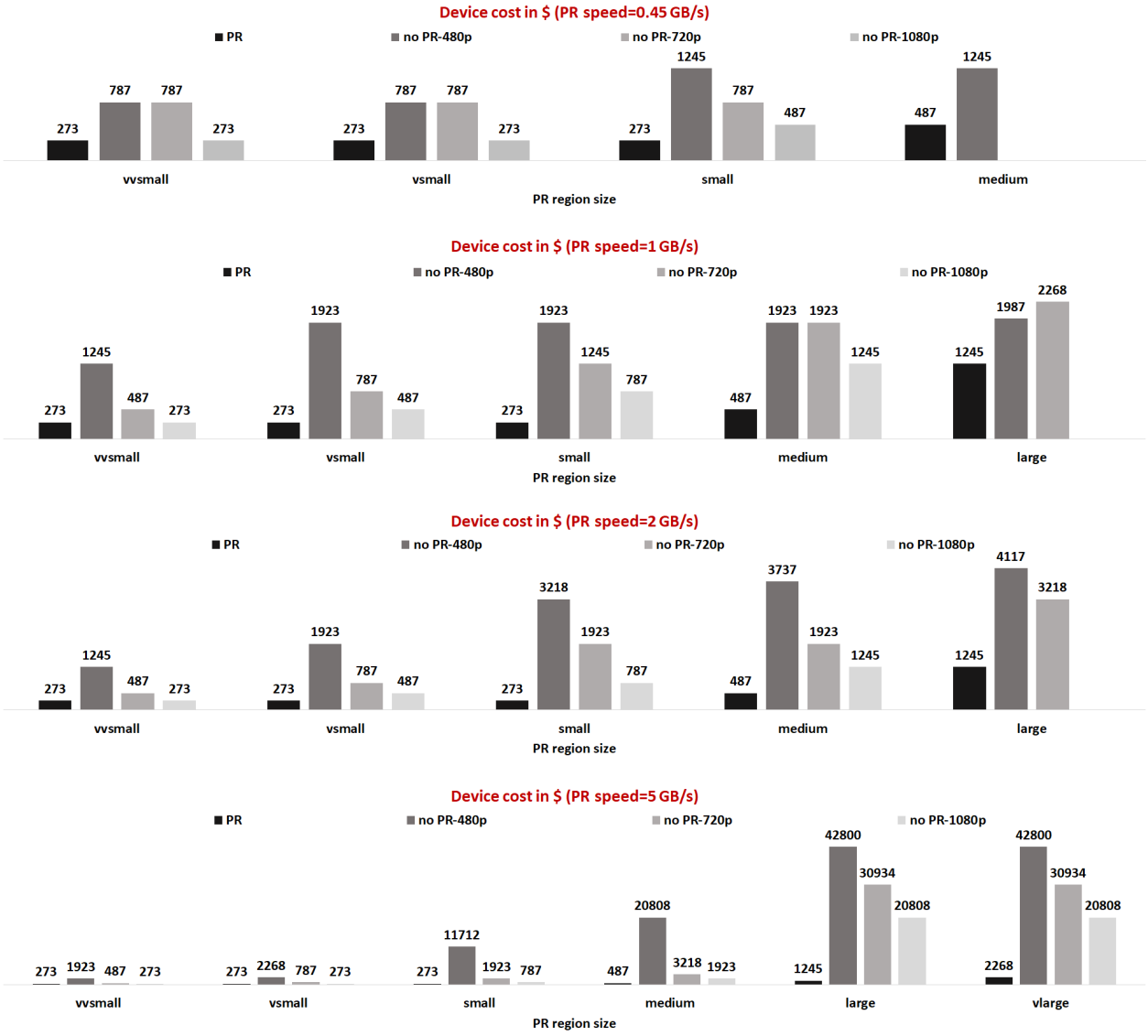

Figure 7.5: Device cost when mapping ASIC-style (no PR) and PR-style designs for different PR region sizes, frame resolutions and PR speeds.

7.3 Area and Device Cost Results

Figure 7.4 shows the number of LUTs used in the PR-style and ASIC-style designs for different PR region sizes, frame resolutions, and PR speeds. In the PR-style design, we account for the LUT utilization of the static I/O infrastructure and the PR region. In the ASIC-style design, the number of LUTs is equal to the product of N_{modules} (equal to the number of tasks that can be time-shared for a specific image resolution,

PR region size, and PR speed) and the LUT utilization of a single module (equal to the number of LUTs available in the PR region). The PR-style design uses roughly a factor N_{modules} less LUTs than the ASIC-style design (same saving factor for on-chip memory and DSP resources). Note that the LUT utilization of the ASIC-style design follows the same trends observed in Figure 7.2: (1) for a given frame resolution and PR speed, the bigger the PR region, the larger t_{reconfig}, and therefore, the less the number of tasks to time-share, and therefore, the smaller the LUT utilization of the ASIC-style design, (2) with faster PR speeds, t_{reconfig} decreases, and therefore, more tasks can be time-shared leading to an increase of the ASIC-style design's LUT utilization. (3) For a given PR region size and PR speed, the greater the frame resolution (i.e. larger t_{compute}), the less the number of time-shared tasks, and therefore, the smaller LUT utilization of the ASIC-style design.

The device cost is based on the amount of logic resources used by a design. Specifically, we find the smallest FPGA part on which a design can be mapped based on its logic resource utilization (LUT, DSP and on-chip memory). Figure 7.5 shows the device cost for the PR-style and ASIC-style designs when mapped on FPGAs from the same FPGA device family (Zynq Ultrascale+ 20-nm FPGAs) whenever possible for different PR region sizes, frame resolutions and PR speeds. For future FPGAs, we assume that device cost scales with area by the same factor as with current FPGAs. We observe that (1) on current FPGAs, a PR-style design can provide up to 4.5× device cost savings compared to an ASIC-style design, and (2) the greater the PR speed, the greater the device cost savings (up to 38× at 5 GB/s).

The area and device cost reduction reported in this study corroborate the findings reported in Chapter 6. Next, we discuss the power/energy overhead of real-time time-sharing including the power/energy overhead for external memory accesses.

7.4 Power and Energy Results

Energy Model. We derive a first-order energy model that breakdowns a design's energy consumption into its different components (similar to the model presented in Chapter 6). The main difference is that we consider the energy overhead to access external memory in the PR-style design for (1) fetching PR bitstreams and (2) loading/storing intermediate data. The model considers the energy consumed by the ASIC-style and PR-style designs during the time interval $T_{\text{camera, frame}}$ (repeated for every video frame). In this time interval, the ASIC-style and PR-style designs perform the same amount of work, that is, each module processes the same frame.

During $T_{\text{camera, frame}}$, the total energy consumed by the ASIC-style design $E_{\text{no PR}}$ is proportional to $T_{\text{camera, frame}}$ and to the average power consumed by the ASIC-style design, which is a function of its frequency and logic resource utilization.

The total energy consumed by the PR-style design $E_{\text{PR, total}}$ has three contributions: (1) the total energy spent for compute E_{compute}, (2) the total energy spent when the design is idle E_{idle}, and (3) the total energy spent for external memory accesses E_{memory} to fetch the PR bitstreams and to load/store intermediate data. We leave the evaluation of the total energy spent by the configuration network and by the PR controller during reconfiguration to future work.

$$E_{\text{PR, total}} = E_{\text{compute}} + E_{\text{idle}} + E_{\text{memory}}$$

E_{compute} is proportional to the total time spent for compute and the average power spent for compute by the PR-style design P_{compute}, which is a function of its frequency and its logic resources. The total time spent for compute is equal to $N_{modules} \times t_{\text{compute}}$.

$$E_{\text{compute}} = N_{\text{modules}} \times t_{\text{compute}} \times P_{\text{compute}}$$

E_{idle} is equal to the product of the total idle time and the average power consumed by PR-style design when a module is idle P_{idle}. The total idle time is equal to $t_{\text{camera, frame}} - N_{modules} \times t_{\text{compute}}$.

$$E_{\text{idle}} = (t_{\text{camera, frame}} - N_{modules} \times t_{\text{compute}}) \times P_{\text{idle}}$$

To estimate E_{memory}, we assume that data can be read/written from on-board external memory (DDR4 32-bit). E_{memory} is the sum of the energy spent for (1) reading the PR bitstreams $E_{\text{bitstream}}$ and (2) for loading/storing intermediate data from/to external memory E_{data}. $E_{\text{bitstream}}$ is equal to the product of the average energy to read/write a 32-bit word from DDR4 and the PR bitstream size. E_{data} is equal to the product of the average energy to read/write a 32-bit word from DDR4 (640 pJ) and the size of the input/output data.

$$E_{\text{memory}} = E_{\text{bitstream}} + E_{\text{data}}$$

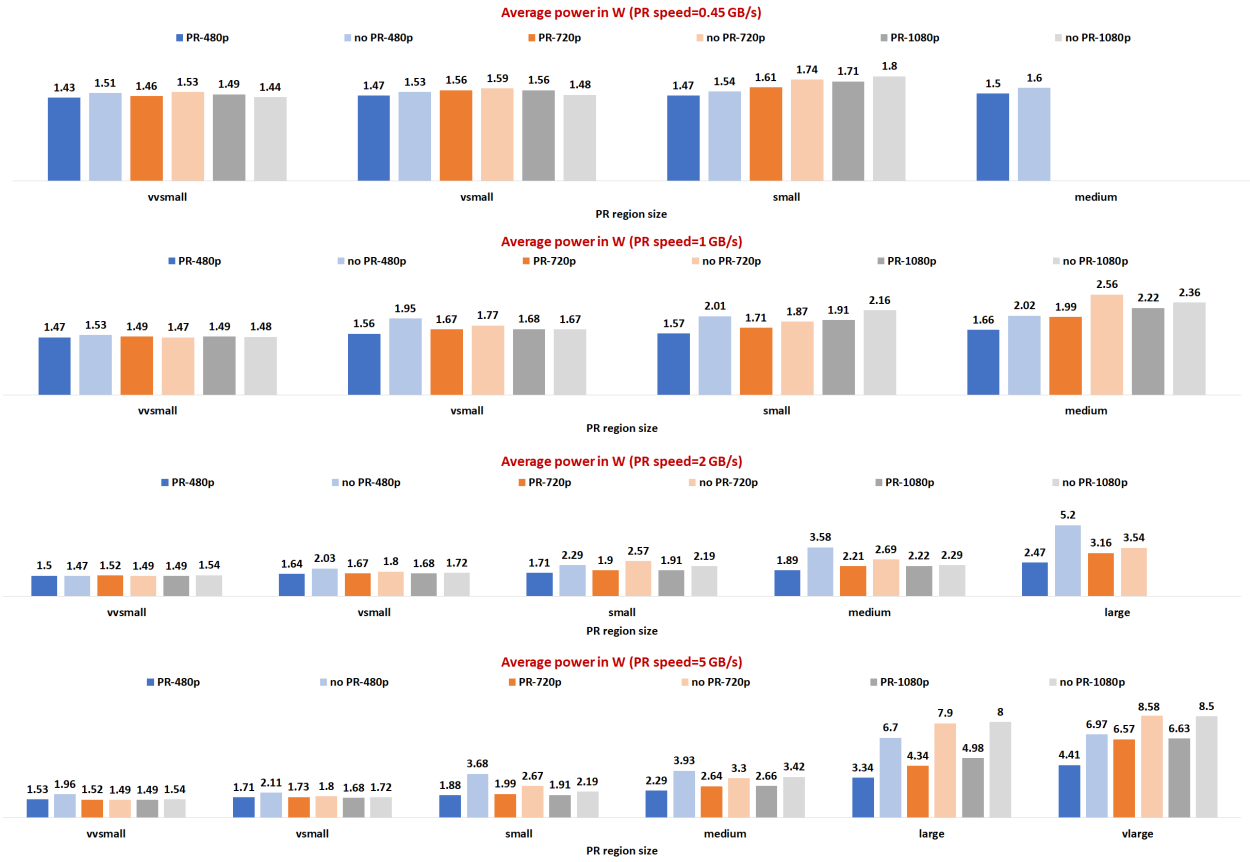

Figure 7.6: Average power of ASIC-style (no PR) and PR-style designs for different PR region sizes, frame resolutions and PR speeds.

