Apps with Hardware: Enabling Run-time Architectural Customization in Smart Phones

Michael Coughlin, Ali Ismail, and Eric Keller, *University of Colorado, Boulder*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/coughlin

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Apps with Hardware: Enabling Run-time Architectural Customization in Smart Phones

Michael Coughlin, Ali Ismail, Eric Keller
University of Colorado, Boulder

Abstract

In this paper we present a novel system which incorporates programmable hardware (an FPGA) into a smart phone to enable a vision where apps can include both software and hardware components, or apps with hardware. We introduce a novel mechanism to enable sharing the FPGA in a practical manner by leveraging the unique deployment model of mobile applications - namely that deployment is via an app store, where we introduce a new cloud-based compilation. We present our prototype smart phone using the Zedboard, which pairs a Xilinx Zynq FPGA with an embedded Cortex A9, running an Android-based system which we extended to provide run-time system support for dynamically managing apps with hardware and providing a secure loading system. With this prototype, our evaluation demonstrates the performance gains for an AES encryption module (representing cryptography), a QAM modulation module (representing software-defined radio) of 3x to several orders of magnitude, with room for improvement and a hardware-based memory scanner (representing custom co-processors). We demonstrate the feasibility of our cloud-based compilation within the context of real app store statistics. Finally, we present a case study of a complete integration of hardware into an existing application (the Orbot Tor client).

1 Introduction

In designing new smart phone devices, the vendor must operate under a number of constraints – form factor, functionality, cost, energy use, etc. This leads to the vendor making a number of decisions regarding the various tradeoffs. These decisions, however, can then lead to the case where the device has both too little (the application developers/users want more) and too much (the application developers/users don't use what is there). *What if there was a way to put these trade-offs into the hands of users and application developers?*

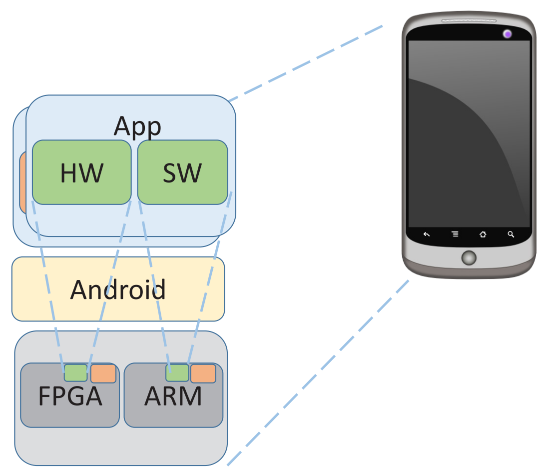

Figure 1: Smart phone with a processor (ARM) coupled with programmable hardware (FPGA).

In this paper, we present our 'apps with hardware' vision (illustrated in Figure 1), design, and implementation which incorporates programmable hardware, such as an FPGA (field programmable gate array), into a smart phone[1], and extends a mobile operating system to allow for application control of the current hardware configuration (*e.g.,* by including the hardware configuration with their app). The high-level idea is to couple software-like (re)programmability with hardware-like performance. In providing programmability, the phone vendor empowers the application developers (and by extension the end users) with the ability to influence the design decisions.

Developers, for example, would be able to introduce

[1] We envision this as being commercially available smart phones, not just in prototyping devices – a vision supported by the commercial availability of system-on-chip devices which already couple an ARM processor that is widely used in smart phones (such as the ARM Cortex A9 processor found in the iPhone 4) with reconfigurable logic [5, 25], with or more recently the ARM Cortex A53 [26], and further supported by recent advances by vendors where hardware modules can be designed using a high-level language, such as C++ [23].

(and deploy) new communication technologies, such as those that work on the emerging dynamic spectrum access paradigm [30], where they can perform ‘software’ radio at the needed hardware performance levels and gain system wide benefits (*e.g.,* from not needing phones to include many dedicated radio interfaces). Developers will also be able to introduce new accelerators, such as for cryptography or other parallel processing that improve overall performance and efficiency. Finally, developers will be able to introduce independent co-processors, which can, for example, provide additional security [57] capabilities not possible in today’s smart phones. In general, we introduce programmability of the smart phone hardware by creating an architecture centered on an FPGA with an embedded processor – with which, as we’ve seen with other programmable technology, such as graphical processor units (GPUs) and FPGAs in other contexts within the network systems community, developers will find creative ways to use the available processing power [52, 59, 45, 41].

