Cloud RTR: Cloud Infrastructure for Apps with Hardware

by

A.Y. Ismail

B.S., Penn State University, 2013

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Masters of Science

Department of Electrical, Computer, and Energy Engineering

2015

This thesis entitled:
Cloud RTR: Cloud Infrastructure for Apps with Hardware
written by A.Y. Ismail
has been approved for the Department of Electrical, Computer, and Energy Engineering

Eric Keller

Prof. Dirk Grunwald

Prof. Pavol Cerny

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Ismail, A.Y. (M.S., Computer Engineering)

Cloud RTR: Cloud Infrastructure for Apps with Hardware

Thesis directed by Prof. Eric Keller

There has been a great deal of innovation in the software space for smart phones, however, there has been virtually no room to innovate in the architecture space. By introducing a Field Programmable Gate Array (FPGA) on mobile phone platforms, developers are able to produce hardware that their applications can use. We call re-programmable hardware on mobile phones mōbware. In order to introduce mōbware to mobile platforms, we leverage technology that contains a processor (ARM) coupled with FPGA so we can introduce reconfigurable logic to smart phones, expose the hardware to applications, and extend a mobile operating system to allow for software control of the current hardware configuration. There are obstacles with deploying mōbware so any phone can simply download it and deploy it on their FPGA. This dynamic loading of mōbware is called run-time reconfiguration (RTR). Achieving RTR of hardware requires tool support and a deployment infrastructure to efficiently implement it. We present a cloud RTR deployment model that allows for the production and distribution of mōbware. The deployment models converges the phone manufacturer, the application and mōbware developer, and user. With these parties converged, this allows for the sustainable production and deployment of mōbware.

Dedication

I dedicate this work to mom and dad, the people who supported me from day one.

Acknowledgements

I'd like to thank my advisor Eric Keller. He provided many resources and insights that lead to the success of this project. I thank Michael Coughlin, who worked primarily on the end-system section of this project. He provided a lot of support, especially the times when there never seemed to be a solution. I thank Phil James-Roxby and Derrick Woods, from Xilinx Inc. They provided a lot of technical support and advice on how to build this system. Also, I would like to thank AppFigures for providing last years upload figures for the Google Play Store.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

While there's a great deal of innovation in the software space for smart phones, there's relatively limited ability to innovate in the architecture space. Only the large vendors and a very few semiconductor companies are able to introduce new advances, whereas anyone can write software. There are many different examples of new technologies that greatly benefit from hardware support, including new wireless spectrum protocols, hardware-accelerated encryption and transparent scanning of memory or network packets. Many of these technologies can be demonstrated in laboratory environments using powerful PC or Field Programmable Gate Array (FPGA)-based platforms, or even custom silicon, but face significant hurdles to general deployment.

We have explored an alternative architecture to today's processor-centric mobile devices that introduces reconfigurable logic to smart phones, exposes the hardware to applications, and extends a mobile operating system to allow for software control of the current hardware configuration. A processor coupled with programmable logic (PL), such as an FPGA, along with operating system and development support, would open possibilities for developers to introduce new hardware as easily as they are able to introduce new software.

With the ability to expose re-programmable hardware to mobile phone applications, developers gain new hardware functionality. Technologies such as software-define radio (SDR), hardware-accelerated encryption (AES and RSA), and digital signal processing (FFT and FIR) can be im-

plemented on the FPGA. The applications can utilize these hardware technologies to give them even greater functionality. In addition, FPGA's have proven to increase performance and reduce power consumption [1][2][3][4]. Mobile applications can take advantage of the benefits of FPGAs to increase their performance and reduce their power footprint. With this realization, it is advantageous to produce a deployment model that gives developers the ability to produce hardware that applications can use and a way to deploy their mōbware (hardware utilized by mobile applications).

The greatest obstacle to introducing an FPGA to mobile phones is supporting the diversity of hardware modules required by different applications. It is not trivial to insert and replace hardware modules onto the FPGA as the hardware requirements for applications change. For example, application A utilizes an FFT and application B utilizes an AES hardware module. Between these two hardware modules, assume they take up the entire FPGA area. Application A is closed down and application C starts and it requires a FIR filter. Since there is no more room on the FPGA, the operating system would remove the FFT and replace it with the FIR filter. This process of swapping newly introduced hardware modules in and out of the FPGA at run-time is known as run-time reconfiguration (RTR). In addition, there is the more difficult scenario of general RTR, where, it is not known at compile-time what hardware modules will be swapped in and out.

There are many technical limitations to general RTR and the tools provided by FPGA vendors that have not met the demand for it, forcing the research community[5] [6][7][8][9] [10][11][12][13][14] [15] to develop their own tools and hardware. Fast forwarding to today, the hardware and tools introduced by the research community have not been adopted. The tools provided by FPGA vendors are finally able to accommodate specific scenarios of RTR with minimal support. These tools can be leveraged to implement general RTR in mobile phones and develop tools to automate the process so we can accommodate a deployment that allows developers to upload hardware designs, phone manufacturers to upload their base hardware configuration, and users to download mōbware.

We present a cloud-based deployment infrastructure that relies on the Xilinx tools for RTR and automation. We also leverage the application deployment model, in which applications are distributed through a cloud provider. This allows us to implement general RTR using only mainstream vendor tools, since all compilation can be done within the cloud. This cloud-based deployment infrastructure creates an environment that allows phone manufacturers to define and configure the base hardware, developers to design and upload mōbware, and users to download this hardware without being burdened by compilation.

The contributions of this thesis is an infrastructure and process for leveraging available technology to accommodate mōbware. Section 2 describes the motivation for introducing FPGAs to mobile phones. Section 3 describes the deployment scenario we want to accommodate. Section 4 describes the academic approach for general RTR. Section 5 describes our approach for achieving general RTR to support the deployment scenario. Section 6 is an evaluation of the tools of our approach. Section 7 describes work to continue in the future and section 8 concludes the work.

Chapter 2

Motivation

There is much to gain by coupling CPU's with FPGA's. A computationally intensive workload can be offloaded to the FPGA and computed at lower power and greater performance. FPGA's run at much lower frequencies than CPUs, allowing for lower power consumption. It also provides the ability for hardware parallel computation, allowing for more work to be accomplished per unit time. The benefits of FPGA's have not gone unnoticed, thus, industry and researchers have spent a lot of effort investigating the advantages of utilizing FPGAs to offload work from CPUs to increase performance and reduce power consumption [1][2][3][16][17][18][31].

2.1 FPGA Power Reduction

Since the late 1990's and early 2000's, CPU's have hit a power wall. As clock frequencies started to climb, power consumption climbed with it. The solution to the power wall was to reduce clock frequency, but, at the same time, increase throughput[19]. The way to do this is through parallelization.

FPGA's are an excellent way to achieve parallelization that would allow for lower clock frequencies, while increasing throughput. It has been a standard misconception that large FPGA designs have a strong relationship with power consumption, however, this is not true. It is true that power increases with larger designs, but the effect of design size on power consumption is not as

great as the clock frequency. FPGA's have two sources of power consumption, static and dynamic power consumption. Static power consumption refers to power consumed as a result of current leaking from the FPGA's transistors. Dynamic power consumption refers to power consumed from current flow through transistors as they switch states [16]. The greater the frequency, the more power that is consumed. Designers have found that they can take advantage of these relationships to reduce power. The idea is simple, utilize more of the FPGA hardware to increase throughput at lower frequencies.

Possa et al. [1] explored the increased performance and reduced power consumption abilities of FPGA's when used as accelerators. They utilized a System-on-a-Programmable-Chip (SoPC), which contained a Nios II 32-bit RISC embedded processor and Altera's Cyclone III FPGA. They implemented a 15^{th} order FIR filter in both software and hardware. They compared the power consumption of the processor that utilized a software FIR filter to the combination of the processor and hardware FIR filter. They implemented five different hardware accelerators, each differing by how they interface with the CPU. Figure 2.1 shows the power reduction achieved by the accelerators. The different colored bars correspond to the mode of the processor, such as, fast (green), economic (blue), and standard (red). After measuring power for each case, all of their hardware implementations were able to achieve power reduction in the range of 83 % to 93 %.

It is also interesting to note that they achieved 83 % power reduction utilizing a hardware block they designed using the C programming language. Being able to describe hardware in a high-level programming language such as C and achieve 83% power reduction means developers can achieve FPGA power reduction without the burden designing hardware in HDL. This work is an early indicator that high-level synthesis tools that convert high-level languages to RTL are promising and can provide the power-saving benefits of FPGA's.

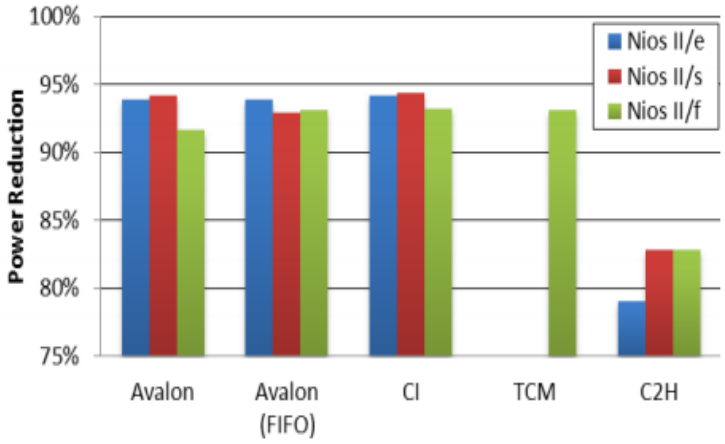

Figure 2.1: Power reduction achieved by the accelerators compared to software only execution [1].

2.2 FPGA Performance Improvement

In addition to power reduction, many studies have been done to quantify the performance increases of FPGA's when they are used as accelerators. As discussed earlier, Possa et al. measured the performance of the software and hardware implementations of the 15th order FIR filter. Figure 2.2 shows the speedup achieved by the hardware accelerators. All five hardware accelerators achieved speedup, however, there was a much wider range of speedup, from 5x to 117x. Nonetheless, speedup was achieved by offloading a CPU workload to an FPGA.

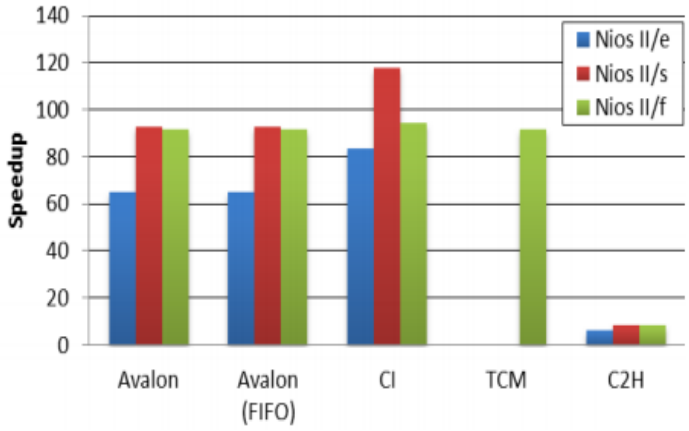

Figure 2.2: Speedup achieved by the accelerators compared to software only execution [1].

Cullinan et al. [3] did a comparative study of CPU's, GPU's and FPGA's. They utilized two Intel Xeon 5650 CPUs, Xilinx Virtex-5 FPGA and NVIDIA's GeForce GTX460 and 9600 GTX+ GPU. They tested the performance of each platform using a wide range of benchmarks. In their research, they implemented a Fast Fourier Transform (FFT) in each platform and compared execution times. They found the FPGA to be 10.76 times faster than the CPU and 26.67 times faster than the GPU (when including transfer time for the GPU to send and receive data).