No External Memory Overhead. We first discuss the power/energy of PR-style and ASIC-style designs assuming no power/energy overhead for external memory accesses in a PR-style design. Figure 7.6 shows the average power of the ASIC-style (no PR) and PR-style designs for different PR region sizes, frame resolutions and PR speeds over a fixed time interval $T_{\text{camera,frame}}$. We estimate a design's power using the Xilinx power estimation tool based on the frequency and logic resource utilization of the design. When not accounting for the power/energy overhead for external memory accesses, the average power of the PR-style design is less than the average power of the ASIC-style design in most cases. Also, the greater the PR speed, the greater the projected power savings. A PR-style design is projected to save up to 7.4% and up to 50% power when PR speed=0.45 GB/s and PR speed=5 GB/s, respectively, compared to an ASIC-style design. This is due to the PR-style design being mapped on a smaller FPGA which dissipates less leakage power than a larger FPGA. If the PR-style design was mapped on the same FPGA as the ASIC-style design, the power consumed

by the PR-style design would be higher than the power consumed by the ASIC-style design. Even though the PR-style design uses less logic resources than the ASIC-style design, it runs at a much higher frequency (300 MHz) than the ASIC-style design (between 13.5 MHz and 74.5 MHz). When mapped on the same FPGA, a smaller design clocked at a higher frequency consumes more power than a larger design clocked at a lower clock frequency due to the greater power consumed by the clock circuit in a design operating at a higher frequency.

At 1080p, the average power of a PR-style design is generally greater than the average power of the ASIC-style design except in some cases (e.g., when PR speed=5 GB/s and when PR regions are of size greater than small). At 1080p, the overhead of real-time time-sharing outweighs its benefits. In this case, the overhead is due to clocking the PR-style design at 300 MHz, which is about 4× the clock frequency of the ASIC-style design (74.5 MHz), while the number of time-shared tasks is less than 3 in all cases.

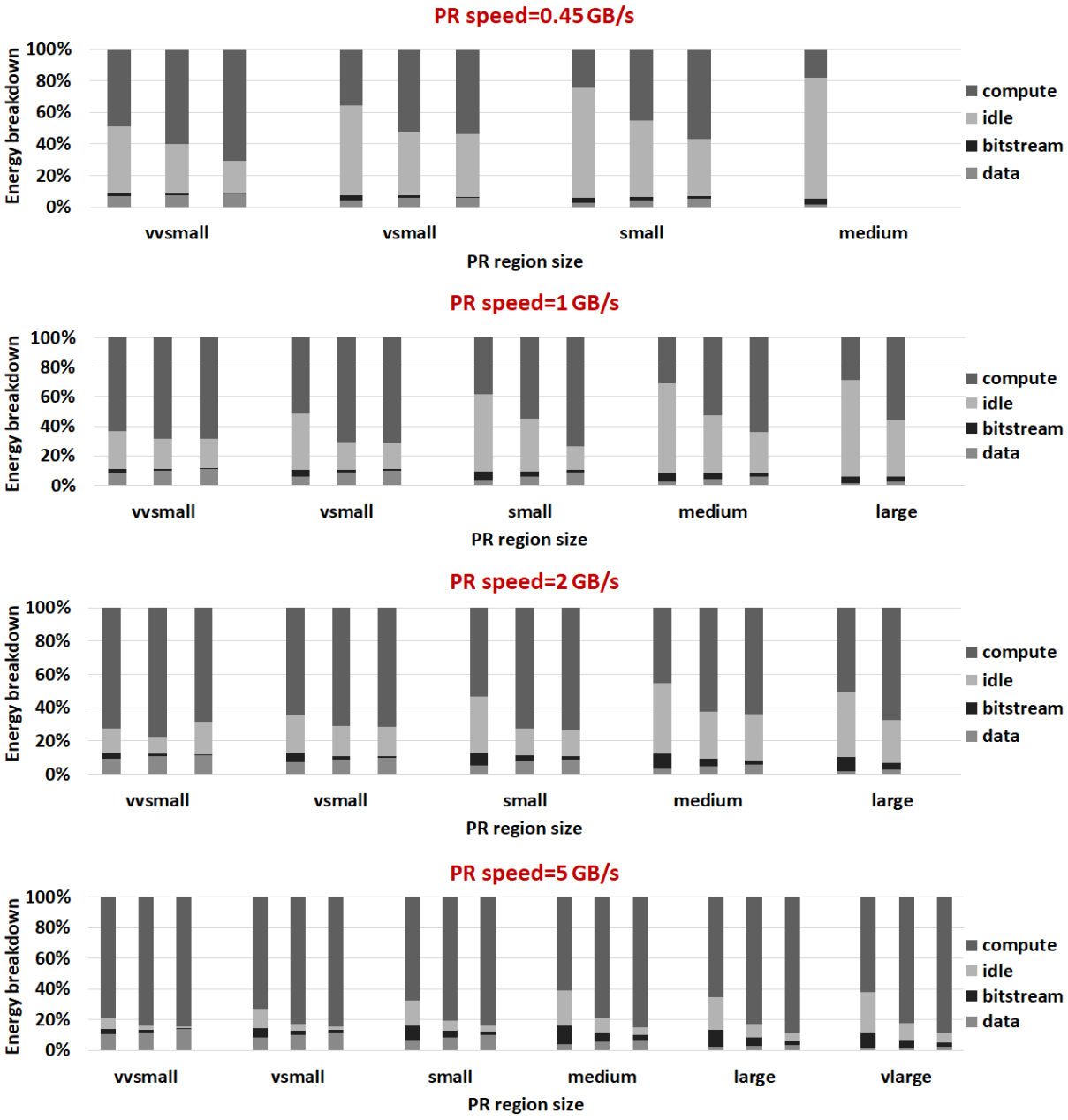

Figure 7.7: Energy breakdown for PR-style designs.

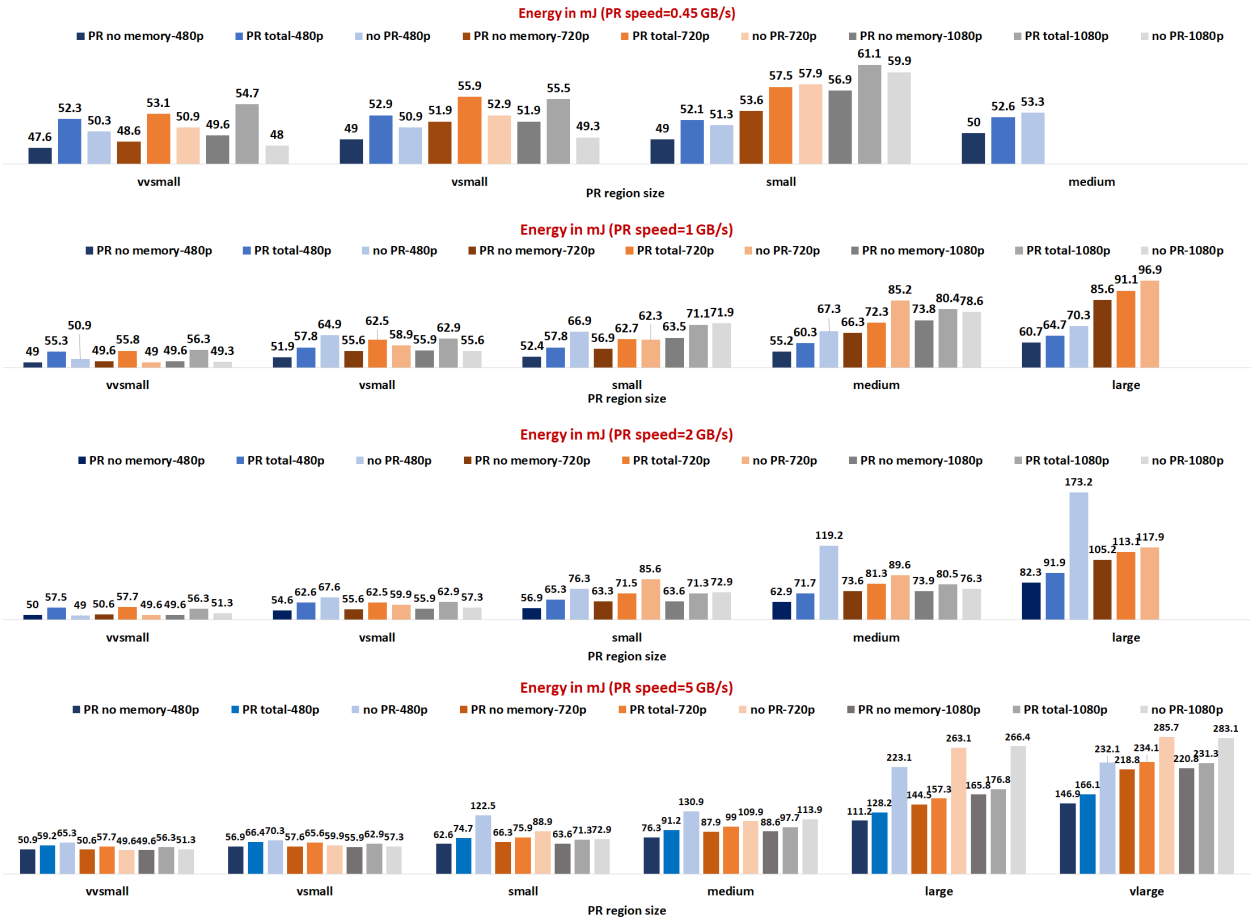

Figure 7.8: Average energy of ASIC-style (no PR) and PR-style designs for different PR region sizes, frame resolutions and PR speeds.