Previous research has proposed adding reconfigurability to mobile devices, such as [11] [61] [32], but they have limitations that prevent them from being used today, such as lacking a method to share FPGA hardware, a distribution system for applications, or integration into modern operating systems or devices. Therefore, there are a number of challenges that we need to address to make our vision possible. First, we need to be able to share an FPGA between different smart phone applications – existing FPGA hardware and software are heavily centered on running a single application and not on idea of temporal or spacial sharing of resources. Second, we need a way to distribute apps with hardware to smart phones with compatible hardware – there is no binary compatibility in FPGAs or operating system which abstracts resources. Finally, we need the ability to manage the FPGA so that applications only have access to authorized resources – while processors have been adapted overtime to isolate running tasks, FPGAs have not.

In this paper we present our Cloud RTR system that addresses these challenges. In doing so, we introduce a system-level contribution that makes use of cloud technologies and builds on existing FPGA technology that together solve a problem that has eluded researchers for years. Specifically, we make the following contributions:

A slot based solution that allows for practical FPGA sharing: A central need to be able to allow apps to span software and the FPGA hardware is to enable the FPGA to be shared, as apps will be concurrently running. Our approach is based on Run-time reconfiguration (RTR), or the ability to change an FPGA’s configuration at run-time. Specifically, our Cloud RTR system builds on the idea of “slots” [38] [49] [42], or areas of the FPGA that can be reconfigured separately and shared between applications. To make this practical, where previous systems have failed, we provide a new approach to slot-based reconfiguration using a compilation system that abstracts away the underlying FPGA requirements. The resulting platform supports the use of slots at run-time, whereas previous systems only support slots at design time, and can share the FPGA between multiple parties, as we discuss next. Further, we introduce operating system services to manage slots at run-time to allow for on demand access from apps.

An app store based approach that allows for multiple parties to distribute apps: Without operating system and binary compatibility, envisioning a system which allows for multiple parties to create apps and have them be distributed to a wide variety of devices may seem difficult. We introduce a new app-store system which extends existing app stores to to allow for both the compilation and the distribution of apps with hardware. Developers can upload apps with hardware to an extended app store, which will interface with the compilation system in order to generate the required slot configurations. We extend the app store system further to ensure that these configurations are distributed to the correct devices in packaged apps, and we provide corresponding operating system support in order to install them.

A security manager that enforces access control to sensitive resources Our Cloud RTR architecture also provides a secure loading subsystem that ensures that only trusted applications have access to sensitive resources. This subsystem interfaces with the slot run-time management system to only allow for signed app hardware to access these resources. This system also uses common hardware security technology to ensure that applications cannot exploit the operating system to override these security restrictions.

In addition to the above system-level advances which enable the apps with hardware vision, we make the following contributions which evaluate and demonstrate their use:

Evaluation of the computational requirements of Cloud RTR: While our approach of performing some compilation in the cloud is, to some degree, simplistic, the fact that it has not been done before does, we feel, point to its novelty. Importantly, we go beyond simply proposing to compile in the cloud and extend our work to fully evaluate the computation requirements of such an app store to support this using data about the current app market ecosystem. We show that for compilation throughput per machine ranges from 51 to 121 apps per day, which translates to needing 1020 servers to support an app ecosystem where 1 percent of all apps use the reconfigurable logic for a case where there is 1000 phone variants.

Demonstration and evaluation of three applications:

In Section 2, we describe three example categories of applications that will benefit from an apps with hardware. For each, we implemented and evaluated a representative application (Section 6). Our evaluation of an app which offloads to a hardware based QAM module (a representative software-defined radio application) shows a 40x speedup and a hardware based AES module (a representative cryptography application) shows a 3x speedup (including all of the interface between hardware and software). Additionally, our evaluation of a simple memory security scanner (a representative architectural enhancement) that is capable of searching the entire system address space only results in 3% overhead for other software running. Finally, to understand the considerations when integrating into existing, complex, applications, we modified the open source and widely used Orbot [17] Tor [37] client for Android to include and use a hardware cryptography module (Section 7).