An even more extensive study was done by Che et al. [2]. They compared the performance of GPU's and FPGA's when doing computational intensive work for CPUs. They utilized the Intel Xenon CPU, Xilinx Virtex-II Pro FPGA, and NVIDIA's GeForce 8800 GTX GPU. They implemented and analyzed three applications for each platform, Gaussian Elimination, DES, and Needle-Wunsch. More specifically, Gaussian Elimination is a linear algebra computation that solves for all variables in a linear system. DES is a cryptographic algorithm that does mostly bit-wise operations. Needle-Wunsch is a dynamic programming algorithm used for DNA sequence alignment. For each platform, they measured the clock cycles to complete each application. Figure 2.3 and 2.4 shows the performance for all three computation platforms for the Gaussian and Needleman-Wunsh computations, respectively. The FPGA outperformed the CPU and GPU for all of the applications. They did not even bother to graph the results from the DES application because the FPGA took only 83 cycles, while the GPU took 5.8 x 10^5 cycles. For Gaussian Elimination computation, the GPU started to catch up to the FPGA for input sizes larger than 2048.

The conclusion of this work is that FPGA's can complete many applications in significantly less cycles than CPUs and GPUs. Note that the CPU, FPGA, and GPU are clocked at different frequencies, therefore, their cycle times differ. FPGAs run at the slowest frequency, therefore, they take the longest time per clock cycle. Also, they may have the longest cycle, but they generally take the least amount of cycles to complete the computations. This suggests comparable execution times to the GPU and CPU. It also suggests that with comparable performance, FPGAs reduce

power consumption since they run at lower frequencies. Based on these performance studies, it is clear that CPUs can gain tremendous performance increases when coupled with an FPGA.

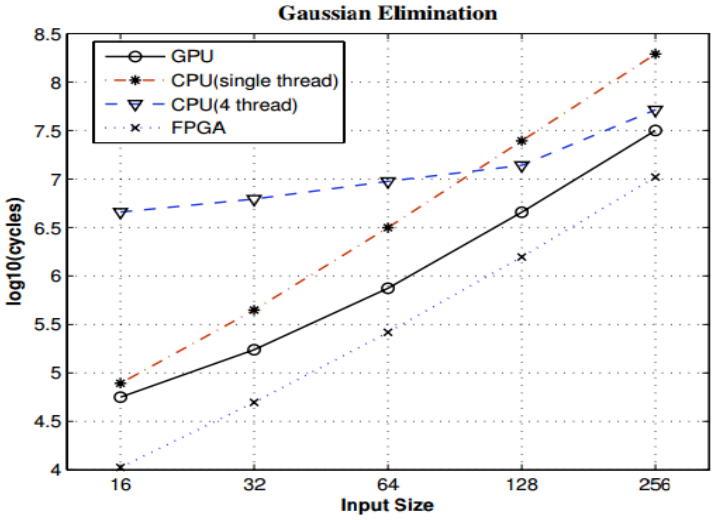

Figure 2.3: Execution cycles of the four versions of Gaussian Elimination. The x-axis represents the dimensions of the computation grid. Note that the y-axis is a log scale[2].

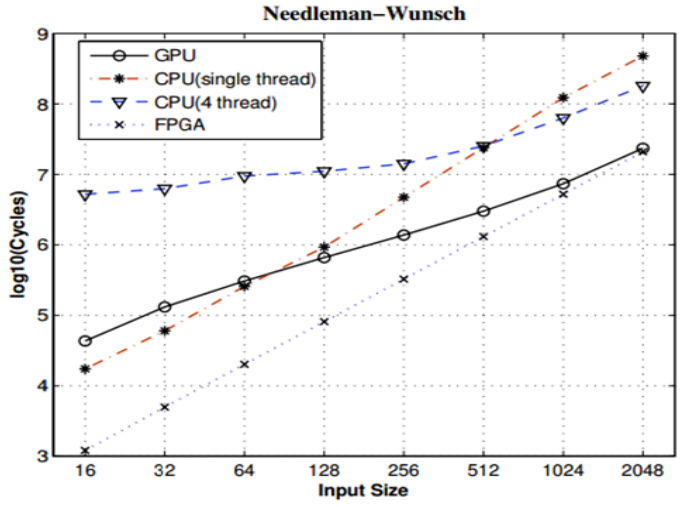

Figure 2.4: Execution cycles of the four versions of Needleman-Wunsch. Note that the y-axis is a log scale[2].

Chapter 3

Background

The case studies described in section 2 reflect the benefits that can be gained by introducing FPGAs to mobile phone platforms. The only obstacle is general RTR, which provides the ability to introduce new hardware and partially re-configure an FPGA at run-time to meet our mobile hardware deployment scenario. Before describing the potential solutions for general RTR, it is important to understand the deployment model and why it requires general RTR. There are three parties involved that make up the deployment scenario, the phone manufacturer, the application and hardware developer, and the user.

3.1 Phone Manufacturer

The mobile phone market is very large and there are a variety of phone architectures that include Apple's Iphone 6, Samsung's Galaxy S5, LG G3, and many more. If an FPGA is going to be placed in a mobile phone, it will be done by the phone manufacturer. They will specify the best FPGA architecture that suites their CPU architecture. Based on this, there is the likely possibility that there could be many hardware architectures. If developers want to design hardware for their applications, they will have to keep in mind the hardware architectures they are designing for. If there is to be any deployment model for hardware applications, there needs to be support for various hardware architectures defined by the phone manufacturer.

3.2 Application and Hardware Developer

The application and hardware developer, like any application developer, are unlikely to be directly associated with the phone manufacturer. The application and hardware developer could belong to Twitter, Facebook, Quora, etc. Since they are unrelated to the phone manufacturer, they need support for their hardware definitions and the type of hardware architectures they are loaded on. It can be very cumbersome for the developer to consider multiple hardware architectures and to design accordingly. They also need a way to deploy their mōbware so that the users (end-system) can download and use them. There needs to be support to allow developers to efficiently and quickly deploy their mōbware.

3.3 User

The user, which is also the end-system, is independent of the phone manufacturer and developer. They decide which applications and hardware they would want to run. They need support for application and hardware handling. They also need a way to explore and download mōbware onto their phones.

A basic representation for the hardware deployment scenario is shown in figure 3.1. The phone manufacturer is providing an FPGA base hardware design. This base design contains supportive hardware to allow modules to be reconfigured in a specified area on the FPGA. The areas that accept these reconfigurable modules (RM) are referred to as reconfigurable partions (RP).

The base design also contains hardware that allows for efficient communication between the processor and FPGA. This base design is then compiled into a static design. A static design is a place and routed hardware design with static hardware, hardware that is not reconfigurable, and RPs, areas fixed in the FPGA that house RMs. The phone manufacturer would upload their static design to the cloud. In parallel with the phone manufacturers, there are developers designing

hardware for their applications. These RMs must be compatible with the static design defined by the phone manufacturer. In figure 3.1, the static design and RM are uploaded to the cloud and compiled together. Once compiled and packaged together, the mōbware is now compatible with the users phone and can be deployed by the cloud.

The foundation of this deployment model is general RTR. It is not known what designs the developers will develop. The design must be compiled for the hardware architecture. We must also not forget about the user. They are consuming the hardware and cannot be kept waiting for that compilation. This is a unique scenario and we look to the tools and past academic work that could provide an architecture to support this scenario.

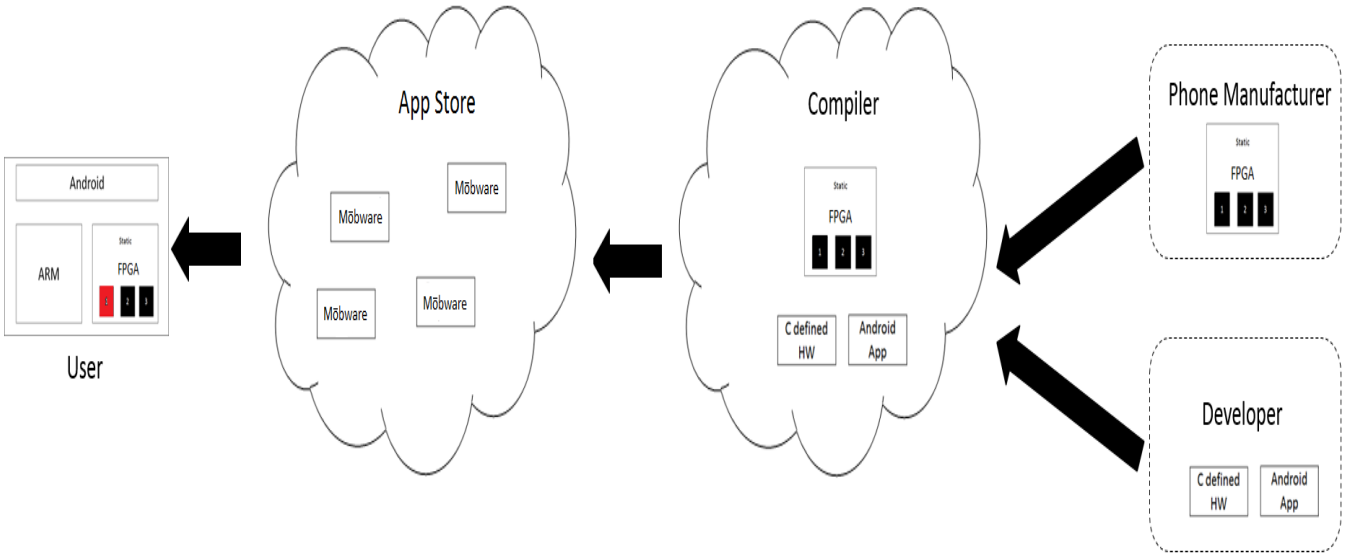

Figure 3.1: Mobile hardware deployment scenario.

Chapter 4

RelatedWork

There have been many attempts to achieve general RTR by the academic community and industry. Currently, the tools provided by FPGA vendors support only non-general RTR. In this section, we describe the current supported scenario of non-general RTR and the academic attempts at achieving general RTR. We also describe the tools made available by industry that support only non-general RTR and why today we can leverage those tools to support our deployment model.

4.1 Non-general RTR

The current supported scenario of RTR is non-general. Everything is known at compile-time and the hardware developer is the only party involved. They are responsible for defining the static design, designing the RMs, execution of Xilinx's partial reconfiguration tool flow, and managing the swapping of RMs in the end-system. For example, the hardware designer has a static design that utilizes a 15^{th} order FIR filter. They find that they want to process their data, instead, with a 20^{th} order FIR filter. They also want the flexibility of swapping the FIR filters at run-time. Using Xilinx's partial reconfiguration technology and tools, they can get this functionality. They can allocate an RP on the FPGA to house either FIR filter. The filters are then compiled for that RP and become RMs. The tools then generate partial bitstreams, which are files that can configure the RP with an RM. With the partial bitstreams, they can load either into the RP, giving them the flexibility of swapping the filters at run-time. All in all, Xilinx partial reconfiguration is a very useful technology, however, it does not provide the multi-party support required by our deployment

scenario. It also requires knowledge of designs at compile-time, making general RTR difficult to achieve.

4.2 Academic Approaches

There are two main approaches that have developed to support general RTR. They are the run-time place and route and slot-based approaches.