Including External Memory Overhead. We next discuss the energy results when accounting for the energy overhead for external memory accesses in PR-style designs. Figure 7.7 shows the energy breakdown for $E_{\text{PR, total}}$ in terms of (1) compute, (2) a module being idle and (3) external memory accesses for different frame resolutions and PR speeds. The energy for external memory transfers (to fetch PR bitstreams and load/store input output data) accounts for between about 10% and 17% of $E_{\text{PR, total}}$ for PR speed=0.45 GB/s and PR speed=5 GB/s, respectively. At higher PR speeds, more tasks are time-shared, and more external memory accesses occur resulting in a greater energy overhead for external memory access transfers. Note also that a PR-style design may be more power/energy efficient than an ASIC-style design even if the PR-style design spends more time for reconfiguration than for compute.

Figure 7.8 shows the energy consumed by the ASIC-style ($E_{\text{no PR}}$) and the PR-style designs for different

PR region sizes, frame resolutions and PR speeds during $t_{\text{camera, frame}}$. The bars with the label 'PR no memory' correspond to the energy consumed by the PR-style design assuming that the energy overhead for external memory accesses is zero i.e. $E_{\text{memory}} = 0$. The bars with the label 'PR total' correspond to the total energy consumed by the PR-style design ($E_{\text{PR, total}}$).

When accounting for the energy overhead for external memory accesses (i.e. E_{memory} not equal to 0), $E_{\text{PR, total}}$ is generally greater than $E_{\text{no PR}}$ when the PR region's size is less than vsmall for all PR speeds and all resolutions. Therefore, when modules are small (less than 12k LUTs), it is generally more power/energy efficient to map all modules statically than to do real-time time-sharing. When the PR region's size is greater than vsmall, $E_{\text{PR, total}}$ is less than $E_{\text{no PR}}$ in most cases (for all PR speeds, and for 480p and 720p resolutions). Having faster PR is beneficial for power/energy efficiency since the greater the number of large time-shared modules, the larger the FPGA used to map the ASIC-style design. The larger the FPGA, the greater the leakage power dissipated. When PR speed=2 GB/s, up to 47% energy savings can be realized with a PR-style design. In this study, we did not account for the energy for reconfiguration (consumed by the PR controller and by the configuration network). Taking this reconfiguration energy overhead into account is an important step towards a better understanding of the cost/benefits of having higher PR speed.

7.5 Summary

Corroborating the findings in Chapter 6, we show that on current FPGAs with PR speed of approximately 0.5 GB/s, PR-style designs allow to reduce logic resource utilization, device cost, and power/energy by up to 11×, up to 4.5× and by 1.3%, respectively, compared to an ASIC-style design when doing real-time time-sharing. The greater the PR speed, the greater the projected savings in logic resource, device cost, power and energy (up to 22×, 42.7×, and 50% for logic resource, device cost, power and energy savings, respectively, when PR speed is equal to 5 GB/s). We also demonstrate that, in some cases, a PR-style design can be less power/energy efficient than an ASIC-style design even with faster PR speed due to the additional power/energy overheads incurred in a PR-style design. For instance, the power/energy overhead to access external memory for additional data movement can account for between 10 and 18% of the total energy of a PR-style design. This finding emphasizes the importance of developing analytical models and of performing preliminary analyses to gain a better understanding of the costs and benefits of a PR-style design prior to implementing a physical solution.

Chapter 8

Related Work

Architectural Improvements to Reduce PR Time. PR technology has been supported for over a decade but remains an under-appreciated capability (e.g., [6,8,21,22,24,29,31,39,51,66,76,80,81]). One of the mainly cited reasons to prevent a more widespread use of PR is the significant reconfiguration time on today's FPGAs. The time to reprogram a PR region (PR time) is mainly a function of the PR bitstream size, the PR interface speed, and the storage medium where the PR bitstream is loaded from (e.g., external DDR). When loading bitstreams from on-board DRAM, the PR speed on Xilinx FPGAs can range between 128 MB/s and 453 MB/s through the PCAP on Zynq 7000 Series and Zynq Ultrascale+ devices, respectively. Therefore, PR time can range between few to tens of milliseconds on current FPGAs. FPGA vendors have introduced architectural features and enhancements to reduce PR time. For instance, Xilinx and Intel have included bitstream compression support in their tool chain to reduce bitstream size, and therefore, PR time. On Xilinx FPGAs, PR speed has been improving from 128 MB/s (Zynq 700 Series FPGAs) to almost 0.5 GB/s (Zynq Ultrascale+ FPGAs), and is announced to be faster in next generation Xilinx FPGAs [28]. Another example includes the Intel Stratix 10 FPGA which is logically divided into multiple regions, called sectors, that can be reconfigured in parallel to reduce reconfiguration time. Compared to a full FPGA reconfiguration, PR allows to reduce reconfiguration time since only a subset of the FPGA resources is reprogrammed. However, the time to reprogram a PR region is three orders of magnitude higher than a context switch on a general-purpose processor (tens of microseconds) due to the difference between software and hardware programmability as explained in Chapter 2.

Role-and-Shell Approach. Commercially, PR has been mainly used in a "role-and-shell" approach ([9,42]) where the FPGA is reconfigured very infrequently (minutes, hours or days between reconfigurations). Most FPGA resources are contained in a single PR region that is enclosed by a static shell region that provides I/O and isolation. Independent designs with different functionalities, or roles, can be loaded as required in the PR region over time (e.g.,[9,42]). Each role design is an ASIC-style design created to use the entire PR region alone, with no consideration for sharing resources or interacting with other role designs. Using PR in the "roll-and-shell-approach" provides multiple benefits over a full FPGA reconfiguration. First, the shell part of the design can remain active while the role part of the design is reconfigured. This is important for use-cases where the shell region contains components that have to remain active all the time (e.g., network, I/O). Second, PR allows to reduce the area needed potentially resulting in device cost savings if the number of role designs is very large and/or each role design uses a significant amount of resources.

FPGA OSes and Virtualization. With the growing interest of deploying FPGAs in the cloud or in the data centers, the amount of work on FPGA OSes ([2,26,47,67,73]) and FPGA virtualization ([7,23,82]) has been increasing quite rapidly. Past works on FPGA OSes have focused primarily on execution models for statically mapped FPGAs and have not focused on PR-based FPGA systems. Example of FPGA OSes that aim to provide analogous OS/runtime services (such as process control and I/O) include BORPH [73], ReconOS [2] and LEAP [26]. They provide high-level management and virtualization of the FPGA environment and resources. LEAP OS provides a set of convenient high-level system services analogous to system services enjoyed by software processes (for example, STDIO library). BORPH carries the analogy further by presenting the FPGA to OS management as a process with analogous process control and I/O interfaces.

Recent works on virtualization leveraging PR mostly target multi-tenancy. In a multi-tenant use-case, the FPGA can be shared by multiple tenants or groups of users. Over time, each tenant can request sets of tasks with different resource and performance requirements. Since the sequence of tasks to be accelerated at runtime is not known at design time, it is challenging to determine (1) a good partitioning of the FPGA, i.e., decide on the optimal number and size of PR regions, and (2) an efficient module-to-PR region dynamic allocation strategy to optimize metrics such as quality-of-service (QoS), resource utilization or serviceability defined as the rate of successful allocation requests. Most works either tackle the partitioning or the allocation problem assuming some fixed allocation or some fixed partitioning strategy, respectively.

Effort tackling the partitioning problem proposed search-based strategies to find the optimal number

and size of PR regions for a given use-case [63]. To support the dynamic requirements of multi-tenancy, [82] proposed an elastic partitioning strategy that can enable either multiple smaller and slower tasks to execute concurrently, or one single fast and larger task to occupy most FPGA resources. Note that the same partitioning and allocation challenges arise in the embedded systems domain and have also been thoroughly investigated [34,53,67].

Allocation and Defragmentation Strategies. A vast body of work focused on the allocation problem in dynamically managed FPGAs (e.g., [32,40,72,75,85]). They mainly discussed the theory of spatial and temporal sharing, mechanisms for preemption [4], or hardware/software co-scheduling based on some fixed partitioning of the FPGA [20,59,75,85]. Allocation heuristics have also been explored to reduce resource fragmentation in dynamically-managed FPGAs [15,25,52]. Internal and external resource fragmentation can arise due to the division of the FPGA fabric into a fixed number of reconfigurable partitions with static boundaries at design time (i.e. fixed amount of resources in each PR region). External fragmentation occurs if the fabric is divided into many small partitions that can not host large modules. Internal fragmentation occurs if we try to make the PR partitions large enough. This reduces the number of modules that can run concurrently; the resources in the large partitions would frequently be wasted on small modules. Examples like [20,25,52] proactively defragment the fabric by relocating modules at runtime. These past work predominantly focused on algorithmic solutions to a formalization of the problem where modules are dealt with as geometrical shapes to be packed into a two-dimensional area that represents the fabric. There is comparably much less work on addressing PR allocation and fragmentation problems under working technology and implementation assumptions (e.g., [50]). Dessouky et al. developed an orthogonal approach to efficiently share BRAM without fragmentation [19]. Their hardware runtime system manages BRAMs centrally as a pooled resource and supports modules with managed virtualized access.

PR to Ease Design Development. We start to see PR being leveraged by domain-specific frameworks to ease the development of FPGA designs for non-FPGA experts. In such frameworks, the FPGA has been divided in multiple PR regions surrounded by a static I/O infrastructure that provides connectivity between the PR regions and the rest of the system (I/O). A module library is built by FPGA experts and contains off-the-self functional and optimized modules that accelerate computation stages. Typically, a non-FPGA expert can specify his/her task to accelerate by using a domain-specific language and provided APIs. The user code gets parsed, and a runtime system is in charge of mapping the computation on the FPGA. The runtime manager is responsible for sequencing the computation and for choosing the appropriate module variants

to be reprogrammed in the PR regions. For instance, [70] proposed a framework to ease the development of FPGA accelerators for data analytics applications. A data analyst or data scientist can specify his/her computation graph using a Big Data Environment (e.g., Apache Spark). The required task is then mapped and accelerated on the FPGA.

Fine-Grain PR Usages. Few academic projects have attempted to use PR at a fine-grain, i.e. the time interval between reconfigurations is within the sub-second range, due to the non-trivial PR time on today's FPGAs. When using PR at a fine-grain, the entity to schedule is typically a processing stage of a task, resulting in smaller PR regions, and therefore reduced PR time. For many of these works, the sequence of modules to execute is known ahead of time and therefore, the schedule can be determined offline. The work in [83] leverages PR to implement streaming pipelines for convolutional neural network (CNN) acceleration. If the pipeline was mapped entirely, all resources occupied by the design would not be active all the time due to, for instance, the lack of off-chip memory bandwidth (presence of slack). Instead of mapping the entire pipeline on the FPGA, pipeline stages are reconfigured over time, one or multiple at a time. The authors show that a PR-style design can achieve greater throughput than an ASIC-style design given the same area budget for CNN acceleration. In other words, they show improved performance density, defined as throughput per LUT in their case, due to a better utilization of available resources when using PR. To amortize PR time, they used techniques such as batching, that is, multiple inputs are processed by each pipeline stage before the stage is reconfigured.