In the remainder of the paper we will further motivate the proposal to incorporate an FPGA into a smart phone (Section 2), describe past FPGA sharing attempts (Section 3), and describe our system, including our Cloud RTR architecture, which includes both the architecture for our slot compilation and app store extensions (Section 4), our runtime management and secure loading architecture (Section 5. We then provide an evaluation (Section 6), describe our case study of the Orbot Tor client (Section 7), describe other related work (Section 8), and then conclude (Section 9). We are also providing the code for the entire implementation (compilation, applications, FPGA system, and Android enhancements) in a git repository: `https://github.com/nsr-colorado/cloud-rtr`.

2 Motivation (Why an FPGA)

The premise of incorporating an FPGA into a smart phone lies in the general benefits of an FPGA – that it provides hardware-level programmability which will enable phone manufacturers to defer some decisions about tradeoffs to the end user and enable developers with the ability to innovate in the hardware space.

Here we discuss a few examples that help motivate an FPGA within a smart phone, including a description of a demonstration application that we implemented for each of these categories.

2.1 Architecture enhancements

For our first set of motivating examples, we present several architectural enhancements that have been proposed in the research community that each required a hardware plug-in and were targeted at a server. With our work, similar benefits could be brought to a smart phone. It is important to note that FPGAs are not limited to streaming and highly parallel processing (though they do excel at that). These types of applications can implement security functions, which are supported by our secure loading technology (discussed in Section 5).

CoPilot: CoPilot [57] is a PCI card designed to detect rootkits. As rootkits execute at the highest privilege, detection mechanisms at the same (or lower) privilege are presented with a significant challenge. The CoPilot PCI card is independent of the processor and operating system and has access to all memory via the PCI bus. Rootkit detection (or more generally, security applications) have tremendous potential with the introduction of an FPGA within a smart phone.

Somniloquy: The Somniloquy [27] work observed that the energy consumption on servers was impacted by a number of low-rate type of tasks that prevented the servers from entering the power saving states. As such, they proposed a small, low-power processor that could perform these tasks, and if needed, trigger the main processor to exit a low-power state. In the case of a smart phone with an FPGA, similar types of activity has been observed in smart phones [31], so a small co-processor in the FPGA fabric could provide a solution (while also enabling the main processor to shut off completely). We leave full exploration of power as future work.

As a demonstration of architectural enhancements, we have implemented a memory scanner module, as a simplified proxy for a CoPilot-like function, that scans our device's system address space.

2.2 Software-defined Radio

A great deal of research has resulted in many innovations in wireless communications which allow wireless interfaces to have better performance or more functionality. Research papers in this space commonly use FPGA platforms (such as the WARP Board [28, 53]), or devices to interface to high performance desktop machines (such as the USRP [22]) in order to meet the needs of the new innovation. While these papers provided promising research results, there is little opportunity for deployment – requiring the researchers to commercialize the technology, or get adoption from a major chip vendor.

With a smart phone that has an FPGA along with a more flexible radio front end (*e.g.,* a tunable antenna), developers of a new communication protocol could simply create an app, enhancing the impact of the research. This architecture also has benefits for production systems, as existing devices could be upgraded to new wireless systems without requiring replacement, such as upgrading such a system from 3G to 4G wireless technology.

As a demonstration of an SDR application, we have included an example implementation of a Carrier Phase

Recovery Loop for a single carrier Quadrature Amplitude Modulation (QAM) demodulator. QAM is a representative building block in signal processing applications including many real-world modulation systems.

2.3 Cryptographic and Parallel Processing

FPGAs have the ability to perform large amounts of processing in parallel. This allows them to achieve higher throughputs and lower latencies.

An exemplary application for FPGA acceleration on a smart phone is cryptographic processing, as it both faster in an FPGA and widely used – including the encryption of internet communication using SSL and communication protocols such as Tor [37], the accountable internet protocols [29], and Named-data Networking [44] For example, an FPGA (Altera Stratix V) was shown to be 520 times faster than a general purpose processor (Intel Xeon E5503) for AES encryption (and 15x speedup over an AMD Ratheon HD 7970 GPU) [4]. While the exact numbers will depend on a number of factors, this is illustrative of the potential.