4.2.1 Run-time Place and Route

The general approach for run-time place and route is simple. The designer provides a source of hardware they want reconfigured into the FPGA. The form of the source can vary. It can be a hardware description language (HDL), netlist, or a partial bitstream, which is a stream of data that can configure portion of an FPGA. Next, the source is preprocessed at compile-time and then placed and routed onto the FPGA at run-time. To achieve fast run-time place and route, many tools have been built to manipulate already generated full bitstreams and change only the sections that correspond to the preprocessed source introduced at compile-time.

Guccione et al. [7] created the JBits software, an API to access Xilinx FPGA bitstream. Using JBits, a developer has the ability manipulate FPGA resources such as look-up-tables (LUTs), routing, and flip-flops (FFs). The API abstracts out manually setting or clearing bits in an FPGA bitstream, making resource configuring simpler. Also, the API handles all devices in the XC4000 and Virtex family. Finally, JBits manages the device bitstream and supports the reading and writing of bitstreams. By providing this functionality, JBits can support general RTR. After the source has been preprocessed, a full bitstream can be partially re-configured with the preprocessed source, at run-time. There are limitations to JBits, the most important being all bitstream configuration is done manually. All resources are explicitly stated and this can become a very cumbersome task, making general RTR very slow. Also, the hardware designer is forced to be very familiar with the

FPGA architecture. In our scenario, it is unreasonable to expect application developers to learn the the nuances of FPGAs and their underlying architecture.

To solve the limitations of JBits, Keller [8] introduced JRoute, an API for routing Xilinx FPGA devices. Using JBits as a foundation, JRoute can handle routing of resources at many levels of control, from single connection to auto-routing of a bus connection. There is also support for parameterizable hardware cores, making it even easier for users to simply define ports and routing happens automatically. JRoute eliminates the JBits limitations of manual routing and strong familiarity of the FGPA device. JRoute also contained support for unrouting. When a route is unneeded or routes have changed, the API frees the resources and updates accordingly. This makes general RTR much less difficult and time consuming, however, there are still limitations. JRoute, a great effort, still proved to be inefficient to support a robust general RTR. Routing, at times, took too long or unable to make the routes. In addition, routing was not optimal, causing the performance of the hardware to plummet.

Patterson et al. [13], using the same run-time place and route approach, introduced a slotless module-based reconfiguration software. At compile-time, the software precompiles hardware modules, by providing them with external and internal communication interfaces. The modified modules are then compiled into a module bitfile (partial bitstream). At run-time, this module bitfile is placed within a sand boxed region on the FPGA, allocated routing channels, and connected to other modules. Like JRoute, this software uses a bitstream manipulation tool (BitShop) to retrofit partial bitstreams onto the FPGA and connect them to other other modules and the static design. The advantage of this slotless module-based reconfiguration approach is they are optimally placing the modules, allowing for efficient routing. They also fix all communication, internally and externally, simplifying routing between modules and the static design. Like the fate of JRoute, this software was never adopted, due to not offering the robustness and usability required by hardware designers. Also, this work only supported a single device family.

4.2.2 Slot-based

Slightly more successful than the run-time place and route approach, the slot-based approach uses predefine reconfigurable partitions on the FPGA to load hardware modules. At compile-time, a source design is converted into a partial bitstream. The partial bitstream is then retrofitted into the predefined slots using a bitstream manipulation tool. The advantage of predefined slots is the reduction of work at run-time. Since a partial bitstream is being placed in a slot in the fabric, configuration has already been defined. There is no need to place and route an entire source design at run-time. Only minor modifications must be made to the full bitstream to accommodate the partial bitstreams into one of its slots.

Horta et al. [20] introduced the Dynamic Hardware Plugin (DHP), a module that can be loaded into or removed from an FPGA without disturbing the rest of running hardware. In order to realize DHPs, they utilize a tool called PARBIT that is able to modify and restructure bitstreams. When a DHP is compiled, PARBIT can dynamically place the module into any region of the FPGA. The next step is to isolate regions in the FPGA to house the DHPs. They use special wires called gasket antennas that fix communication between the non-reconfigurable part of the FPGA and the DHPs. Routing within an isolated area is restricted to only routing within that area and can only connect to rest of FPGA from antennas. This constraint is supported by a modified router. Finally, they wrap the VHDL module, soon to be DHP module, to contain the antennas and add constraints to synthesis, place, and routing to place the DHP modules and mitigate any routing through the DHP site. The main disadvantage with this technology is that it only supports a single device from the Xilinx Virtex-E family. It requires a lot of preprocessing of modules and bitstream manipulation, which all can be time consuming to do at compile and run-time, respectively.

Like the slot-based approach of Horta, Majer et al. [12] developed a dynamically reconfigurable FPGA-based computer (The Erlangen Slot Machine) with a slot-based architecture that

mitigated the limitations of general RTR. An FPGA is broken down into slots and constrained so no signals feed-through the modules. Each slot contains resources for inter-module communication, access to external shared memory, and access to all available peripherals. Since each slot has access to every available resource, hardware modules can be loaded into each slot and the proper connections are made that will give the module all the resources it requires. This also eliminates feed-through signals since all slot interfaces are fixed and external to all other slots. In addition, just like the other attempts at general RTR, a bitstream modifier tool is required to manipulate the fabric to accommodate the new hardware module. The Erlangen Slot machine uses a hardware reconfiguration manager to handle scheduling of loading modules, slot segmentation and partitioning, loading, unloading, and relocation of modules in slots. Similarly, just like the DHPs, the Erlangen slots are designed for a specific Xilinx FPGA device and requires multiple FPGA's for slot area and slot management.

4.3 The Time is Now

Both approaches, run-time place and route and slot-based, attempted to solve many of the same issues that surround RTR by constraining wire routing and modifying bitstreams to accommodate a diverse set of hardware in the same fabric location. Today, RTR is supported by Xilinx partial reconfiguration (PR), however, it is non-general. Fortunately, with the support of Xilinx PR, there are other tools and technology that can be leveraged to implement general RTR required by our mobile hardware deployment scenario described in figure 3.1, such as, Vivado High-Level Synthesis (HLS) [21] and the Xilinx's Zynq 7000 series system-on-chip (SoC) [22].

Partial reconfiguration is in mainstream tools: As discussed in the previous section, there has been lack of support in the mainstream tools for general RTR. Researchers were forced to develop their own tools and hardware to support an efficient and automated run-time reconfiguration of FPGA's. We have shown how researchers [23][8][7] were reverse-engineering and/or

altering full bitstreams generated by an FPGA vendors tools to accommodate fast RTR. Todays mainstream tools (those provided by the FPGA vendors) have full support for PR. They contain simple PR tool flows that handle floor planing and constrained placement and routing [24].They also contain a Tcl interface that we can leverage to make partial reconfiguration automated .

High level synthesis tools are available: It has been a long-held belief by many that developing hardware is hard (when compared to developing software). We have seen that a lack of usability leads to a tool not being adopted. For this reason, it is a requirement for application developers to design hardware in a high-level language, otherwise, developers would never attempt to design mōbware. Fortunately, high-level synthesis tools exist. They can accept the C/C++ language and convert it into an RTL design. Today, Xilinx (with Vivado's C and C++ based compilation [21]) provides high-level synthesis tools as a main design flow and are seeing commercial success. This tool provides libraries to instantiate hardware intellectual property (IP) in C/C++ design. They contain testing support, where designers can write test benches for their C/C++ defined hardware. Also, modules are easily exported from Vivado HLS to Vivado for partial reconfiguration.

FPGAs have embedded ARM Cortex A9: The main event that makes now the right time is that there exists a commercial off-the-shelf FPGA coupled dual-core embedded ARM Cortex A9 processor [22]. The ARM Cortex A9 (or its successor) is the same processor that is used in many smart phones today. With this, the FPGAs can be a drop-in replacement and be compatible with little or no changes to software today (in fact, these FPGAs can turn off the FPGA logic, making them simply a dual-core ARM processor, further illustrating the point).

Chapter 5

Cloud RTR Approach

The run-time place and route and slot-based approaches failed to provide an efficient general RTR that supports the deployment scenario described in section 3. Fortunately, the tools of today can be leveraged to create a new approach to achieve efficient and automated general RTR that allows mobile phone applications to have access to hardware. The approach is called the cloud RTR approach. Figure 5.1 shows the complete cloud RTR approach that converges the three parties described in section 3 (refer back for a detailed description of each party). The phone manufacture provides the static design to the cloud. The developer provides an Android application and C defined hardware to the cloud. The cloud compiler takes the hardware defined by the user, synthesizes it into a netlist, and compiles it for each RP in the phone manufacturer's static design. Once each RP is compiled, partial bitstreams are created for each RP, which later can be swapped in or out of their respective RP. The application and partial bitstreams are placed into the app store. A user can then download the application and hardware bitstreams required by that application. Android then chooses a RP to place the hardware and loads it with the respective partial bitstream.

As mentioned earlier, the tools and technologies of today are leveraged to implement the cloud RTR approach. More specifically, Xilinx PR is used to achieve RTR. HLS is used to convert C defined hardware into an RTL design that is later synthesized into a netlist and loadable into a RP. Finally, the deployment model of mōbware is leveraged to achieve the general RTR.

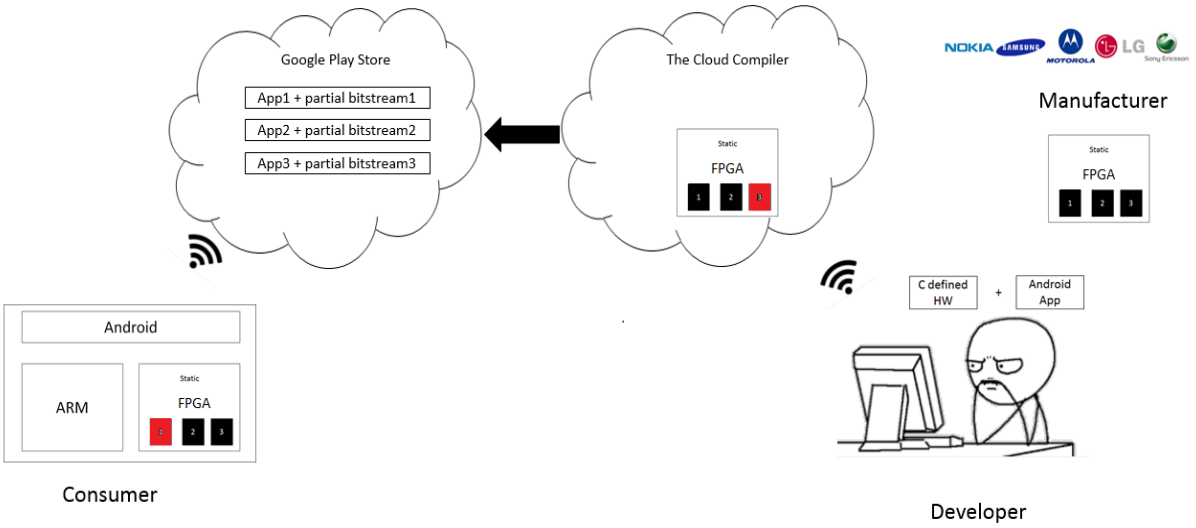

Figure 5.1: Cloud RTR approach to the generation and deployment of mōbware.

5.1 Brute Force Compilation

The Xilinx tools alone do not achieve general RTR. They support a very specific deployment scenario where all is known at compile-time. As described in section 4.1, their scenario also only supports a single party (Hardware Developer) that wishes to achieve more functionality without using up more space of the FPGA. We also described an example of different ordered FIR filters that could be accommodated in an RP.