Benefits of PR. PR has been used to dynamically reuse, adapt or customize the datapath over the same fabric resources to improve performance without incurring additional cost in fabric area. Along the line of our vision framework introduced in Chapter 5, past systems that divided the fabric and managed its use as PR regions include [18,55,58]. For domain-specific applications that can tolerate current PR time, prior works demonstrated the potential benefits of a PR-style design in terms of area/device cost [3,13,51, 76,83], power/energy efficiency [62,64]. Rather than maximizing performance, the design objective for these applications is to achieve greater area (power/energy) efficiency, that is, (1) for a given area (power/energy) budget, the PR-style design achieves higher performance than the ASIC-style design, or that (2) for a given performance target, the PR-style design consumes less area (power/energy) than the ASIC-style design. In our work, we analyze in great details the benefits and overheads incurred when using PR at a fine-grain under different assumptions and use-cases (Chapters 6 and 7).

Techniques and Tools for Better PR Support. While FPGA vendors have steadily improved PR sup-

port in their tool chain (e.g., from script-based PR flow to GUI-based PR flow, APIs to send reconfiguration command to PR controllers), implementing a PR-style design is still cumbersome. Notably, since PR region-specific bitstreams need to be compiled for each module, a potentially very high number of PR bitstreams needs to be managed at runtime for larger systems. Creating partition-specific bitstreams also increases the placement & routing time. Prior works [5,41,53,77] investigated bitstream relocation, a technique that can effectively reduce both the compile time and the number of bitstreams to manage. Bitstream relocation allows a partition-specific bitstream to be modified at runtime so that it can be relocated into different RPs. Enabling bitstream relocation in future tools and FPGAs could help reduce compilation time and runtime management complexity.

Chapter 9

Conclusion

9.1 Summary

To re-iterate, this thesis explores the question of when, how, and why FPGA designers should consider using PR over a traditional ASIC-style approach. *This work identifies slack as the main opportunity that can exploited by PR-style designs to improve upon ASIC-style designs.* The first part of the thesis defines the concept of slack in ASIC-style designs and how PR can reduce this inefficiency with better area-time scheduling (Chapter 3). In ASIC-style designs, FPGA designers commit to a fixed allocation of resources at design time. At runtime, some of the occupied resources can be left idle or under-utilized. An ASIC-style design has slack when resources occupied by the design are not active all the time. Slack can result in (1) the design not running at the desired performance given an area budget, or (2) the design running at the desired performance but being too big to fit in the given area. We discuss different sources of slack such as tasks not needed all the time, operation dependencies or pipeline imbalancing. More subtle forms of slack exist for applications that need to meet a performance target given an area, power or energy budget (e.g., video analytics, image processing applications). For such applications, the ASIC-style design may not be clocked at the maximum possible frequency to achieve the desired performance. A PR-style design can leverage this slack to compensate for PR time and meet required performance.

Using PR, a designer can attempt to reduce slack by changing the allocation of resources over time. We present our PR execution strategies to build area-efficient PR-style designs to (1) maximize performance (latency or throughput) given an area budget or (2) minimize area given a performance bound (Chapter 4). When optimizing for throughput given an area budget, the best strategy is to serialize the execution of

modules on one PR region using the largest module variants possible. Whenever possible, batching allows to amortize PR time given enough intermediate memory capacity. When optimizing for latency given an area budget, two possible execution strategies exist: (1) serializing execution on one PR region and (2) interleaved execution on two PR regions. When attempting to minimize area and meet a performance target, a strategy is to serialize the execution of modules on one PR region using the smallest variants possible. There should be enough slack in the ASIC-style design in the form of sufficient clock frequency to compensate for PR time. Using the proposed methodology and analytical model, a designer can quickly evaluate the best PR execution strategy for a given problem and whether a PR-style design improves upon an ASIC-style design. The model considers the trade-offs between PR region size, PR time and module performance. We also account for the impact of memory bandwidth requirements on module performance. The model can be used in the early stage of design space exploration given a module library that has been characterized in terms of area, latency, throughput, memory bandwidth requirement, etc. The more accurate the characterization, the more accurate the model's predictions. In our evaluation, we validate the accuracy of the model on case studies of implemented designs.

Leveraging the findings from our theoretical analysis, the second part of the thesis presents our practical investigation on using PR to accelerate computer vision applications (Chapter 5). Computer vision applications are good candidates for a PR approach since (1) many vision applications can be accelerated by streaming pipelines proven to greatly benefit from FPGA acceleration, (2) they have slack and can tolerate current PR time (30+ fps performance requirements), and (3) are deployed on systems with area/device cost, power or energy constraints. When an ASIC-style solution falls short due to its resource limitation or inflexibility, PR presents a viable alternative. We design and implement a framework leveraging PR for spatial and temporal sharing of multiple vision tasks. In the framework, multiple tasks can be mapped and executed simultaneously on the FPGA (spatial sharing). Tasks can also be temporally shared at a coarse or fine grain, that is, when the time interval between reconfigurations is within hours or milliseconds, respectively. Unlike most works that mainly leverage PR at a coarse-grain (e.g.,"role-and-shell" approach), we investigate a very aggressive, fine-grain usage of PR referred to as real-time time-sharing where the FPGA fabric is time-multiplexed by multiple pipelines within the time scale of a camera frame. Challenges such as asynchronous module execution, rate mismatch and wasteful reconfigurations arise in this new mode of operation. We present the theory and the techniques for solving these challenges and implement them in the framework. These architectural and runtime strategies mitigate the impact of PR time on performance. For

instance, to reduce the number of reconfigurations needed, we design a software programmable crossbar and leverage the commonality of modules in vision applications. We overlap compute and reconfiguration to hide PR time, and batch to amortize PR time. Note that most of these optimizations are not specific to computer vision and can be used for other application domains. In our evaluation, we first leverage our framework to implement PR-style designs for repurposing (the scheduling entity is a task). We demonstrate that tasks executed individually within our framework (1) achieve comparable performance to tasks mapped statically on the FPGA, and (2) are up to two orders of magnitude faster than a CPU implementation. When up to six vision tasks are executed simultaneously within our framework deployed on a Zynq FPGA, we show that they can achieve the same performance as when executed individually (up to 169 MPixels/s). Second, we use our framework to implement PR-style designs for real-time time-sharing (the scheduling entity is a pipeline stage). We demonstrate the feasibility of realizing real-time time-sharing on current FPGAs (up to three time-shared vision pipelines can run at 30+ fps on a Zynq 7000 Series FPGA).

Using the framework, we also design and implement application examples to demonstrate that a PR-style design can be more area/device cost, power or energy efficient than an ASIC-style design with slack (Chapter 6). We first demonstrate the benefits of a PR-style design on two application studies used as proxies for a rapidly emerging class of AI-driven applications with limited area, power or energy budget. Notably, we show that a PR-style design can be more power and energy efficient than an ASIC-style even when reconfigurations are very frequent (i.e. most of the total execution time is spent for reconfiguration) as long as the application has tolerance for frame drops. Note that in these specific studies, the clock frequency of the ASIC-style and PR-style designs are identical. Also, the data source and sink of the ASIC-style and PR-style designs is the on-board external memory. Therefore, there is no extra power/energy overhead incurred in the PR-style design due to a higher clock frequency and to additional data movement.

We conduct a limit study and make projections on the impact of having higher PR speed on the costs and benefits of a PR-style design when doing real-time time-sharing (Chapter 7). Real-time time-sharing serves as a proxy for very aggressive PR usage that incurs the most overheads in terms of PR time, power and energy due to (1) the very frequent reconfigurations, (2) the additional data movement for fetching PR bitstreams and for loading/storing intermediate module data, and (3) the higher clock frequency of the PR-style design to compensate for PR time. Therefore, in this study, we ask whether PR benefits grow infinitely with greater PR speed or whether the overheads incurred in real-time time-sharing outweigh the benefits of a PR-style design. The main findings of this study are that, with faster PR, more tasks can be time-shared on

a given FPGA, or alternatively, a smaller FPGA can be used for time-multiplexing a fixed number of tasks. In other words, the greater the PR speed, the greater the area and device cost savings. The power/energy overheads for (1) additional data movement to fetch PR bitstreams and to load/store intermediate data from/to off-chip memory, and (2) clocking the design at the maximum achievable frequency can have a significant impact on the design's power/energy. Notably, we show that a PR-style design can be less power/energy efficient than an ASIC-style design. For instance, in a PR-style design with many small PR regions, the energy overhead for additional data movement can account for up to almost 20% of the design's total energy resulting in the PR-style design consuming more energy than the ASIC-style design mapped on a larger FPGA. Another example is with a PR-style design spending most of its total execution time computing while operating at the maximum achievable frequency to compensate for PR time. Compared to an ASIC-style design mapped on a larger FPGA but clocked at a lower frequency, the PR-style design may consume more power/energy. Overall, this study highlights the importance of considering all power/energy overheads incurred in a PR-style design for a fair comparison with ASIC-style designs.

In summary, PR mostly benefits applications with slack that are concerned with efficiency. An ASIC-style design is the best approach when the goal is to solely maximize performance. When using PR at a coarse-grain (e.g., for repurposing), the time and power/energy overhead has negligible impact on the design's performance and power/energy, respectively. A PR approach allows to save area/device cost, power and energy compared to a traditional static approach. When using PR at a fine-grain, a PR-style design is generally more area/device cost efficient than an ASIC-style design, that is, (1) a PR-style design uses less resources while achieving the same performance as an ASIC-style design. A PR-style design can be mapped on a smaller FPGA resulting in device cost reduction. or (2) a PR-style design can achieve better performance using the same amount of resources as an ASIC-style design. When considering power/energy, the power/energy overheads of a PR-style (due to the frequent reconfigurations, the additional data movement, and the higher clock frequency) can outweigh the benefits of using a smaller FPGA. This latter observation on power/energy overheads is also true at higher PR speeds.

9.2 Limitations and Future Work

We discuss some of the limitations of the current work and present possible solutions.

Exploring Different PR execution Strategies for Dynamic Load Scenario. This thesis focuses on offline scheduling strategies assuming that the sets of modules to be accelerated on the FPGA are known at

design time. In a design scenario where this information is not available, the problem of allocating modules to PR regions may involve more complex decisions. For instance, in the data centers or in the cloud, different combinations of tasks with different resource and performance requirements are needed over time and are not known at design time (i.e. dynamic load requirements). Dynamic PR execution strategies can attempt to find the best allocation to optimize quality-of-service and/or maximize utilization. These strategies can involve module preemption, module relocation or even defragmentation of the fabric. When modules are preempted or relocated, additional mechanisms are required for saving/restoring a module's context. Fabric defragmentation is an attempt to improve utilization when resources are internally or externally fragmented.