Parallel processing goes beyond cryptography. One recent example used an FPGA based server [10] to implement common functions used in analytics (search, fuzzy search, and term frequency), and in each case demonstrated that it would require 100-200 servers running Spark [65] to match the performance. This example is really geared towards cloud scale applications, but we believe this would allow us to perform some analytic processing locally (on the phone) without needing to send private data to some cloud based backend.

As an example of this type of application, we have implemented a 128-bit AES encryption module that can encrypt an arbitrary number of 128-bit contiguous regions of memory. We also incorporated this AES module into the Orbot Tor client (Section 7).

3 Past Attempts

A central challenge in reaching our vision relates to how to share the FPGA between applications and the system. That is, we wish for multiple apps to be able to simultaneously use some of the FPGA's programmable fabric, while at the same time allowing the operating system to use some of the programmable fabric as well (*e.g.,* to connect to some I/O devices).

The core concept required is run-time reconfiguration, or the ability to dynamically change the FPGA's configuration (completely or partially) at run-time while it is still operating. Despite over a decade of research in run-time reconfiguration [33, 39, 48, 34, 36, 51, 62, 43] there has yet to be a practical solution which would enable hardware modules from various sources to be loaded onto a variety of platforms.

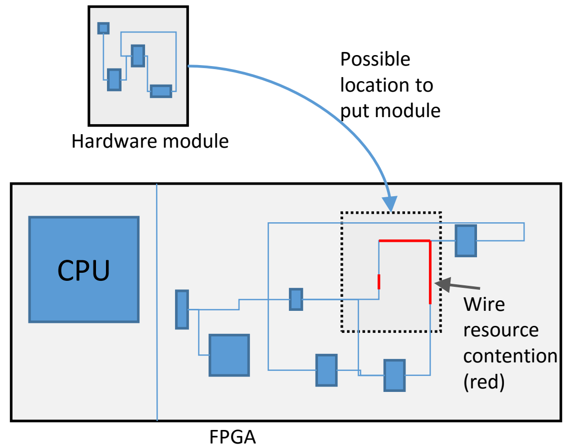

Figure 2: Example of partial reconfiguration in a running FPGA configuration.

3.1 Why is sharing an FPGA difficult?

The main challenge in achieving FPGA sharing is ensuring that the apps' modules in the FPGA do not conflict with each other, or with other logic that is present in the FPGA. FPGAs are difficult to share because a complex mapping of resources must occur in order to generate a configuration for an FPGA. This is because application's logic must be mapped to physical resources in the FPGA, and connections must be made between these locations, just as in any physical circuit.

As an example, consider Figure 2, which illustrates a single module to be loaded into an FPGA at run-time. The dotted area indicates one possible location to put that module. As indicated, however, there will be contention for resources – *i.e.,* this module cannot co-exist with the current FPGA configuration. Because of this, the partial reconfiguration mechanism supported by the vendors (Altera and Xilinx) comes with great restrictions – the modules can only work with a single design (in our case, they wouldn't work across phone architectures), and they can only be loaded into a single location. These restrictions make partial reconfiguration unusable in its current form to enable apps with hardware.

More general run-time reconfiguration approaches have been proposed in the research community that fall into one of two categories, which we describe next. In general neither of these approaches are practical.

3.2 Soln. 1: Run-time Place and Route

The first approach is to perform place and route at run time [60] [46] [56] (rather than when it is normally performed – at design time). As background, place and route is a computationally expensive task of first mapping logic elements from the design (place) and then determining a collection of wire resources to use to connect the logic elements from the design (route).

This approach enables reconfigurable modules to be created entirely separate from the FPGA configuration. They can be loaded into the FPGA by being placed around existing hardware and connected with free wiring resources.

This is a general approach and supports our model, but there are two major problems. First, place and route can take a long time, depending on both the size of the reconfigurable module as well as the sparseness of the current FPGA configuration – *i.e.,* if there are few resources available, it will be a more difficult task to find a solution. The implication relates to the second problem – that a solution is not always possible, which means that the app would fail to load.