The example shows that any time a new RM is introduced, it must be compiled for an RP so it can become swapped in and out at the developers convenience. If there are multiple RPs and the developer wanted an RM to be placeable in all RPs, then they would compile the RM into each RP. This brute-force method could allow all RMs to be compatible for all RPs allocated onto the FPGA. By compiling each RM for each RP, this provides the general RTR that is required by the cloud RTR deployment model.

More specifically, Xilinx's PR technology can be used to implement general RTR in the cloud-based deployment infrastructure by brute-force compiling the RMs, that are uploaded by

developers, on all the static designs provided by the phone manufactures. Since the cloud RTR approach converges the phone manufacturer, developer, and user, the brute-force method can be implemented in the cloud. The only potential limitations are the computational requirements to support brute-force compilation of RMs. It could take too long and too many machines to service a reasonable amount of RMs. These potential limitations are investigated later.

5.2 Xilinx Parital Reconfiguartion

Since we can leverage Xilinx's PR to achieve general RTR through brute-force compilation, we had to learn how to implement and automate partial reconfiguration. The first step is to define a static design. The static must contain hardware to support communication between the processing system (PS) (i.e. the processor) and PL. It must also contain hardware to support partial reconfiguration. Next, the static design is fitted with N black boxes. Black boxes are simply a defined module interface with no hardware logic. The next step is to complete base PR. Base PR is creating a base static design where the static region has been routed and RPs have been floor planned and are empty. With this static routed design, we can compile any RM for any RP, assuming it meets the PR requirements that we will describe later. The process of compiling RMs using the base PR design is called dynamic PR. RMs are dynamically compiled for every RP and partial bitstreams are created. These four tasks represent the nuances of partial reconfiguration and, with brute-force compilation, allowing for general RTR.

5.2.1 Define Static Design

Figure 5.2 is the static design we designed to accommodate partial reconfiguration and PS to PL communication. The PS block configures the ARM cores. It sets how the ARM cores communicate with the PL and which hardware modules it drives. There are many ways that the PS can communicate with the PL, most of which utilize the AXI4 communication protocol. AXI4 is a master/slave communication protocol and contains three interfaces.

- AXI4 - for high-performance memory-mapped requirements.
- AXI4-Lite - for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream - for high-speed streaming data [25].

The PS can only communicate through AXI4 or AXI4-Lite, suggesting that all communication transactions between the PS and PL are memory mapped. If an RM is not memory mapped, then a DMA is coupled with the RM to make it memory mapped. When the PS needs data processed by a streaming RM, it communicates an address to the DMA, the DMA extracts the data and sends it to the RM. The data is processed and returned to the DMA. The DMA then places the processed data in an area in memory that the PS can access. For this reason, we place a DMA next to every RM. This ensures that if the RM is stream-only hardware, the PS can still drive it. If the RM is memory-mapped, then the PS can directly communicate with the RM via the interconnect, shown in figure 5.2.

In addition, the AXI4 protocol is very flexible in that a master can drive multiple slaves using an AXI interconnect. An AXI interconnect gives the PS (master) control over all hardware in the PL, as long as they are connected to the interconnect as slaves. All hardware blocks to the right of the interconnect block, in figure 5.2, are slaves to the PS. The static design also contains a GPU that handles graphics for the Android OS. This block will be unneeded in practice, since mobile phones have dedicated GPUs to handle graphics, however, we need this block to display graphics on our emulation platform.

Finally, the static design contains a reset hardware block. When an RM is swapped with another RM, the new RM must reset. This ensures that the new RM starts in a good starting state and will function correctly. It also clears the hardware that may contain unwanted values that would disrupt function of the new RM. Normally, this is handled by the hardware itself. A

property known as RESET_AFTER_RECONFIG can be placed on every RM, forcing a reset after reconfiguration. Unfortunately, this did not meet the requirements for our static design.

All RMs have an AXI interface, meaning the state of the transactions between the AXI master driving the original RM remains even after reset of the new RM. During the first attempts at partial reconfiguration, loading of a partial bitstream by the PS would hang. To mitigate this issue, a reset block was implemented to put AXI communication of the RM into reset. When this happens, the PS notices that the RM communication interface is in reset and resets its own communication with the RM. Once communication on both sides have been reset, the original RM can be replaced by the new RM, without hanging the PS during loading of the partial bitstream. Note that the PS is driving the reset block. When partial bitstream is to be loaded, the PS first resets the RMs AXI communication.

Note the thin box around the RM, reset, and DMA hardware blocks. This is to signify that these blocks are based on the number of RMs that are defined in the static design. A reset and DMA block are needed for each RM. We later will discuss the resources utilized by the support hardware and its impact on the number of RPs we can place in the FPGA.

5.2.2 Static Design Black Box Instantiation

When defining the static design, we must place an N amount of black boxes to serve as place holders for RPs. Black boxes are hardware modules with only an interface and no internal logic. When a black box is synthesized, the tools recognize it as a black box and allow partial reconfiguration properties to be applied to it. Note that static design in figure 5.2 is designed in Vivado as a block design. Block designs are one level of abstraction above HDL. When a block design is compiled, it is converted into HDL. Once the block design has all the components in figure 5.2, N amount of hardware modules are added to serve as place holders for the black boxes. Since the HDL contains instantiations of N actual hardware modules, we must modify the HDL to reflect N

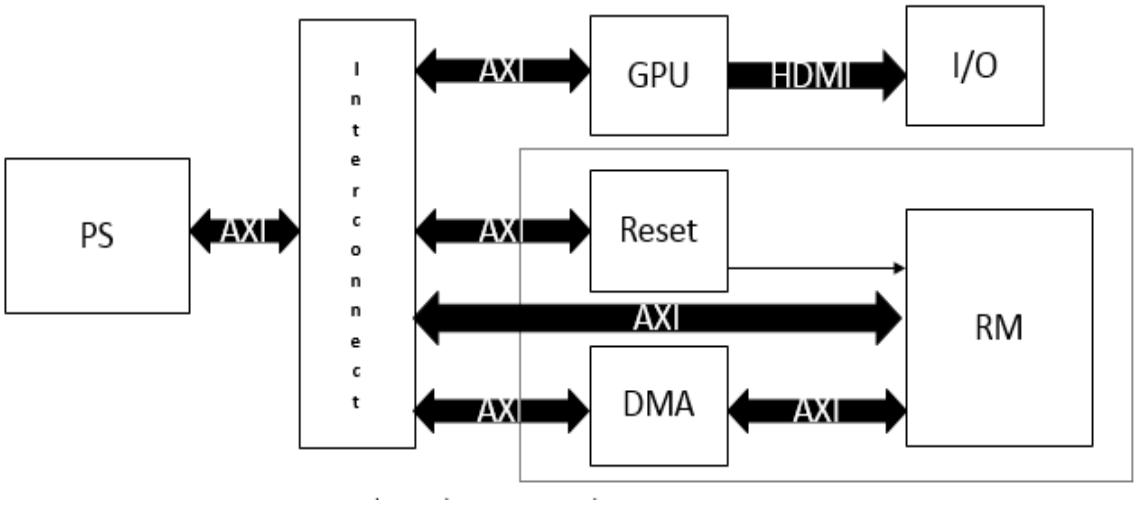

Figure 5.2: Static design that accommodates partial reconfiguration.

black boxes.

```
1  system_simple_fft_0_0 simple_fft_0
2  (.aclk(sys_100m_clk),
3   .ap_start(GND_1),
4   .aresetn(reset_axi_0_reset_out),
5   .in_r_TDATA(axi_dma_0_M_AXIS_MM2S_TDATA),
6   .in_r_TREADY(axi_dma_0_M_AXIS_MM2S_TREADY),
7   .in_r_TVALID(axi_dma_0_M_AXIS_MM2S_TVALID),
8   .out_r_TDATA(simple_fft_0_out_r_TDATA),
9   .out_r_TREADY(simple_fft_0_out_r_TREADY),
10  .out_r_TVALID(simple_fft_0_out_r_TVALID));
```

The verilog code snippet above is an instantiation of an FFT in a top-level design (static design). This snippet is also HDL that was generated from a block design. This top-level design does not use all of the ports of the FFT. If this FFT instantiation was turned into a black box and then an RP, all RMs compiled for that RP would be restricted to that interface. By using a reference design as a place holder, there is the risk that the reference interface can be shortened if the top-level design

does not use these ports. There are two solutions for this problem. The first is to add the missing ports before moving further in the PR tool flow. The second is to ensure that the RMs loaded into this eventual RP have the same interface. The latter is optimal since the top level design does not use these ports. For the sake of simplicity and efficiency, we do the former because it is less work to add missing ports than to modify the original design.

We have developed a tool that scans the top-level design for the reference design instantiations and adds the missing ports. It is given a top-level design HDL file and a reference design interface that defines what the interface should be and makes changes accordingly. The code snippet below shows the updated FFT instantiation. It now contains three added ports, ap_ready, ap_done, and ap_idle, which are absent from the original definition of the FFT hardware module.

To make this instantiation a black box, the module instantiation in the code below must be placed at the end of the top-level file. This will indicate to the Vivado that this module is a black box module and can be given partial reconfiguration properties. Note that there must be a module instantiation for each black box. In the code below, both instantiations would lead to a single black box. To accommodate more black boxes, we would add another FFT interface and FFT module instantiation. The tool described above also adds these module instantiations to the top-level design once it finishes adding the missing ports.

Interface matching between RMs and the static design is a requirement for PR to work. All RMs do not make any assumptions on the top-level environment. They are synthesized separately from the top-level design, which is known as out-of-context (OOC) synthesis. If the static design is to accommodate RMs, it must contain the correct interface, even if some of the ports are unused. For example, if the FFT with an interface below was OOC synthesized, all the ports would be visible in the generated netlist. If this FFT was placed in an RP that was defined by the interface above, PR of the FFT onto the FPGA would fail.

```
1  //Updated FFT interface
2  system_simple_fft_0_0 simple_fft_0
3  (.in_r_TVALID(axi_dma_0_M_AXIS_MM2S_TVALID),
4  .in_r_TREADY(axi_dma_0_M_AXIS_MM2S_TREADY),
5  .in_r_TDATA(axi_dma_0_M_AXIS_MM2S_TDATA),
6  .out_r_TVALID(simple_fft_0_out_r_TVALID),
7  .out_r_TREADY(simple_fft_0_out_r_TREADY),
8  .out_r_TDATA(simple_fft_0_out_r_TDATA),
9  .aresetn(reset_axi_0_reset_out),
10 .aclk(sys_100m_clk),
11 .ap_start(GND_1),
12 .ap_ready(ap_ready0),
13 .ap_done(ap_done0),
14 .ap_idle(ap_idle0));
15
16 //Module instantiation placed at the end of top level file
17 module system_simple_fft_0_0
18 #(parameter
19 RESET_ACTIVE_LOW = 1
20 )(input in_r_TVALID ,
21   output in_r_TREADY ,
22   input [32 - 1:0] in_r_TDATA ,
23   output out_r_TVALID ,
24   input out_r_TREADY ,
25   output [32 - 1:0] out_r_TDATA ,
26   input aresetn ,
27   input aclk ,
28   input ap_start ,
29   output ap_ready ,
30   output ap_done ,
31   output ap_idle );
```

5.2.3 Base PR

Base PR refers to building the base static design with RPs, so that new RMs can be compiled and partial bitstreams generated. The base partial reconfiguration tool flow executes the following:

(1) Convert black boxes of a synthesized static design into floor-planned RPs

(2) Fill the RPs with a reference design and place and route (i.e. implement the design)

(3) Save reference implementation for verifying future implementations

(4) Remove the RMs from the referent implementation

(5) Save static routed implementation for future implementation of RMs

(6) Generate a full bitstream

More specifically, the first step of base PR is configuring black boxes as reconfigurable. This is easily done by applying the following Tcl command to each black box:

```
1 set_property HD.RECONFIGURABLE true [get_cells <Black Box Module Name>]
```

Once each black box has been configured as reconfigurable, they are loaded with an RM. This is done to floor plan each black box. By loading hardware into the black boxes, the tools can then determine the correct amount of resources that would be required by that RM. Figure 5.3 shows the process of floor planning of black boxes into RPs, known to the tools as pblocks. Each square, outlined in purple, are a section of resources of the FPGA that are dedicated to any RM that is loaded into that RP. Also, the RPs are placed on clock boundaries (boxed in red). When an RM is removed from an RP and replaced with another, the new RM will reset itself. This is configured by placing the RP clock boundary and the following Tcl command:

```
1 set_property RESET_AFTER_RECONFIG true [get_pblocks <pblock Name>]
```

We briefly mentioned the RESET_AFTER_RECONFIG property in section 5.2.1. This indicates to the tools that we want reset to occur after reconfiguration. Setting the RP on the clock boundary is a prerequisite. It is imperative to have this property, otherwise, will have to manually reset the entire RM. Even though there is a manual reset for AXI transactions, it does not reset the entire RM.