Investigating Different PR Region Division Strategies. This work assumes, for the most part, that the PR-style design is divided into homogeneously-sized PR regions, (i.e. PR regions roughly contain the same amount of resources) which may not be the optimal division, for instance, if the module library consists of a mix of small and large modules. If the fabric was divided into many small PR regions of the same size, large modules would not fit. If the fabric was divided into fewer large PR regions of similar size, less modules could execute concurrently. In this case, dividing the fabric into multiple heterogeneously-sized PR regions might be the best option. One attempt at finding the optimal layout (i.e. the number and size of PR regions) appeared in [63] leading to improved resource utilization and quality-of-service. We explore the amorphous technique, a non-conventional method that removes the fixed PR region boundary constraint enforced by current design tools and that allows to change the size of PR regions at runtime (Appendix A). Using this technique, the FPGA designer does not need to commit to a fixed layout at design time; the size of PR regions can be changed at runtime. Though the amorphous technique can improve resource utilization and reduce fragmentation in PR-style designs, it incurs additional complexity at design time and at runtime due to an increased number of bitstreams to generate and manage.

Analytical Model Limitations. The analytical model presented in Chapter 4 has several limitations. First, the model does not provide efficient PR execution strategies (1) to minimize a design's power/energy and (2) for mixed design objectives, for instance, maximize throughput while keeping latency smaller than a given threshold. One possible solution for mixed problems is to leverage the proposed PR execution strategies and search the design space for the best design. Second, the estimations' accuracy is based on the ability to place and route modules at the required clock frequency, which can be challenging depending on the problem. The model could include different confidence levels based on a design's complexity. Third, the methodology presented to find an efficient PR-style design currently relies on manual search of the design

space. If the number of module variants is very large, searching exhaustively the design space considering the multiple dimensions of PR time, area, performance, and memory bandwidth requirements would quickly become tedious. A solver could automate this search process, and selects the appropriate modules for a given problem.

Improving the Framework's Usability for Non-FPGA Experts. The framework presented in Chapter 5 was used to facilitate the implementation of PR-style designs for evaluating their benefits. This work has fully not explored the potential of using PR to ease FPGA design development for non-FPGA experts, e.g., computer vision application developers. The main components of the framework could be the foundations of a "design by module composition" approach. A non-FPGA expert can build his/her application to accelerate on the FPGA by specifying the different processing stages and their connectivity. The application code is parsed by a runtime system which reprograms the plug-and-play architecture with the needed modules from a module library. Frameworks using PR to ease FPGA design development have started to emerge, for instance, for data analytics acceleration [70]. A data scientist expert can express the task to accelerate on the FPGA using a Big Data environment such as Apache Spark. The task is modeled as a computation graph accelerated by, for instance, a streaming pipeline on the FPGA. The FPGA fabric is divided into a shell region that provides I/O connectivity between the PR region and external components such as PCIe, DRAM or Ethernet. A dataflow compiler parses the user-level code and generates the intermediate representation passed to a hypervisor which is responsible for mapping the computation on the FPGA. Specifically, the hypervisor picks the appropriate module from a module library to accelerate the required task.

The main benefits of such approach would be to (1) provide functional and optimized off-the-shelf processing modules that can be easily composed to design an application. Similar to software development where optimized APIs are provided for executing common vision processing tasks (e.g., stereo, corner detection), the module library should consist of a rich enough set of vision modules to accelerate standard processing tasks. The module library should consist of parameterizable modules with different variants allowing for different trade-offs (e.g., performance, area, accuracy, power) (2) hide the complexity of I/O connectivity between the FPGA and external I/O components (e.g., external memory, cameras, display) from the user with the plug-and-play architecture.

Among the challenges to be addressed, one is to provide the right abstraction for a non-FPGA expert to interface with the FPGA (e.g., user application code written in domain-specific language), and the intermediate layers needed to move from this user-level application representation to a lower-level representation.

The adoption of a domain-specific language such as Halide could make the framework more vision experts friendly. A compiler could translate the Halide user-level code to an intermediate representation leveraging the proposed APIs and constructs for module-to-PR-region mapping and for establishing the desired connectivity between modules. Also, the complexity of the PR flow should be completely hidden from the user. For instance, the processes of allocating modules to PR regions, establishing the connectivity between PR regions, and adding new modules in the library should be automatically handled by the framework without needing the user's intervention. An elaborate runtime could be designed to automatically handle these allocations and to manage fabric resource, external memory bandwidth, and time (e.g., scheduling and mapping of stages to PR regions). After the plug-and-play architecture is implemented and PR bitstreams are generated for a set of modules, a problem occurs if the user wants to add a module in the library that is too large to fit into any of the PR regions already floorplanned. If it is the case, a new fabric layout needs to be determined with different PR region number and size to accommodate for the new module. PR bitstreams for the entire module library need to be regenerated for this new layout, that is, PR region-specific bitstreams for all modules in the library need to be recompiled for this new layout, which may result in very long compilation time if the module library is large. The amorphous technique (Appendix A) can potentially help with this problem since (1) the size of PR regions can be changed at runtime and (2) only PR region-specific bitstreams for the new module need to be generated. This technique could be relevant for systems that need frequent upgrades, for instance, in use-cases where the functionalities to accelerate on the FPGA keep changing or add up over time.

Efficient Mapping of Neural Networks in the Framework. The framework could also be extended to support the execution of neural network accelerators mapped as streaming pipelines on the FPGA. Some layers in large neural networks have high external memory bandwidth requirements and are typically I/O bound on FPGA platforms with less than 10 GB/s of external memory bandwidth. For neural networks, a strategy where multiple stages are reprogrammed and executed in a pipelined fashion might be better suited than serializing execution on one PR region (this strategy mostly benefits compute-bound stages).

Increasing PR Speed with Multiple PR Controllers. This work discusses the costs and benefits when increasing PR speed through one PR controller. Another way to increase overall PR bandwidth is by having multiple PR controllers. While not supported on current FPGAs, multiple PR controllers, each with a lower PR speed, allow multiple reconfigurations to happen in parallel, and might be more power/energy efficient than having a single very fast PR controller.

Appendix A

The Amorphous Technique

This appendix presents the amorphous technique, a non-conventional method to overcome the fixed boundary constraint imposed by current FPGA tools. Section A.1 first discusses the motivations behind using this technique vs standard PR. Section A.2 presents the amorphous DPR technique. Section A.3 introduces our evaluation setup. Lastly, Section A.4 presents our results.

A.1 Overview

Use-Cases. In the framework presented in Chapter 5, the FPGA fabric is divided into multiple reconfigurable partitions with fixed boundaries at build time. Each PR partition is provided with a standard interface connection (e.g., AXI4 or AXI4 streaming). The PR partitions are enclosed by an static I/O infrastructure that provides datapath to connect the reconfigurable partitions, through the standard interface, with each other and with off-fabric resources (e.g., on-chip embedded processor, off-chip DRAM and I/O). At runtime, the reconfigurable partitions can be dynamically reconfigured for use by independent or loosely-coupled modules to flexibly share the fabric spatially and temporally. Example systems of this kind of dynamically managed fabric use-case include [7,30,58].

Problem: Fragmentation. Dividing a fabric into reconfigurable partitions with fixed boundaries causes the available fabric resources to become fragmented. We risk creating *external fragmentation* if we divide the fabric into many small reconfigurable partitions. An over-sized module cannot be loaded onto the fabric unless a sufficiently large reconfigurable partition had been allocated when the reconfigurable partitions were floorplanned. We risk creating *internal fragmentation* if we try to make the reconfigurable partitions large

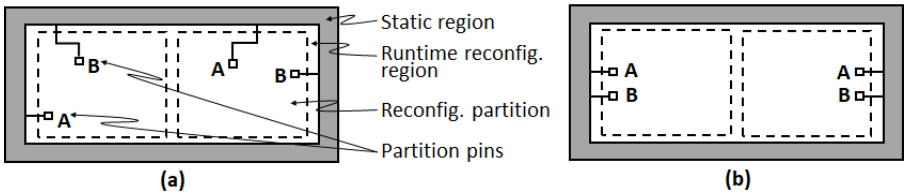

Figure A.1: An FPGA fabric organized into a top-level static region enclosing a runtime reconfigurable region subdivided as two reconfigurable partitions. The partition pins in (a) have been arbitrarily placed; the partition pins in (b) have been placed deliberately.

enough. This reduces the number of modules that can run concurrently; the large reconfigurable partitions would frequently be wasted on under-sized modules. In our vision processing use-case (Chapter 5), the effect is especially pernicious for SRAM and DSP resources that are in very high demand.

Solution: "Amorphous" PR. We devise a technique that does away with the need to commit upfront to a layout of fixed reconfigurable partition boundaries. This technique relies on the assumption that reconfigurable partitions only physically connect with the static I/O infrastructure and never directly with each other. Only the boundary of the static region and the locations of the AXI4 interface nets have to be fixed at build time. Instead of mapping a module to fit in a reconfigurable partition's fixed boundary, we map a module to a custom floorplan that only encloses the minimum consumed fabric region around an interface. In fact, for each module, we compile multiple bitstream versions corresponding to differently shaped footprints; each footprint option is chosen to minimize the potential for conflict with other modules' footprints. At runtime, a desired combination of modules can occupy the fabric at the same time if a non-overlapping packing of footprints can be found from the available versions.

Contributions. We verify the feasibility of the amorphous technique on Xilinx Zynq SoC FPGAs using Vivado. We further integrate amorphous PR into our vision processing pipeline framework. Doing away with the impositions of fixed reconfigurable partition boundaries removes resource fragmentation and thus, greatly expands the allowed module combinations that can co-exist on the fabric simultaneously.

We evaluate the improvement in placement rate (fraction of a given set of module combinations that can be placed successfully) when using the amorphous technique vs. standard PR in our vision framework. We also evaluate the savings in PR time because amorphous PR reconfigures only the footprint area actually used for a module (instead of a complete reconfigurable partition regardless of the degree of utilization within when using standard PR). The results show that amorphous PR offers significant improvement in flexibility

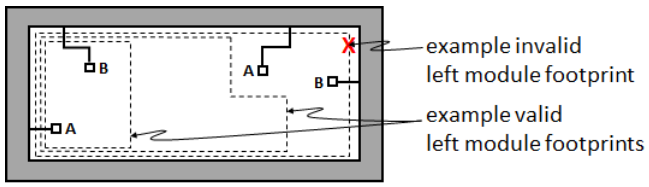

Figure A.2: The elements actually locked down as the result of building the floorplan in Figure A.1(a). The dashed outlines indicate examples of valid and invalid footprint options for building modules to attach to the left interface.

and efficiency over standard PR in our vision processing pipeline framework.

A.2 Amorphous PR

For the dynamically managed fabric use-case (e.g., framework in Chapter 5, allocating too-large reconfigurable partitions creates internal fragmentation; allocating too-small reconfigurable partitions creates external fragmentation. Either way, the effect is that some un-utilized resources become off-limits—due to some boundary line—to a module that needs them. This inefficiency and inflexibility is a significant obstacle to the dynamically managed fabric use-case.