3.3 Soln. 2: Slot-based Reconfiguration

The second method that has been proposed also seeks to support a general approach where the static design and the reconfigurable modules can be created independently. This approach does so by reserving empty and identical areas in the static design [38] [49] [42]. These areas, or slots, are analogous to PCI slots on a motherboard, where any card can be plugged in independent of the processor. In this case, the 'cards' are partial bitstreams (a binary file used to configure an FPGA). Two constraints emerge:

Partial bitstreams need to be relocatable – So that a partial bitstream can be loaded into any slot, each area needs to be identical. This is not difficult from a logic standpoint as FPGAs are fairly regular structures. In order for the static and reconfigurable portions to be able to communicate, however, there need to be wires that cross the boundaries which, in turn, need to be identical for each slot. This puts incredible strain on the creation of the static design, to the point of not being practical (because place and route becomes very constrained if certain circuit elements need to use certain physical wires).

Partial bitstreams cannot conflict with the static design – That is, when loading a partial bistream, it cannot, for example, use a wire, that the main system design used (and vice versa). To achieve this, the static design is highly constrained to reserve areas such that no logic is used (generally, easy to achieve) and such that no wires are used (in Figure 2, this would mean that static portion of the design would not have been allowed the wires that are in the dotted area). Such constraints are ultimately possible (through a painstaking process of reverse engineering and over-constraining), but highly constrains the static portion of the design – forcing wires to be routed around these areas, causing them to be extra long and resulting in congested areas.

In short, this is a good abstraction, but not practical.

4 Cloud RTR: A Practical Approach For Sharing the FPGA

In order to realize the apps with hardware vision, we need two things. First, we need a mechanism to be able to share the FPGA resources – *i.e.,* a practical run-time reconfiguration mechanism that overcomes the limitations of past solutions in terms of usability and deployability. Second, we need a mechanism to be able to manage the apps at run-time. Here, we describe our novel solution for enabling FPGA sharing, and in Section 5 we describe our system support for run-time management of apps.

4.1 High-level Overview

Central to our design, we adopt the general idea of slots – that is, reserved areas within the FPGA where modules can be loaded. As previously mentioned, we believe this is a good abstraction, but the previous realizations of it are not practical. The key difference with our approach is that our slots are less constrained – only logic resources need to be left free (which is easier), but the wiring resources within these areas can be used by the static design logic (*i.e.,* the portion of the FPGA configuration that does not change and provides system functionality for different phones). Other key differences with our approach are that the reconfigurable modules can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

The key idea to enable this is that by leveraging the delivery model of mobile apps (*i.e.,* via an app store), we can effectively merge the modules into various static designs in the cloud, before delivery to the end user. We call this Cloud RTR (RTR for run-time reconfiguration). As illustrated in Figure 3, each phone manufacturer and app developer would submit their design to the Cloud RTR system, and the Cloud RTR system would perform a compilation step to enable a general run-time reconfiguration mechanism.

In this section we describe the architecture of the phone in order to support this model (Section 4.2), how the apps are designed to work within the framework (Section 4.3), and finally discuss how Cloud RTR performs the compilation (Section 4.4).

4.2 Static (Phone) Design Architecture

The key requirement for the phone's design lies in the ability to support interfacing the reconfigurable modules

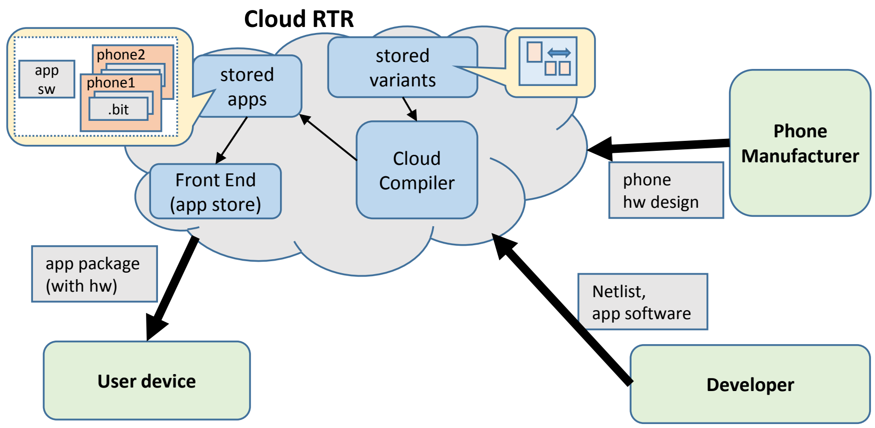

Figure 3: Cloud RTR approach to the generation and deployment of apps with hardware