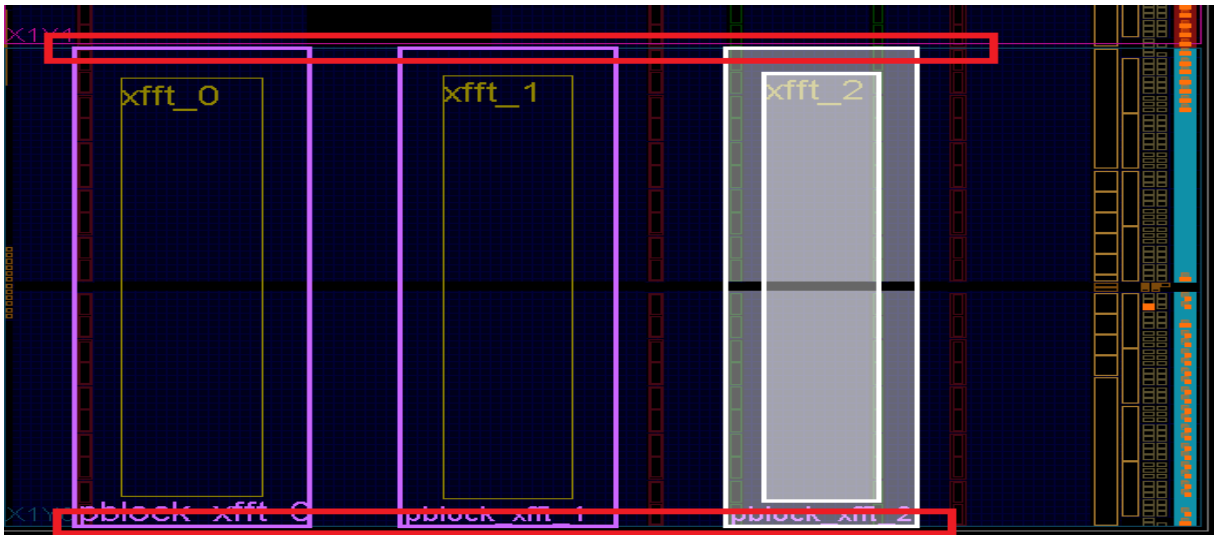

Figure 5.3: Floorplanning of reconfigurable partitions.

Once floor planning is finished, the entire design is placed and routed. This is done to create a base implementation that can be used to verify future implementations with different RMs loaded into the static design. Next, the RMs are carved out of the RPs and this base design is saved as a checkpoint. At this point, we have a checkpoint that has routed all of the static design and contains empty partitions for other RMs. This checkpoint can be used to compile other RMs. There is no need for floor planning and defining black boxes as reconfigurable for RMs we wish to compile in the future. Finally, we generate a full bitstream. This full bitstream will serve as the out-of-the-box configuration for the FPGAs in the mobile phones.

In addition, base PR is generally only run once. If there are updates to the static design, then it must be done again. With respect to our cloud RTR approach, we expect this to happen

infrequently. Also, to support the case of static design updates, we have automated this process. We developed a tool that contains the parameters for floor planning and looks in a specific directory for an RM to place in the newly defined RPs.

5.2.4 Dynamic PR

Dynamic PR refers to the partial reconfiguration tool flow when a new RM is introduced and must be compiled for every RP in the FPGA. This process is very similar to base PR and executes the following:

(1) Lock the design

(2) Load each RM into every RP

(3) Place and route (i.e. implement the design)

(4) Verify

(5) Generate partial bitstreams

More specifically, dynamic PR starts with the checkpoint created by base PR, a routed static design with RPs. First, the routing in the check point is locked down with the following command:

```
1 lock_design -level routing
```

This provides routing consistency when new RMs are compiled for each RP. The next step is to load the new RM into every RP and then placed and routed. Now that the design with the new RM has been implemented, it can be compared to the implementation from base PR. If there is any inconsistencies, there has been an error in the partial reconfiguration tool flow. If verification passes, bitstreams are generated. A partial bitstream is generated for each slot. To load the RM

into a slot, it's respective partial bitstream must be loaded in.

In addition, in order to load an RM into the RP, it must be synthesized OOC. When hardware is uploaded to the cloud, it comes in the form of the C language. The C defined hardware must be synthesized by HLS, so it can be placed into the RPs. This adds to the overall overhead of dynamic PR. This overhead will also vary depending on the size of the RM.

Similarly, like base PR, we have automated this process. Our tool generates the script to compile an RM for each of the available RP's defined in the static routed design. It verifies against the base PR implementation and generates the partial bitstream. All in all, the tool expects the OOC synthesized RM and the static routed checkpoint to complete dynamic PR.

Chapter 6

Evaluation

By building tools to complete base and dynamic PR, we have the tools needed to support the cloud RTR deployment infrastructure shown in figure 5.1. We have implemented the cloud RTR approach and ran three experiments to answer two fundamental questions. The first question is, how many RPs will be available to the developer/user? Throughout this paper, we refer to N RPs and never an actual fixed amount. The second question is, is the brute-force method practical? What throughput is achievable using the brute-force RM compilation method and can this throughput handle daily RM uploads? In this section, we describe the three experiments and the answers to the two fundamental questions.

6.1 How many RPs will be available to the developer?

So far in this work, we have not described how many RPs will be available to the developers/users. This is a very difficult question to answer without a reference, therefore, we developed a hardware module to gain insight on its resource utilizations. Using this information, we determined how many reference hardware modules can be placed onto the fabric. With this information, we learned more about how many and where to place RPs.

6.1.1 Experiment 1

6.1.1.1 Experiment 1 Setup

The first experiment is aimed at determining how many RPs can be placed on an entire FPGA using a reasonably useful hardware design. We developed an FFT hardware with the following characteristics:

- 1024 samples/frame
- single channel
- 16-bit data width

We placed as many FFT's on the FPGA fabric to get an idea of how many RPs can be accommodated for reasonable hardware designs uploaded by developers. An FFT hardware module is a very useful hardware module for digital signal processing. It is also a common benchmark used when testing for performance and power reduction [3]. We manually execute the base PR approach described in section 5.2.3. We instantiated eight black boxes with our FFT's interface. We synthesized the design and floor planned RPs for each of the black boxes. Note that instantiating eight boxes is an educated guess. At this point, we do not know how many FFT's can be placed. We will not know until the FFT is loaded into each black box and RPs are allocated based on the resource requirements for the FFT. We then do this and exhaust all available resources on the FPGA.

6.1.1.2 FPGA Resources

FPGA resources can be broken down into the following categories:

- Slice Logic
- BRAM
- DSP

Table 6.1 shows the amount of resources available for each category. Each slice is comprised of look-up tables (LUT), multiplexers (Muxes), adder logic, and registers. Memory is comprised of block RAM tiles. Each tile contains two 18 Kb RAM components. Also, there are also DSP components specifically designed for signal processing applications.

Site Type	Available
Slice LUTs	53200
LUT as Logic	53200
LUT as Memory	53200
Slice Registers	106400
Registers as Flip Flop	106400
Registers as Latch	106400
F7 Muxes	26600
F8 Muxes	13300
Block RAM Tile	140
RAMB36/FIFO	140
RAMB18	280
DSPs	220

Table 6.1: Slice Logic Resource

All three types of resources are utilized by the FFT. This limits the placement for the RPs to areas on the FPGA with all three resources. Figure 6.1 shows the layout of FPGA. The red columns are the BRAM components, the green columns are the DSP components, and the blue are the slice components. When allocating RPs, as done in figure 5.3, the block must encapsulate enough of all three resources to satisfy the requirements for the FFT. If it does not, a design rule check will fail and give an error.