Flexible Boundaries. We realize we could avoid fragmentation by doing away completely with the requirement of fixing the reconfigurable partition boundaries at build time. This is because in our use-case, the modules to be configured at runtime only connect physically with the static region and never directly with each other. At build time, we only have to fix (1) the boundary of the static region and (2) the resources reserved for the AXI4 interface nets and the partition pins. Figure A.2 depicts the elements actually locked down as the result of building the floorplan in Figure A.1(a).

Instead of confining a module to a predetermined reconfigurable partition boundary, we could build a module to attach to the left interface using any of the several possible valid footprints (examples shown in dashed lines in Figure A.2). For a given module, the footprint only needs to be large enough to contain the required fabric resources. The same flexibility is available when building modules for the right interface. Please note that all resources (including routing) needed by a given module must be entirely contained within its footprint.

At runtime, two PR bitstreams—one for the left and one for the right interface—can be simultaneously loaded provided their footprints do not overlap. Figure A.3 shows the example where the large footprint

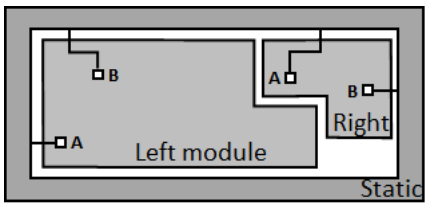

Figure A.3: A valid packing of two non-overlapping footprints of modules for the left and right interfaces. The left module would have been too large to fit within the fixed boundaries of the left reconfigurable partition in Figure A.1(a).

of a resource-demanding module, attached to the left interface, co-exists with the small footprint of a less resource-demanding module, attached to the right interface. Some resources are left over, not needed by either footprint. This combination of modules would have been prevented by resource fragmentation had we followed the fixed, equally resourced reconfigurable partitions in Figure A.1(a) or Figure A.1(b).

Interface Placement. Using amorphous PR, we no longer have to make hard decisions on how to divide up the runtime reconfigurable region upfront. The decision is reduced to how many AXI4 interfaces to support and the placement of the AXI4 interfaces' partition pins. The placement of partition pins should not be arbitrary as they can interfere with the packing of module footprints. For example, the largest footprint in Figure A.2 is not valid for attaching a module to the left interface because it also encloses the partition pins for the right interface. Thus, we can see that the deliberate placement of partition pins in Figure A.1(b) is preferable to Figure A.1(a) because the deliberate placement is less restrictive. *Please note that the resources withheld for the interface nets do not pose similar restrictions.*

When extrapolating to a realistic implementation supporting many more interfaces, the placement of the partition pins becomes of strategic importance. The goal is to allow one module's footprint—which must include its own interface's partition pins—to grow, as necessary, unimpeded by other interfaces' partition pins. For the sizes of contemporary available FPGAs, one heuristic is to place the interfaces evenly along the periphery of the runtime reconfigurable region. This heuristic allows interfaces to access more freely the resources in the runtime reconfigurable region, by allowing the module footprints to grow toward the interior of the region.

To place the large number of signals associated with the AXI4 and AXI4-Lite interfaces, we use the floorplanner to tightly constrain the outline of a placeholder reconfigurable partition so the interface signals will be automatically placed into an intended area. Later, when building a module to attach to a particular

interface, we use the floorplanner to expand the associated placeholder reconfigurable partition's original boundary to the desired rectilinear footprint. This final footprint outline, as well as any of the partition pins and nets reserved within, is then used to constrain the place-and-route to produce a footprint- and interface-specific version of the PR bitstream.

Footprint/Bitstream Management. Under standard DPR, a module needs to have different bitstream versions to be instantiated in different reconfigurable partitions. Under amorphous PR, one module can have still more versions of PR bitstreams, each corresponding to a particular interface attachment and a particular footprint. This extra degree of freedom in footprint choice expands the set of valid combination of modules that can be loaded on the fabric simultaneously. The downsides to this degree of freedom are (1) increased storage for additional bitstream versions and (2) algorithmic complexity in optimizing the compile-time decisions of footprint choices, and the runtime decisions of bitstream version selection.

A.3 Evaluation Methodology

In this section, we explain the metrics and methodology used to evaluate the effectiveness of amorphous PR over standard PR in our vision framework. We use synthetic benchmarks to focus the evaluation on the fragmentation of BRAM and DSP blocks, which have been the resource bottleneck in our usage. We consider 3 synthetic module workloads ($\text{Workload}_{\text{BRAM}}$, $\text{Workload}_{\text{DSP}}$ and $\text{Workload}_{\text{mixed}}$) that focus on BRAM-only, DSP-only, and mixed BRAM/DSP, respectively.

A.3.1 Metrics

Placement Rate. The primary metric is the placement rate. For this measurement, we assume there exists a library of modules where each module has a number of bitstream versions available corresponding to different interface attachments and, in the case of amorphous PR, also different footprint shapes. An user can demand a combination of up to $N_{\text{interfaces}}$ modules to be in-use at a time ($N_{\text{interfaces}}$ is the number of AXI4 interfaces available). Some combinations may not be feasible due to FPGA resource bounds. In standard PR, a combination is not feasible when some of the demanded modules cannot fit into the fixed reconfigurable partitions available. In amorphous PR, a demanded combination is not feasible due to footprint conflicts, that is, a valid non-overlapping packing of the available footprints cannot be found. *Placement rate is the fraction of feasible combinations for a given set of demanded combinations.*

PR Time Overhead. During PR, the affected fabric region is not contributing to computation for a time, resulting in a loss of performance. Amorphous PR can be faster than standard PR because amorphous PR reconfigures only a required footprint size. Standard PR reconfigures the entire PR partition regardless of the actual resource utilization within.

To quantify the difference in reconfiguration overhead, we consider an interactive scenario where the user demands a sequence of module combinations. Consecutive module combinations in the sequence differ by $N_{\text{module-delta}}$ modules, where $N_{\text{module-delta}}$ is an experimental parameter that specifies how many modules change between consecutive combinations in a sequence. We measure PR time overhead as the total time lost to PR over the demanded sequence. The reconfiguration process is handled through the processor configuration access port (PCAP), with an empirically observed bandwidth of 128 MB/s.

Keep in mind, this is a direct measurement of overhead. In practice, the overhead's significance must be weighed against the execution interval between PR events. Also, in measuring overheads, we assume execution interval is synchronized such that modules are only changed together in between intervals. In general, the lifetime of different modules needs not be coupled.

A.3.2 Evaluation Platform

FPGA and tool. We use the Xilinx ZC702 development board with an XC7Z020 SoC FPGA for our evaluation. The XC7Z020 SoC FPGA has 53,200 logic cells, 140 BRAMs and 220 DSP blocks. We use Xilinx Vivado version 2014.4 for all the builds. All designs are placed-and-routed at 100 MHz.

static region. We built three instances of our parameterized vision framework (Chapter 5) to support the three workloads. All three static region instances support six modules ($N_{\text{interfaces}} = 6$), but the AXI4 interfaces provided are specialized to the workload. $\text{Static}_{\text{BRAM}}$ provides AXI4 interfaces to DMA; $\text{Static}_{\text{DSP}}$ provides AXI4-Stream interfaces; and $\text{Static}_{\text{mixed}}$ provides both. When building the static region, we manually positioned the AXI4 interfaces' partition pins.

On the small XC7Z020 SoC FPGA, the static region can consume as much as 45% of the available logic cells and 25% of the available BRAMs. Although the static region does not make use of DSP blocks, it can still prevent some of the DSP blocks from being used by modules loaded into the runtime reconfigurable region region. Table A.1 summarizes the resources available in the runtime reconfigurable region regions for the three workloads.

The real deployment of our vision framework is on a custom embedded board with an XC7Z045 SoC

Table A.1: Resources in runtime reconfigurable region region by workload

Workload	Logic Cell	BRAM	DSP	AXI4
BRAM36Kb	27816	80	90	memory
DSP	23968	38	120	streaming
mixed	22712	40	80	memory+streaming

FPGA. There, the static region supports up to 12 AXI4 interfaces for modules, consuming about 5% of the available logic cells and 1% of the available BRAMs. The runtime reconfigurable region region has over 200,000 logic cells, 500 BRAM blocks and 900 DSP blocks to be flexibly shared by the 12 modules. In our experience, we can reliably use up to around 70% of the available resources in the runtime reconfigurable region region before the tool experiences difficulty in routing and timing-closure.

A sample screenshot of the static region floorplan on the XC7Z045 SoC FPGA is shown in Figure A.4. The screenshot gives an indication of the relative sizes of the static region and the runtime reconfigurable region. Within the runtime reconfigurable region region, the areas enclosing the individual AXI4 interfaces are highlighted as well.

PR Regions and Amorphous Footprints. When evaluating standard PR, the naive baseline case divides the runtime reconfigurable region into six roughly equally resourced PR regions. In addition, for each experiment conducted, we tested 1000 randomized layouts of six PR regions that enclose different fractions of the total resources (nonsensical layouts are pruned from consideration). For each experiment, the best result from among the 1000 layouts is reported as best-effort. This is to approximate the results of a tuned layout when the workload mix is known ahead of time.

In order to conduct the best-effort study, a large number of bitstream versions has to be generated for each module, corresponding to different interface attachments and differently shaped PR regions. We directly adopted this collection of bitstreams as the bitstream database for amorphous PR. As such, in our evaluations, amorphous PR can always match the results of best-effort standard PR by using the corresponding selection of module bitstreams. Amorphous PR can exceed the best-effort standard PR because it can also combine bitstreams arising from different layouts, whereas standard PR is limited to one fixed layout at a time.

In a real scenario, instead of generating a large number of random footprint bitstream versions, one would strategically maintain a much smaller number of well-chosen footprints following heuristics such as to pack

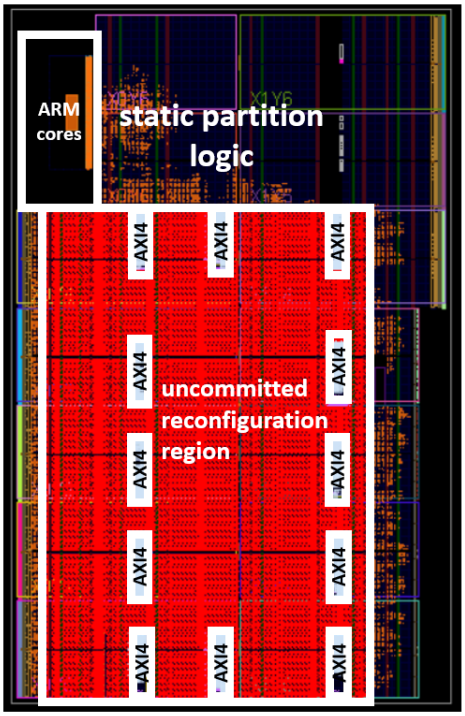

Figure A.4: A sample screenshot of the static region floorplan on the XC7Z045 SoC FPGA. This instance of the vision framework supports 12 AXI4 interfaces.

tightly around the reserved interface region and to obey handedness when consuming a fraction of a column (i.e., consume from the bottom if reaching from the right and vice versa).