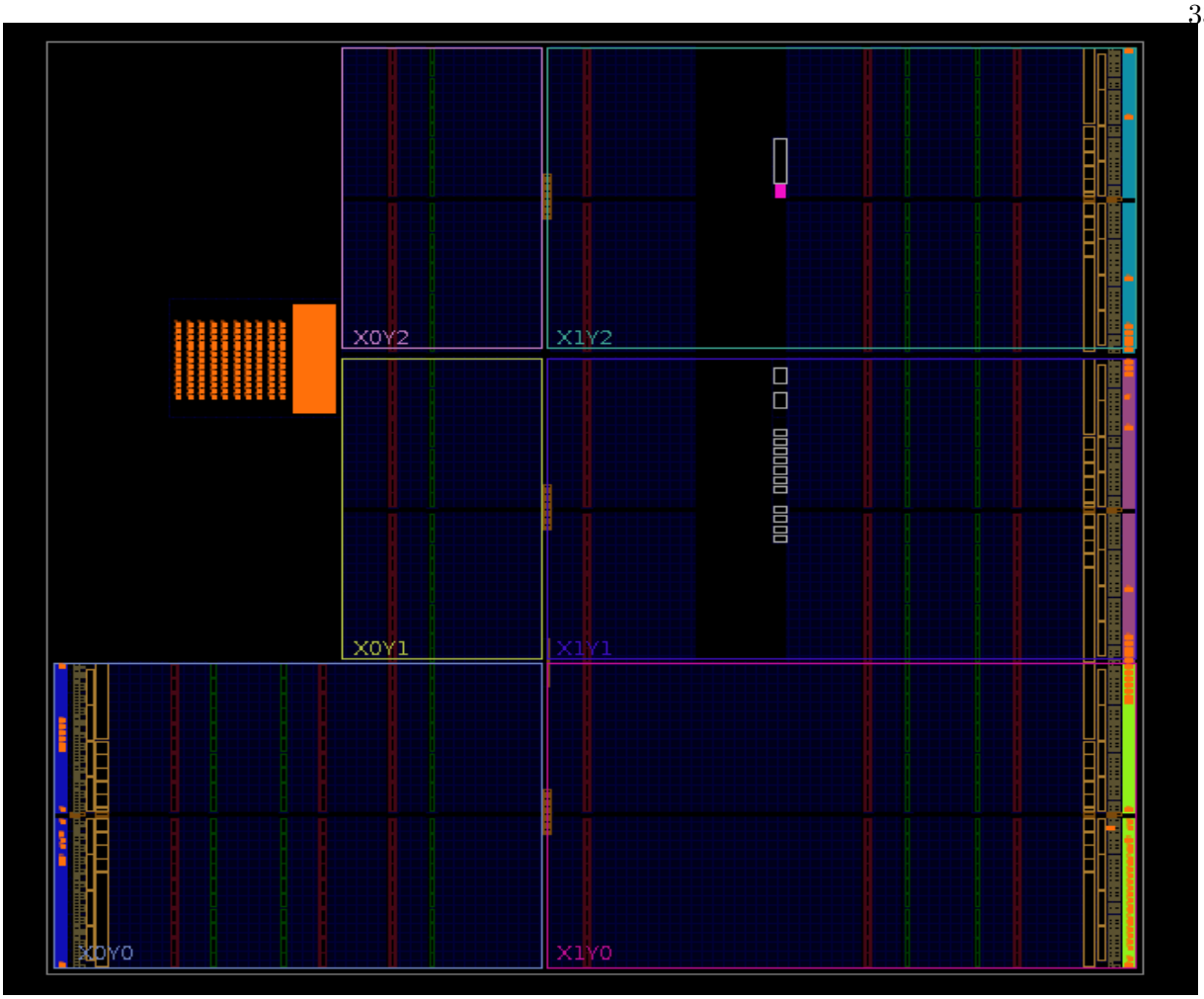

Figure 6.1: Zynq FPGA resources layout

Table 6.2 shows all the resources required by the FFT. Based on these values, we have a rough idea on how large the RP's must be to accommodate the FFT. Keeping in mind the clock boundaries, FFTs do not require spanning multiple clock boundaries. They can be safely placed within a single clock boundary and widened until they have the correct amount of resources. The clock boundaries are the colored horizontal lines in figure 6.1. They serve as a constraint to how to place the RPs vertically.

We discussed earlier the need for hardware support for RMs. We introduced the need for reset hardware and a DMA for each RM. This suggests that as the number of RMs grow, the number of

reset and DMA modules grow. We measured their resource utilizations and we have better idea on the number of FFTs that can be placed as RPs. Tables 6.3 and 6.4 show the resources required by the reset and DMA hardware, respectively. The reset block hardly uses any resources, therefore, it is negligible. The DMA uses 2.21 % of the Slice LUTs, 1.51 % of Slice Registers, and 1.42 % of the BRAM. We must add these utilizations to the FFT, since they are added for each RM.

Note that the resource utilizations for the DMA and reset hardware are more accurate than the FFT resource utilizations. When laying out RPs, they are based on the synthesized estimates of the FFT. Synthesized estimates overestimate the usage of the slice resources. We present the FFT's synthesized estimates because this is what floor planning is based on. The DMA and reset utilizations presented were measured after optimization, placement, and routing of the netlist.

Site Type	Used	Available	Util %
Slice LUTs	3258	53200	6.12
LUT as Logic	2055	53200	3.86
LUT as Memory	1203	53200	6.91
Slice Registers	5018	106400	4.71
Registers as Flip Flop	5018	106400	4.71
Registers as Latch	0	106400	0.00
F7 Muxes	66	26600	0.24
F8 Muxes	32	13300	0.24
Block RAM Tile	1.5	140	1.07
RAMB36/FIFO	0	140	0
RAMB18	3	280	1.07
DSPs	12	220	5.45

Table 6.2: Synthesis resource utilization for FFT RM.

Site Type	Used	Available	Util %
Slice LUTs	40	53200	0.07
LUT as Logic	40	53200	0.07
LUT as Memory	0	53200	0.00
Slice Registers	72	106400	0.06
Registers as Flip Flop	72	106400	0.06
Registers as Latch	0	106400	0.00
F7 Muxes	0	26600	0.00
F8 Muxes	0	13300	0.00
Block RAM Tile	0	140	0.00
RAMB36/FIFO	0	140	0
RAMB18	0	280	0.00
DSPs	0	220	0.00

Table 6.3: Implementation resource utilization for reset hardware.

Site Type	Used	Available	Util %
Slice LUTs	1178	53200	2.21
LUT as Logic	1139	53200	2.14
LUT as Memory	39	53200	0.22
Slice Registers	1612	106400	1.51
Registers as Flip Flop	1612	106400	1.51
Registers as Latch	0	106400	0.00
F7 Muxes	0	26600	0.00
F8 Muxes	0	13300	0.00
Block RAM Tile	2	140	1.42
RAMB36/FIFO	2	140	1.42
RAMB18	0	280	0.00
DSPs	0	220	0.00

Table 6.4: Implementation resource utilization for Direct Memory Access (DMA) hardware.

Figure 6.2 shows the FFT's placed onto the FPGA as reconfigurable modules. The purple outlines are the resource boundary for each FFT. The blue are the routed resources used by that FFT. Anything blue outside of the purple outlines are the resources used by the static design. At this point it is clear to see that a majority of the FPGA resources are being used up. There are 8 purple boxes, meaning 8 FFTs have been accommodated into this design.

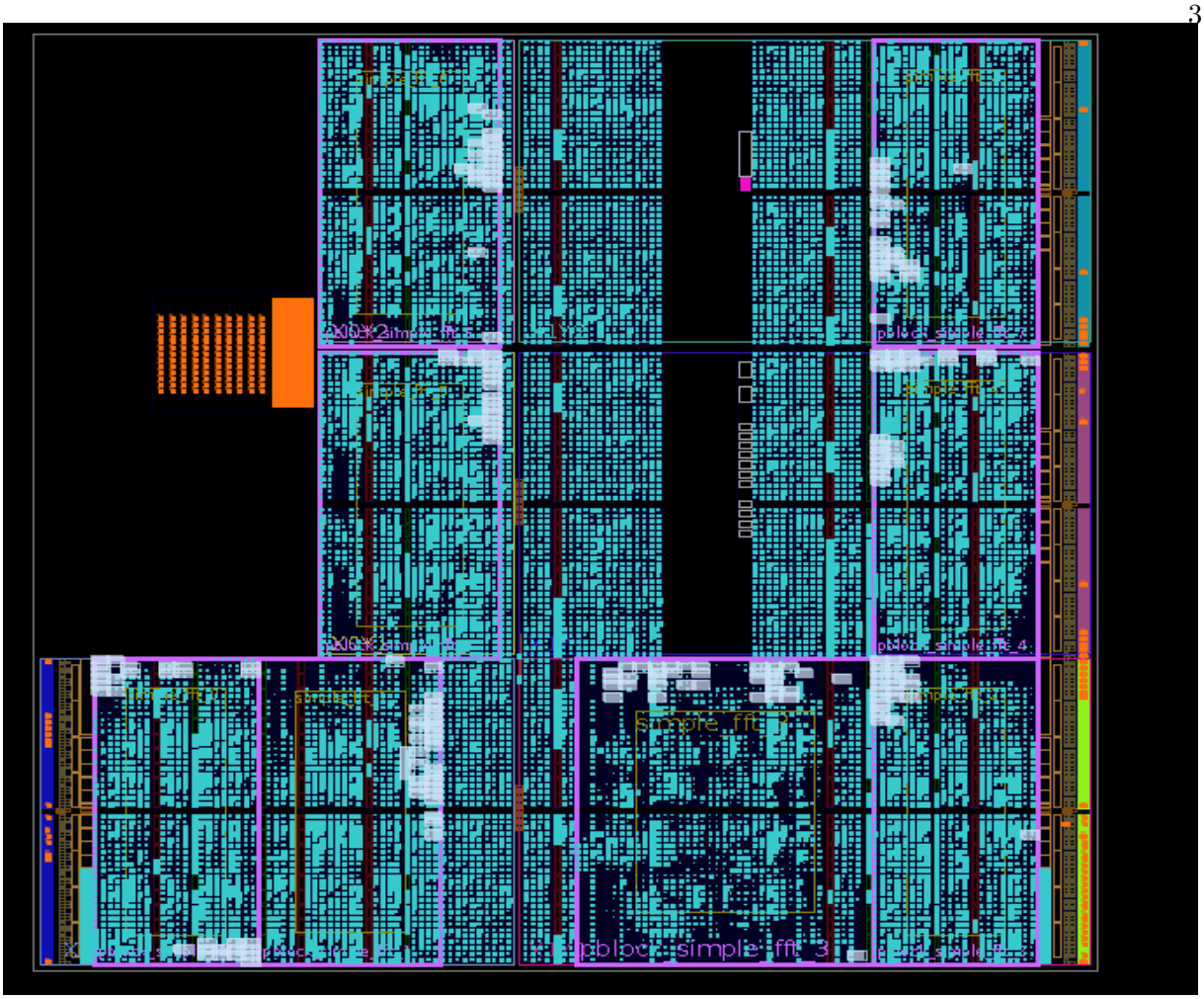

Figure 6.2: Zynq FPGA resources layout filled with FFT hardware modules.

6.1.1.3 Discussion

Experiment 1 has lead to development of a few design rules to consider when allocating RPs. We recommend that the phone manufacturer allocate all three resources to all RPs. This ensures that each slot can accommodate a wide variety of designs. Since the design is not known at compile-time, they must anticipate a diverse set of hardware and allocate all types of resources to accommodate the hardware diversity. We also recommend using less and larger RPs. We do not anticipate that a phone will be utilizing eight applications that utilize eight different hardware designs. It is more likely that there will be less amount of applications at one time using hardware, such as, two or three. With this realization, reducing the number of RPs and increasing their area

suggests larger hardware designs can be accommodated with out reducing functionality requirements.

In addition, the less RMs that are placed, the less overhead incurred by adding DMA hardware blocks. Area of the FPGA can be conserved using less RPs and more area of the FPGA can be dedicated to the RM. Based on the resource utilizations, the slices seem to be the most sought. The DMA modules primarily use slices and using less of them frees this resource. It also possible that DSPs and BRAM can be a bottleneck, specifically because of their location on the FPGA. Since they are so close to each other, large RPs would consume BRAM and DSP components that it might not even use. This waste can potentially limit the number of RPs that can be placed on the FPGA and the types of designs that can loaded in the partitions.

6.2 Is the brute-force compilation method practical?

In order to achieve general RTR, we are using a cloud deployment model to brute-force compile every loaded RM into every RP in the static design(s). If brute-force compilation takes too long, it is not practical to implement the cloud RTR deployment model. We must determine how much mōbware can be serviced each day and compare to the actual mobile uploads to an application cloud such as Google's Play Store.

6.2.1 Experiment 2

6.2.1.1 Experiment Setup

The second experiment is aimed at determining the elapsed time, CPU time, and memory usage for brute-force compilation of RMs during the base PR automated tool flow. Since we have determined we can have up to eight RPs with the FFT, we test brute-compilation for 2 to 8 RPs. We use the same FFT from the first experiment as our RM for base PR.

For this experiment we use the following hardware:

- Intel Xenon CPU 2.1 GHz (6 cores with 48 GB RAM)

6.2.1.2 Results

# of RPs	Elapsed Time (m:s)	CPU Time(m:s)	Memory Usage (MB)	Tcl Generation (s)
2	7:55	11:48	1986	0.128
3	11:56	18:19	2088	0.127
4	16:03	22:27	2187	0.127
5	22:15	29:37	2300	0.128
6	28:39	38:10	2391	0.123
7	36:00	45:59	2477	0.128
8	46:03	60:14	2584	0.128

Table 6.5: Execution times and memory usage for base PR compilation.