A.3.3 Synthetic Workloads

Below we describe the three synthetic module workloads ($\text{Workload}_{\text{BRAM}}$, $\text{Workload}_{\text{DSP}}$ and $\text{Workload}_{\text{mixed}}$) that focus on BRAM-only, DSP-only, and mixed BRAM/DSP, respectively. Each workload has three variants of different degrees of difficulty.

$\text{Workload}_{\text{BRAM}}$. We use Vivado HLS to develop a simple module design to read a large number of values from DRAM into BRAM and to compute the sum of those values. The module design is parameterizable to use different numbers of BRAMs. We constructed a library comprising different module instances utilizing between 0 and 40 BRAMs in increments of 5 BRAMs. (The runtime reconfigurable region region in $\text{Static}_{\text{BRAM}}$ has 80 available BRAMs total. Modules with more than 40 BRAMs almost always result in failed synthesis even for the largest PR region/footprint considered.) From this library, we randomly select modules to form the demanded module combinations to measure placement rate and overhead. Selecting a

0-BRAM module corresponds to a combination where less than six modules are demanded. As in our real usage experience, the modules use relatively little logic cell resources so their fragmentation and conflicts are not considered in this evaluation; this applies to all three workloads studied.

The advantage of amorphous PR over standard PR depends on resource utilization pressure. Therefore, for each workload, we consider three variants with different degrees of difficulty, *Easy*, *Hard*, *Harder*. For *Easy*, we restricted module selection to come from modules utilizing 0 up to 20 BRAMs. The selected modules on average utilize $10 = (0 + 5 + 10 + 15 + 20)/5$ BRAMs, less than the average number of BRAMs, $13.3 = 80/6$, available to each interface. For *Hard* and *Harder*, we raise the BRAM ceiling to 30 and 40, respectively.

Workload$_{\mathbf{DSP}}$. We use the FFT IP with AXI4-Stream interface from Vivado's IP Library. The FFT IP is parameterizable to use different numbers of DSP blocks. Similar to Workload$_{\mathrm{BRAM}}$, we construct a library comprising different module instances utilizing between 0 and 50 DSP blocks in increments of 5 DSP blocks. For *Easy*, *Hard* and *Harder*, we restrict module selections to come from modules utilizing a maximum of 30, 40, and 50 DSP blocks, respectively. The runtime reconfigurable region in Static$_{\mathrm{DSP}}$ has 120 available DSP blocks total.

Workload$_{\mathbf{mixed}}$. This last workload mixes modules from the two previous workloads. For *Easy*, *Hard* and *Harder*, we restrict module selection to come from modules utilizing either a maximum of 20, 30 or 40 BRAMs; or a maximum of 20, 30, or 40 DSP blocks. The runtime reconfigurable region region in Static$_{\mathrm{mixed}}$ has 40 available BRAMs and 80 available DSP blocks total.

A.4 Results

This section presents the outcomes of the evaluations outlined in the last section.

A.4.1 Placement Rates

Following the procedures described in the last section, for each placement rate measurement, we generated 1000 module combinations, each with up to six modules randomly selected according to workload and degree of difficulty.

Figure A.5 reports the placement rates (y-axis) for naive standard PR vs. best-effort standard PR vs. amorphous PR in experiments corresponding to different workloads (separated by plots) and degrees of

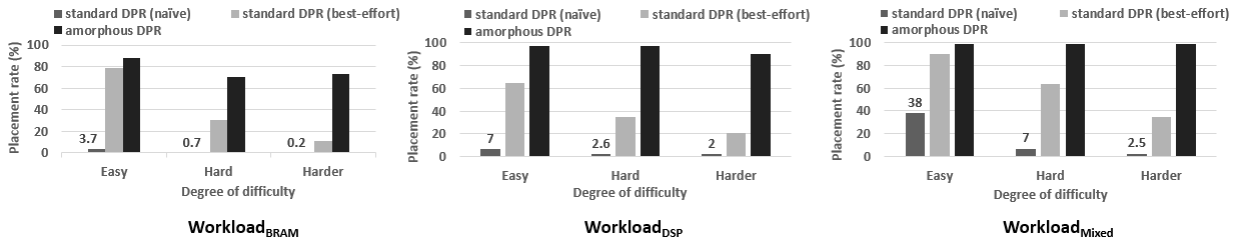

Figure A.5: Comparing the placement rates achieved by naive standard PR vs. best-effort standard PR vs. amorphous PR.

difficulty (x-axis). The placement rate for naive standard PR is poor even for the *Easy* variant of the workloads. The *Easy* workload variants are setup such that the module average resource requirement is just less than the resource available in a naive standard PR region. However, a combination fails if any of the six modules is above average. By tuning the PR region sizes according to workload at build time, best-effort standard PR does well on the *Easy* variant of the workloads (up to 80% placement rate) but is unable to cope with the utilization pressure as the degree of difficulty increases to *Hard* and *Harder*.

Amorphous PR achieves over 80% placement rate on all workload variants, except for the *Hard* and *Harder* variants of $\text{Workload}_{\text{BRAM}}$ at over 70%. More telling than the absolute values are the improvements from standard PR to amorphous PR. As expected, we observe that amorphous PR yields greater improvement going from *Easy* to *Hard* to *Harder* workloads. The significant differences on the *Harder* variants translate tangibly to a much greater effective usable capacity in a dynamically managed fabric use-case like our vision framework.

A.4.2 Reconfiguration Overhead

To evaluate reconfiguration overhead, we randomly constructed 1000-combination long sequences. The sequences include only combinations that are valid in both best-effort standard PR and amorphous PR. Figure A.6 reports for $\text{Workload}_{\text{BRAM}}$ the average reconfiguration time (y-axis), in milliseconds, spent in transitioning between consecutive combinations. We report results when using best-effort standard PR vs. amorphous PR in experiments corresponding to different degrees of difficulty (x-axis). We do not report results for naive standard PR because it accepts too few combinations to be included for comparison. The separate plots in Figure A.6 correspond to results using sequences with $N_{\text{module-delta}}$=1, 2, 3, and 4, respectively. (Recall, $N_{\text{module-delta}}$ is a parameter that specifies how many modules change between consecutive combinations in a

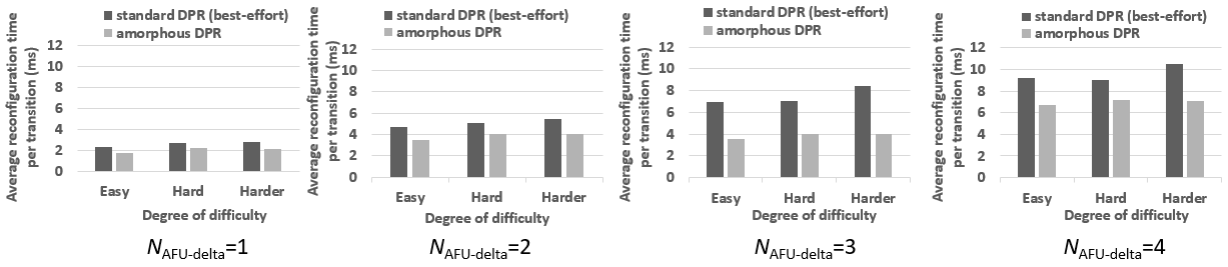

Figure A.6: Comparing the average reconfiguration times per transition for $\text{Workload}_{\text{BRAM}}$ when using best-effort standard PR vs. amorphous PR.

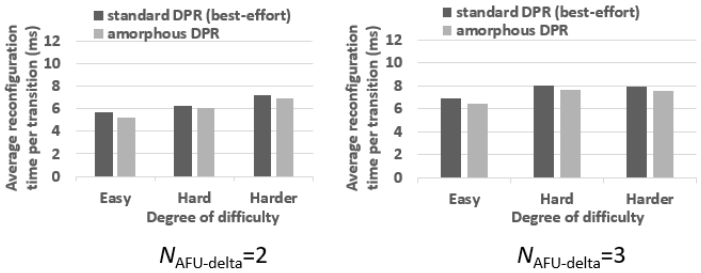

Figure A.7: Comparing the average reconfiguration times per transition for $\text{Workload}_{\text{DSP}}$ when using best-effort standard PR vs. amorphous PR.

sequence.) Figures A.7 and A.8 report the results for $\text{Workload}_{\text{DSP}}$ and $\text{Workload}_{\text{mixed}}$, respectively. Plots for some values of $N_{\text{module-delta}}$ are missing because not enough combinations are acceptable under standard PR to make meaningful comparisons.

The average reconfiguration time spent in transitioning between consecutive combinations correlates most strongly with the bitstream size of loaded modules. We observe that the average reconfiguration time increases directly with the number of modules changed, $N_{\text{module-delta}}$. The average reconfiguration time is also sensitive to the degrees of difficulty, which affects the range of module sizes involved. The ratios of average reconfiguration time of best-effort standard PR over amorphous PR are between 1.1× and 1.5×. This ratio corresponds well with the ratios of their respective bitstream sizes. Though not reported, naive standard PR would do much worse than both best-effort standard PR and amorphous PR because its six equally resourced PR regions would quite often be larger than necessary for the modules, due to variations in module resource requirements.

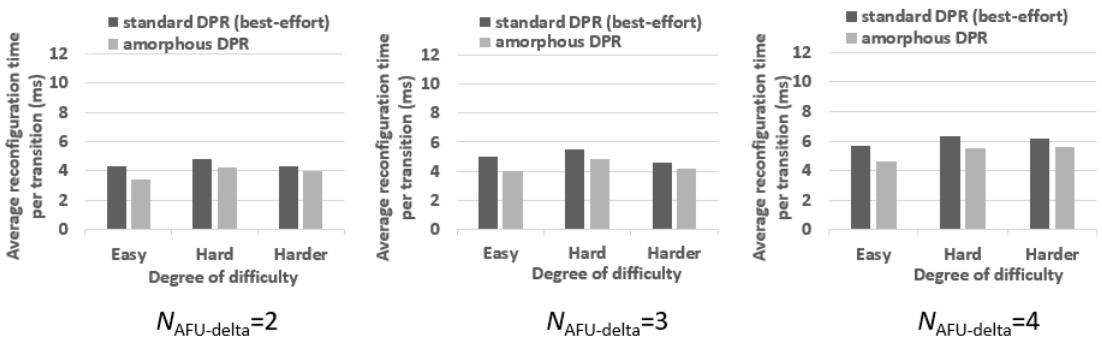

Figure A.8: Comparing the average reconfiguration times per transition for $\text{Workload}_{\text{mixed}}$ when using best-effort standard PR vs. amorphous PR.

Bibliography

[1] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid methods in image processing. 1984.

[2] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl. Reconos: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, Jan 2014.

[3] J. Arram, W. Luk, and P. Jiang. Ramethy: Reconfigurable Acceleration of Bisulfite Sequence Alignment. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 250–259, New York, NY, USA, 2015. ACM.

[4] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 335–340, June 2005.

[5] T. Becker, W. Luk, and P. Y. K. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 35–44, April 2007.