6.2.1.3 Discussion

Table 6.5 shows the execution times and memory usage required to compile an FFT RM into 2 to 8 RPs. Memory usage is in the range of 1.9 to 2.6 GB. Our system has 48 GB of RAM, which is more than enough memory to accommodate the base PR tool flow, no matter the number of RPs. Since at most 2.6 GB is needed for base PR, memory usage will not be a bottleneck of the execution of base PR in the cloud. Also, the process of generating the Tcl scripts that automate the base PR process is negligible. It takes less than a second to generate the scripts for up to 8 RPs, therefore, automation will not affect the elapsed time of base PR.

Similarly, table 6.5 shows the elapsed and CPU times for base PR. The Vivado tool that executes base PR is primarily single threaded, however, it spawns multiple threads during execution when work can done parallel, such as, when completing design rule checks. The multi-threaded

element of Vivado has a significant impact on elapsed time, as shown in table 6.5. For 8 RPs, more than eight minutes was shaved off the elapsed time. Even for two RPs, more than 3 minutes are shaved off. This time is precious and can be used towards compiling more RMs.

Elapsed time is the strongest indicator for the viability of base PR, however, there is a lot of flexibility in how long base PR can take. As discussed earlier, base PR is not executed many times. It needs to only run once to obtain the static routed design for dynamic PR. If there are updates to the static design, which happens infrequently, then it would need to be run again. 46 minutes of compilation time a few times a year is not a significant cost. A low cost for compilation of RMs of most of the FPGA resources suggest that 46 minutes is roughly the cost for compiling over the entire area of the FPGA. If the number of RPs were reduced, but their areas were increased, it should roughly take 46 minutes to compile.

Figure 6.3 shows the elapsed times of base PR for 2 to 8 in bar graph form. Notice that the trend is mostly linear, with the largest gap between 7 to 8 RPs. During the process of base PR, we noticed most of the time was consumed by routing each of the RMs. The more FPGA resources used, the more contention there is for routing resources. We also noticed that depending on where the RPs are placed, routing contention can increase or decrease for the same amount of area. For example, when executing base PR for 4 RPs, the elapsed time went past an hour. We then moved one of the RPs to a new location and the time went down to approximately 12 minutes. If RPs are optimally placed, routing can significantly be reduced. Since routing is the longest task, reducing its elapsed time would significantly reduce the overall elapsed time.

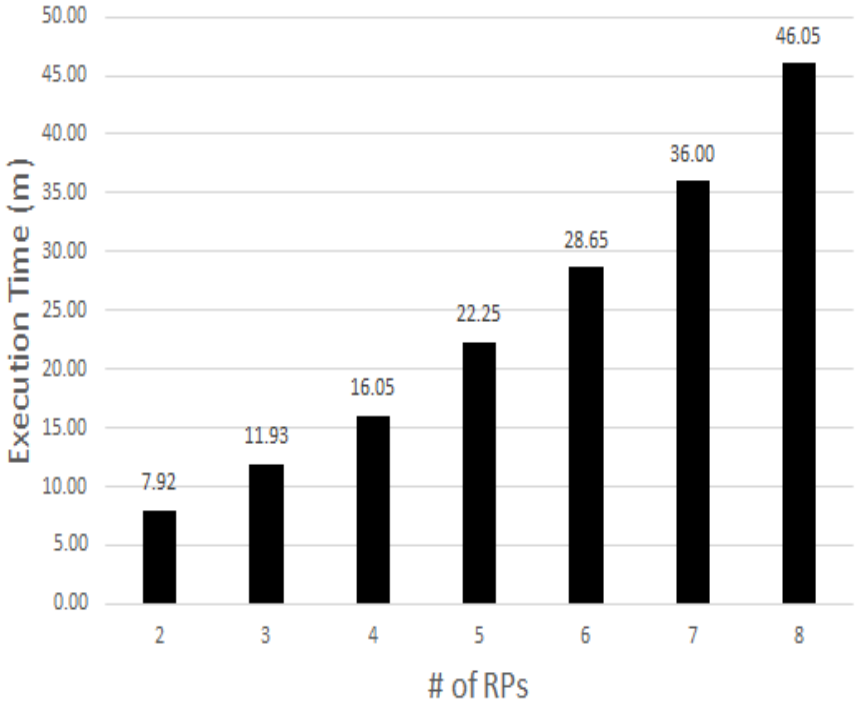

Figure 6.3: Execution times for performing base PR for 2 to 8 RPs.

6.2.2 Experiment 3

6.2.2.1 Experiment Setup

The third experiment is aimed at determining the elapsed time, CPU time, and memory usage for brute-force compilation of RMs during the dynamic PR automated tool flow. Since we have determined we can have up to eight RPs with the FFT RM, we test brute-compilation for 2 to 8 RPs. We use the same FFT from the first experiment as our RM for dynamic PR. We also measure the execution times for synthesis and memory usage required to convert the C defined FFT into an IP and synthesized. Dynamic PR is the tool flow that is continuously run, therefore, the overhead of its compilation will dictate if it is sustainable.

For this experiment we use the following hardware:

- Intel Xenon CPU 2.1 GHz (6 cores with 48 GB RAM)

6.2.2.2 Results

Elapsed Time(s)	CPU Time (s)	Memory Usage (MB)
48.56	61.532	67

Table 6.6: HLS synthesis execution times and memory usage of C defined FFT.

Elapsed Time(s)	CPU Time (s)	Memory Usage (MB)
174	177	1316

Table 6.7: Vivado synthesis execution times and memory usage of FFT IP.

# of Slots	Elapsed Time (m:s)	CPU Time(m:s)	Memory Usage (MB)	Tcl Generation (s)
2	8:13	11:29	2445	0.128
3	11:14	15:34	2532	0.127
4	15:19	21:05	2656	0.127
5	20:32	27:32	2811	0.127
6	24:57	33:26	2987	0.127
7	$>$ 120:00	$>$ 120:00	3088	0.122
8	$>$ 120:00	$>$ 120:00	3201	0.127

Table 6.8: Execution times and memory usage for dynamic PR compilation

6.2.2.3 Discussion

Every time a C defined hardware module is uploaded to the cloud, it must be converted into an IP. Once an IP, it can be synthesized by Vivado and is loadable in RPs. Table 6.6 shows the elapsed time, CPU time and memory usage for synthesizing and exporting a C defined FFT in HLS. Not much memory is required for this process, therefore, it is negligible. Like Vivado, HLS utilizes multiple threads when work can be distributed. It reduces time by roughly 12 seconds, which is

significant since the entire process only takes 49 seconds. The elapsed time is short, however, it cannot be ignored. We must add this time to the total overhead of dynamic PR.

After HLS has synthesized and exported the IP, Vivado synthesizes the IP OOC. Table 6.7 shows the elapsed time, CPU time and memory usage for synthesizing the exported IP in Vivado. This process required 1.32 GB of memory usage. More than enough memory is available and, like base PR, memory will not be a bottleneck. As mentioned earlier, Vivado can utilize multiple threads, however, it did not help much with reducing the elapsed time. Synthesis of the FFT IP took 174 minutes and this will also have to be added to the overall overhead of dynamic PR, since every new C device loaded to the cloud must go through this process.

Table 6.8 shows the execution times and memory usage required to compile an FFT RM into 2 to 8 RPs. Memory usage is in the range of 2.4 to 3.2 GB, which is 0.6 GB more than what is required by base PR. Even though it takes more memory than base PR, the 48 GB RAM available is still more than enough to accommodate the memory requirement. Memory still will not be a bottleneck. Also, the process of generating the Tcl scripts that automate the dynamic PR process still remains negligible. It still takes less than a second to generate the scripts for up to 8 RPs, therefore, automation will not affect the elapsed time of dynamic PR.

Similarly, table 6.8 shows the elapsed and CPU times for dynamic PR. The multi-threaded element of Vivado has a significant impact on elapsed time, as shown in table 6.8. For 6 RPs, more than eight minutes was shaved off the elapsed time. Even for 2 RPs, more than 3 minutes are shaved off. Since dynamic PR is run every time a new RM is introduced, time is very precious. Reducing elapsed time will increases the number of RMs that can be serviced each day.

Elapsed time is the strongest indicator for the viability of dynamic PR, unlike base PR, there is no flexibility in the execution of this tool flow. The longer dynamic PR takes, the less RMs

that can be serviced. If the elapsed time is too large, then the entire cloud RTR approach is not feasible. The elapsed times for 7 and 8 RPs are listed as greater than 120 minutes, as shown in table 6.8. Routing the resources for 7 to 8 RPs takes a significant amount of time because the nets are fighting for routing resources. It is likely a result of locking down the static design. There are much less routing resources and this causes great contention of those resources when routing the RMs. The contention becomes too great and the FPGA takes too much time to route the nets. Based on this, it is wise to only use up to 6 RPs of the FPGA, which is approximately 80% of the FPGA. Using too much area of the FPGA increases the time for routing to complete and it reduces the tool's throughput.

Even with a limit of 6 RPs (i.e. 80% of the FPGA), the elapsed time to compile 6 RPs is approximately 25 minutes. This is a promising result, in that many RMs can be serviced in a 24 hour period. Figure 6.4 shows the execution times of dynamic PR for 2 to 6 RPs in graph form. The elapsed times for dynamic PR execution was very similar to base PR. It has a linear trend and takes slightly less time. Even though there is a verification step in dynamic PR, routing of the static design in base PR is more expensive. At this point, we have determined the latency for dynamic PR and can calculate its throughput. Using its throughput, we can compare to workloads and conclude if it can service these workloads. With this information, we can determine if the brute-force method is practical.

In addition, not all RPs are created equal. Depending on where RPs are floor planned, the time to route resources can vary, just like in the base PR tool flow. To mitigate this issue, we could use a larger FPGA. By introducing an FPGA with more resources, routing contention would be much less. We could accommodate larger designs and spend less time on routing. It would also be advantageous to experiment with different slot placements to optimize the time to route the resources. We also hope to investigate the way resources are routed to find ways of reducing compilation time. All in all, the time for dynamic PR seem reasonable. With these results, we can

determine if the brute-force method is sustainable by a reasonable number of machines.

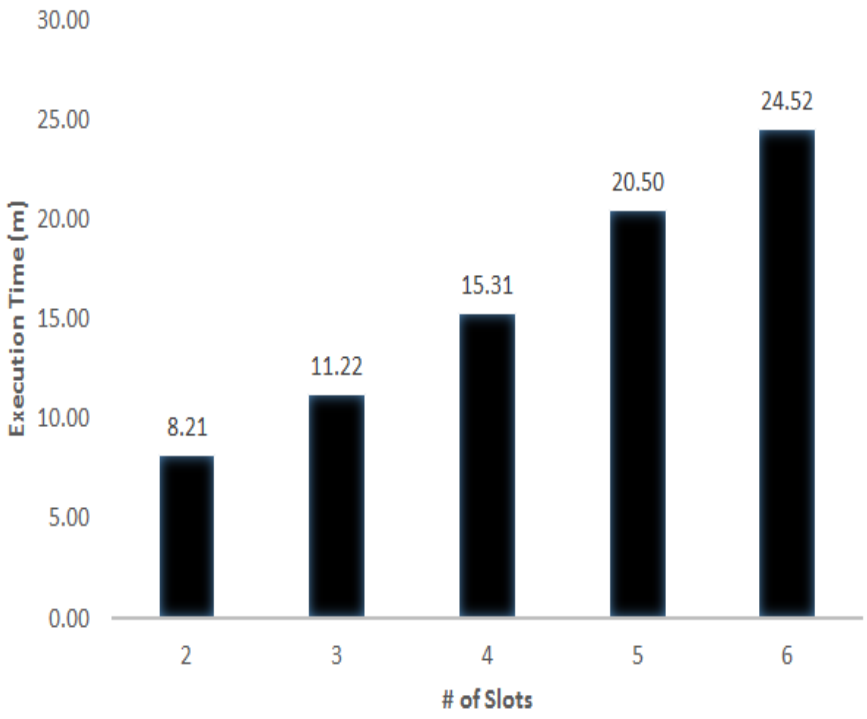

Figure 6.4: Execution times for performing dynamic PR for 2 to 6 RPs.

6.2.3 Cloud RTR Resource Requirements

Using the results from experiment 3, we can determine the daily throughput for compiling mōbware. Also, by making assumptions about application production each month, we can determine how many machines would be required to sustain our cloud RTR deployment model. The company AppFigures was kind enough to provide the Google Play Store application upload figures for the entire year of 2014. Table 6.9 shows the number of applications in the Google Play Store as of December 2014. Also, based on the numbers provided by AppFigures, we calculated an average monthly growth in uploaded applications to be 6.10%. Using this average monthly app growth, we extended it to April 2015. For the month of April in 2015, 104,187 applications are predicted to be uploaded into the Google Play Store. Based on the anticipated number of applications that

Google Play Store Figures	
Number of Apps as of Dec 14	1.43 Million
Average Monthly App Growth	6.10 %
Number of Apps uploaded for April 15	104,187

Table 6.9: Google Play Store app upload figures provided by AppFigures.

will be uploaded in the month of April, we can make assumptions on the percentage of applications that will use mōbware per month. With this assumption, we can compare it to the throughput of our machine and extrapolate the number of machines required to service the uploaded mōbware each day. Table 6.10 shows the latency for compiling a single application. Using these numbers we calculated the daily throughput, which is shown in table 6.11. We are able to service a maximum of 121 RMs per day for 2 RPs and a minimum of 51 RMs per day for 6 RPs.

# of RPs	Execution Time (m)	RM Synthesis Execution Tim (m)	Total Execution Time (m)
2	8:21	3.71	11.92
3	11:21	3.71	14.93
4	15:31	3.71	19.02
5	20:50	3.71	24.21
6	24:52	3.71	28.23

Table 6.10: Compilation components for dyamic PR and RM synthesis.

# of RPs	Execution Time (m)
2	121
3	96
4	76
5	59
6	51

Table 6.11: Daily throughput of apps compiled per day.

Using the calculated throughput, we can determine if the brute-force approach is sustainable for the current app market. To do so, we must make two assumptions. The first assumption is percentage of apps that will utilize the re-programmable hardware. We choose three values, 0.1%, 1%, and 10%. We do not anticipate more than 10% of applications uploaded by developers would use mōbware. Using this assumption, we can determine how many apps of the 104,187 uploaded to the play store in April would need to be compiled. From this number, we can easily determine how many hardware modules per day will need to be compiled and compare this value to our throughput. The second assumption is the number of phone variants. We have discussed earlier that there are many phone manufactures that will define the static design for their phone's FPGA. This suggests that the RM will have to be compiled for each phone variant. For this assumption, we assume from 1 to 1000, with each interval increased by a factor of 10.

Table 6.12 shows the number of machines required to service monthly demands for compiling mōbware, when 2 RPs are used. As discussed earlier, we choose a broad scale of the percentage of monthly apps that require RMs to be compiled (i.e. 0.1%, 1%, and 10%). Table 6.12 shows the number of RMs uploaded each day based on the percentage of applications that require hardware. For example, if 0.1% of applications for the month of April require RMs to be compiled, 4 (rounded 3.47) apps would be uploaded each day. A single machine compiling these RMs on a static routed design with 2 RPs can handle 121 RMs per day, as shown in Table 6.11. If there was only one phone architecture, only one machine would be needed to compile the 4 apps. Table 6.12 shows a broad spectrum of scenarios. The most expensive scenario is 1000 phone architectures and 10 % of the applications uploaded for the month of April utilize mōbware. For this scenario, 2875 machines would be needed to accommodate the workload.

Table 6.13 makes the same assumptions, however, a static routed design of 6 RPs is considered. More RPs means longer compilation time, as shown in table 6.11. When the static design has 6 RPs, it's throughput is 51 RMs per day. Since the throughput is much less, more machines

are required to accommodate the different scenario workloads. For the worst case of 1000 phone variants and 10% of application uploads need RMs compiled, 6809 machines are needed. With this many machines, we would expect work to be offloaded by the phone manufactures. This would relieve the computational burden on the cloud compiler. Even so, 6808 machines is a reasonable number of machines to accommodate such a large amount of phone variants and applications that use mōbware.

2 Slots Requirement	% of April Apps that Use Hardware		
	0.1	1	10
	# of Apps Uploaded per Day		
	3.47	34.71	347.28
# of Phone Architectures	# of Machines Required to Compile RMs		
1	0.03	0.29	2.87
10	0.29	2.87	28.75
100	2.87	28.75	287.48
1000	28.75	287.48	2874.78

Table 6.12: Compilation components for dyamic PR and RM synthesis for 2 RPs.

6 Slots Requirement	% of April Apps that Use Hardware		
	0.1	1	10
	# of Apps Uploaded per Day		
	3.47	34.71	347.28
# of Phone Architectures	# of Machines Required to Compile RMs		
1	0.07	0.68	6.81
10	0.68	6.81	68.08
100	6.81	68.08	680.83
1000	68.08	680.83	6808.31

Table 6.13: Compilation components for dyamic PR and RM synthesis for 6 RPs.

Chapter 7

Future Work

All experiments previously were solely based on an FFT hardware module. To gain better insight on how large and how many RPs should be allocated onto the FPGA, we must run experiment 1 with different hardware modules, such as, AES, FIR filter, and QAM hardware modules. By studying their resource demands, we can better layout RPs and accommodate a wide variety of hardware modules. Similarly, we would like to add inter-module communication to reduce the burden of fixed RP numbers. For example, imagine that an AES hardware module is too large to fit into any of the allocated RPs. The AES hardware module can be split into two modules and connected using inter-module communication.

Also, we previously discussed routing as the most expensive activity during compilation of an RM. We also discussed that routing time varied based on RP location and RM size. By experimenting with RP placement and testing compilation time for the hardware modules described above, we can determine the optimal way to place RPs to reduce compilation time. We must also learn how routing resources are allocated and how we can improve floor planning to reduce routing resource contention.

In addition, we would like to explore altering the deployment model. Currently, we are expecting the phone manufacturer to provide the static design and the number of RPs that will be allocated onto the FPGA. It is very likely that the phone manufacturer would not be concerned

with what the static design looks like. As long as the phone has access to the resources of the PL and other I/O, they are content. This would allow us to define the static design, but provide different variants depending on the needs of the phone manufacturer. By providing standardized static designs and module allocations, we can significantly reduce the number of phone variants. With less phone variants, the time for compilation reduces and so does the number of required machines.

Chapter 8

Conclusion

We have introduce a cloud-based RTR deployment model for mōbware, giving applications access to these hardware modules. We leverage this deployment model to brute-force compile all RMs using Xilinx's partial reconfiguration technology. Also, by using a cloud deployment model, we can compile the RMs of the developers onto the static design of the phone manufacturer and deploy it to users. We placed as many FFTs onto the FPGA to determine how large and how many RPs should be placed onto the FPGA. We also built a cloud system environment and measured execution times of compilation of RMs when uploaded to a cloud. Using these measured execution times, we determined the throughput of our cloud deployment system and used application upload figures to determine the machines required to accommodate our deployment system.

Bibliography

[1] P. Possa, D. Schaillie, and C. Valderrama, "Fpga-based hardware acceleration: A cpu/accelerator interface exploration," in IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2011.

[2] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in Symposium on Application Specific Processors (SASP), 2008.

[3] C. Cullinan, C. Wayant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," MathWorks. 2013.

[4] J. Nunez-yanez and A. Beldachi, "Run-time power and performance scaling with CPU-FPGA hybrids," pp. 55–60, 2014.

[5] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing," in Proc. International Conference on Field Programmable Logic and Applications (FPL), 2007.

[6] C. Conger, R. Hymel, M. Rewak, A. George, and H. Lam, "Fpga design framework for dynamic partial reconfiguration," in Proceedings of Reconfigurable Architectures Workshop (RAW), 2008.

[7] S.Guccione, D. Levi, and P. Sundararajan, "Jbits: Java-based interface for reconfigurable computing," in Proc. Conf. on Military and Aerospace Application of Programmable Devices and Technology, 1999.

[8] E. Keller, "Jroute: A run-time routing api for fpga hardware," in IPDPS Workshops, ser. Lecture Notes in Computer Science, vol. 1800, 2000.

[9] T. Frangieh, R. Stroop, P. Athanas, and T. Cervero, "A modular based assembly framework for autonmous reconfigurable systems," in Reconfigurable Computing: Architectures, Tools and Applications, ser. Lecture Notes in Computer Science, 2012.

[10] R. K. Soni, N. Steiner, and M. French, "Open source bitstream generation," in Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013.

[11] D.Koch, C. Beckhoff, and J. Teich, "Recobus-builder a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas," in Proc. Field Programmable Logic and Applications (FPL), 2008.

[12] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda, "The erlangen slot machine: A dynamically reconfigurable fpga-based computer," in VLSI Signal Processing Systems, 2007.

[13] C. Patterson, P. Athanan, M. Shelburne, J. Bowen, J. Suris, T. Dunham, and J. Rice, "Slotless module-based reconfiguration of embedded fpgas.," in ACM Trans. Embedd. Comput. Syst, October 2006.

[14] T. Frangieh, R. Stroop, P. Athanas, and T. Cervero, "A modular-based assembly framework for autonomous reconfigurable systems," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7199 LNCS, pp. 314–319, 2012.

[15] S. Singh and P. James-Roxby, "Lava and JBits: From HDL to Bitstream in Seconds," The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), 2001.

[16] "Achieve power-efficient acceleration with opencl on altera fpgas."

[17] M. Klein, "Power consumption at 40 and 45 nm," Xilinx. 2009.

[18] S. Liu, R. Pittman, and A. Forin, "Energy reduction with run-time partial reconfiguration," Fpga, no. September, 2010.

[19] D. A. Patterson and J. L. Hennessy, "Computer organization and design," 2009.

[20] E. Horta, J. Lockwood, and D. Parlour, "Dynamic hardware plugins in an fpga with partial run-time reconfiguration," in Proceedings of the 39th conference on Design automation, June 2002.

[21] "Vivado high-level synthesis."

[22] "Zynq-7000 all programmable soc."

[23] E. L. Horta, J. W. Lockwood, and S. Louis, "PARBIT : A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)," 2001.

[24] "Xilinx partial reconfiguration."

[25] "Axi reference guide."

[26] "Vivado design suite."

[27] "Zed board."

[28] "Vivado high-level synthesis user guide."

[29] "Partial reconfiguration user guide."

[30] "Enable software programmable digital pre-distortion in cellular radio infrastructure."

[31] S. Kestur, J. D. Davis, and O. Williams, "BLAS Comparison on FPGA,CPU and GPU,"

[32] E. L. Horta and J. W. Lockwood, "Automated method to generate bitstream intellectual property cores for virtex fpgas," in Proc. International Conference on Field Programmable Logic and Applications (FPL), 2004.

[33] Altera, “An 531: Reducing power with hardware accelerators,” 2008.