[6] S. Bhandari, S. Subbaraman, S. Pujari, F. Cancare, F. Bruschi, M. D. Santambrogio, and P. R. Grassi. High speed dynamic partial reconfiguration for real time multimedia signal processing. In *2012 15th Euromicro Conference on Digital System Design*, pages 319–326, Sep. 2012.

[7] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, May 2014.

[8] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). In *Proceedings of the The Roadmap to Reconfigurable Computing,*

10th International Workshop on Field-Programmable Logic and Applications, FPL '00, page 605–614, Berlin, Heidelberg, 2000. Springer-Verlag.

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

[10] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 14:1–14:14, New York, NY, USA, 2017. ACM.

[11] F. Chollet et al. Keras, 2015.

[12] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, Mar 2018.

[13] C. Claus, W. Stechele, and A. Herkersdorf. Autovision – a run-time reconfigurable mpsoc architecture for future driver assistance systems (autovision – eine zur laufzeit rekonfigurierbare mpsoc architektur für zukünftige fahrerassistenzsysteme). 49:181–, 05 2007.

[14] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich. A comparison of embedded reconfigurable video-processing architectures. In *2008 International Conference on Field Programmable Logic and Applications*, pages 587–590, Sept 2008.

[15] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, June 2002.

[16] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005.

[17] P. K. Daniele Bagni and S. Neuendorffer. Demystifying the lucas-kanade optical flow algorithm with vivado hls. 2017.

[18] C. Dennl, D. Ziener, and J. Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 45–52, April 2012.

[19] G. Dessouky, M. J. Klaiber, D. G. Bailey, and S. Simon. Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[20] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings - Computers and Digital Techniques*, 147(3):181–188, May 2000.

[21] M. Dyer, C. Plessl, and M. Platzner. Partially reconfigurable cores for xilinx virtex. In M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 292–301, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[22] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Performance bounds of partial run-time reconfiguration in high-performance reconfigurable computing. In *Proceedings of the 1st International Workshop on High-Performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07*, HPRCTA '07, page 11–20, New York, NY, USA, 2007. Association for Computing Machinery.

[23] S. A. Fahmy, K. Vipin, and S. Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435, Nov 2015.

[24] B. A. Farisi, K. Heyse, and D. Stroobandt. Reducing the overhead of dynamic partial reconfiguration for multi-mode circuits. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 282–283, Dec 2014.

[25] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. C. van der Veen, J. Angermeier, D. Koch, and J. Teich. Dynamic Defragmentation of Reconfigurable Devices. *ACM Trans. Reconfigurable Technol. Syst.*, 5(2):8:1–8:20, June 2012.

[26] K. Fleming, H. Yang, M. Adler, and J. Emer. The leap fpga operating system. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2014.

[27] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[28] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 84–93, New York, NY, USA, 2019. Association for Computing Machinery.

[29] J. Goeders, T. Gaskin, and B. Hutchings. Demand driven assembly of fpga configurations using partial reconfiguration, ubuntu linux, and pynq. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 149–156, April 2018.

[30] D. Goehringer, L. Meder, M. Hubner, and J. Becker. Adaptive Multi-client Network-on-Chip Memory. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 7–12, Nov 2011.

[31] L. Gong and O. Diessel. Resim: A reusable library for rtl simulation of dynamic partial reconfiguration. In *2011 International Conference on Field-Programmable Technology*, pages 1–8, Dec 2011.

[32] N. B. Grigore and D. Koch. Placing partially reconfigurable stream processing applications on FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[33] D. Gschwend. Zynqnet: an fpga-accelerated embedded convolutional neural network. 2016.

[34] S. M. K. Gueye, G. Delaval, É. Rutten, and J. Diguet. Discrete and logico-numerical control for dynamic partial reconfigurable fpga-based embedded systems: A case study. In *IEEE Conference on*

Control Technology and Applications, CCTA 2018, Copenhagen, Denmark, August 21-24, 2018, pages 1480–1487, 2018.

[35] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll. Fpga-based real-time pedestrian detection on high-resolution images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2013.

[36] M. Harvey. Five video classification methods, 2017.

[37] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[38] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4):85:1–85:11, July 2016.

[39] C. H. Hoo and A. Kumar. An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 400–406, Aug 2012.

[40] P.-A. Hsiung, C.-H. Huang, and Y.-H. Chen. Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC. *J. Embedded Comput.*, 3(1):53–62, Jan. 2009.

[41] C. Huriaux, O. Sentieys, and R. Tessier. Fpga architecture support for heterogeneous, relocatable partial bitstreams. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sep. 2014.

[42] Intel. Intel fpga sdk for opencl intel arria 10 gx fpga development kit reference platform porting guide. 2019.

[43] Intel. *Intel Stratix 10 Configuration User Guide*, 2019.

[44] Itseez. *The OpenCV Reference Manual*, 2.4.9.0 edition, April 2014.

[45] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S. K. Park, M. Kim, and J. W. Jeon. Fpga design and implementation of a real-time stereo vision system. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(1):15–26, Jan 2010.

[46] N. Katsaros and N. Patsiatzis. A real time histogram of oriented gradients implementation on fpga. 2017.

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

[48] N. S. Kim and P. Mehra. Practical near-data processing to evolve memory and storage devices into mainstream heterogeneous computing systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 22:1–22:4, New York, NY, USA, 2019. ACM.

[49] N. S. Kim and P. Mehra. Practical near-data processing to evolve memory and storage devices into mainstream heterogeneous computing systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[50] D. Koch and C. Beckhoff. Hierarchical reconfiguration of FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[51] D. Koch and J. Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on Fpgas for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[52] M. Koester, H. Kalte, M. Porrmann, and U. Rückert. *Defragmentation Algorithms for Partially Reconfigurable Hardware*, pages 41–53. Springer US, Boston, MA, 2007.

[53] Y. E. Krasteva, E. D. La Torre, T. Riesgo, and D. Joly. Virtex ii fpga bitstream manipulation: Application to reconfiguration control systems. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, Aug 2006.

[54] S. Li and W. Deng. Deep facial expression recognition: A survey, 2018.

[55] X. Li, X. Wang, F. Liu, and H. Xu. Dhl: Enabling flexible software network functions with fpga acceleration. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1–11, July 2018.

[56] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, Sept 1999.

[57] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[58] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. *J. VLSI Signal Process. Syst.*, 47(1):15–31, Apr. 2007.

[59] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev. Online hardware task scheduling and placement algorithm on partially reconfigurable devices. In *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ARC '08, pages 306–311, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Microsoft. Live video analytics, 2012.

[61] M. Nguyen and J. C. Hoe. Amorphous dynamic partial reconfiguration with flexible boundaries to remove fragmentation. *CoRR*, abs/1710.08270, 2017.

[62] M. Nguyen, R. Tamburo, S. Narasimhan, and J. C. Hoe. Quantifying the benefits of dynamic partial reconfiguration for embedded vision applications. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 129–135, Sep. 2019.

[63] T. D. A. Nguyen and A. Kumar. Maximizing the serviceability of partially reconfigurable fpga systems in multi-tenant environment. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 29–39, New York, NY, USA, 2020. Association for Computing Machinery.

[64] J. Noguera and I. O. Kennedy. Power reduction in network equipment through adaptive partial reconfiguration. In *2007 International Conference on Field Programmable Logic and Applications*, pages 240–245, Aug 2007.

[65] F. Oleari, F. Kallasi, D. Lodi Rizzini, J. Aleotti, and S. Caselli. Performance evaluation of a low-cost stereo vision system for underwater object detection. volume 19, 08 2014.

[66] H. Omidian and G. G. Lemieux. Software-based dynamic overlays require fast, fine-grained partial reconfiguration. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators*

and Reconfigurable Technologies, HEART 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[67] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, Aug 2006.

[68] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[69] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott. Finn-l: Library extensions and design trade-off analysis for variable precision lstm networks on fpgas. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 89–897, 2018.

[70] B. Samynathan, K. Chapman, M. Nik, B. Robatmili, S. Mirkhani, and M. Lavasani. Computational storage for big data analytics. *10th International Workshop on Accelerating Analytics and Data Management Systems*, 2019.

[71] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, pages 131–140, Dec 2001.

[72] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompson, and A. D. Pimentel. Runtime Task Mapping Based on Hardware Configuration Reuse. In *2010 International Conference on Reconfigurable Computing and FPGAs*, pages 25–30, Dec 2010.

[73] H. K.-H. So and R. Brodersen. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. *ACM Trans. Embed. Comput. Syst.*, 7(2):14:1–14:28, Jan. 2008.

[74] N. K. Srivastava, S. Dai, R. Manohar, and Z. Zhang. Accelerating face detection on programmable soc using c-based synthesis. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 195–200, New York, NY, USA, 2017. ACM.

[75] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov 2004.

[76] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu. Dynamically reconfigurable systolic array accelerators: A case study with extended kalman filter and discrete wavelet transform algorithms. *IET Computers Digital Techniques*, 4(2):126–142, March 2010.

[77] A. Sudarsanam, R. Kallam, and A. Dasu. Prr-prr dynamic relocation. *Computer Architecture Letters*, 8:44 – 47, 03 2009.

[78] R. Tamburo, E. Nurvitadhi, A. Chugh, M. Chen, A. Rowe, T. Kanade, and S. G. Narasimhan. Programmable automotive headlights. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 750–765, Cham, 2014. Springer International Publishing.

[79] T. Taniai, S. Sinha, and Y. Sato. Fast multi-frame stereo scene flow with motion segmentation. 07 2017.

[80] N. Thomas, A. Felder, and C. Bobda. Adaptive controller using runtime partial hardware reconfiguration for unmanned aerial vehicles (uavs). In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7, Dec 2015.

[81] M. Ullmann, M. Huebner, B. Grimm, and J. Becker. An fpga run-time system for dynamical on-demand reconfiguration. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 135–, April 2004.

[82] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource elastic virtualization for fpgas using opencl. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117, Aug 2018.

[83] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[84] P. Viola and M. J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 57(2):137–154, May 2004.

[85] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 290–295, 2003.

[86] S. Wang, C. Zhang, Y. Shu, and Y. Liu. Live video analytics with fpga-based smart cameras. In *Workshop on Hot Topics in Video Analytics and Intelligent Edges (HotEdgeVideo)*, October 2019.

[87] Xilinx. *Xilinx Power Estimation*, 2014.

[88] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*, 2015.

[89] Xilinx. *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)*, 2015.

[90] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*, 2017.

[91] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual*, 2017.

[92] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*, 2019.

[93] Xilinx. *Vivado Design Suite User Guide: Partial Reconfiguration (UG909)*, 2019.

[94] Xilinx. *Vivado Design Suite User Guide: Using the Vivado IDE (UG893)*, 2019.

[95] Xilinx. *Zynq DPU v3.1*, 2019.

[96] J. Zhang, S. Khoram, and J. Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, page 4924–4932, 2018.

[97] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1270–1278, April 2019.

[98] 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. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.