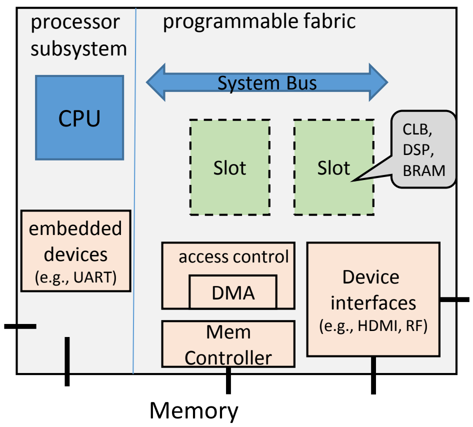

Figure 4: Example static FPGA design.

with the rest of the system resources. Illustrated in Figure 4, and described below, are the main components.

Slots

Slots should have enough of all types of resources to be useful. Today's FPGAs can contain (i) configurable logic blocks (CLBs), which can implement any logic function of N inputs, (ii) block random access memory (BRAM), which are small memory elements (*e.g.,* 36 Kb in the FPGA we use for implementation), and (iii) digital signal processing (DSP) blocks, which are custom building blocks geared toward signal processing applications.

Slots will also need to be able to access various system resources and expose an interface for communication with the processor. For this, we expect that all slots will allow access to (i) a system bus for communication with the processor, and (ii) a direct memory access (DMA) controller for access to system memory.

Module-to-Memory Interface

In order to provide performance benefits, the modules need to be able to directly access CPU-accessible system memory. A DMA controller that is accessible by the hardware modules would allow for modules to access system memory without involving the processor (providing the greatest performance and flexibility). To achieve this, we also need a security module which performs access control – that is, one which limits what memory each hardware module can access and is configured by the operating system.

Processor-to-Module Interface

The ability to stream from memory will be important, but the processor also needs to be able to directly interface to each module. This interfacing is achieved through the use of, for example, a system bus (such as the ARM-based Advanced eXtensible Interface, or AXI).

Device interfacing and other misc. logic

The rest of the static design will include interfacing to the various devices that will connect to the FPGA. Some devices, such as a UART, may have interface logic included in the processor sub-system, but the rest, such as interfacing to a tunable antenna, may go through the programmable fabric with custom logic to interface with it. These devices will be connected to the general interface of the slots, allowing for manufacturers to include custom peripherals without requiring new slot definitions.

4.3 Reconfigurable (App) Module Architecture

In the previous slot-based approaches, the reconfigurable modules are designed for a specific slot design (device, interface, etc.). In our approach, we abstract away the ultimate target such that app developers can develop reconfigurable modules that can be loaded onto a variety of platforms. Of note, the reconfigurable modules in our approach can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

Here we describe the design of an app, with the various components illustrated in Figure 5.

App Hardware

The first major component is the **app hardware**. In order to match the skills of app developers, we focus on the high-level synthesis (HLS) design flow [23] that has emerged in recent years which allows developers to use a high-level language (*e.g.,* C) to describe hardware modules[2]. *What this means is that the argument that FPGAs are hard to design for, and therefore not accessible to the software app developers, is quickly becoming invalid.*

The app hardware (in this example) is written as a C++ function, *example()*. The parameters to the function describe the interfaces to the rest of the system, such as char arrays (*e.g.,* var1), which describe memory mapped registers accessible to the processor or streaming memory interfaces (*e.g.,* var2, which has the type hls::stream), that allow for streaming data from memory (when connected to DMA hardware in the static design). This description is valid C++ code that can be compiled and tested as software which can simplify hardware testing.

While there will need to be some consideration by developers, in general developers will not need to be fully aware of the hardware architecture. For example, the exact bus signals for communicating with the module are not directly used, but are instead inferred based on the types on the function parameters (such as how to perform data transmission handshakes or send valid signals). With this, the same module could actually target various hardware interfaces (*e.g.,* if different handshake protocols or signals are used, or if different bus widths are available). Developers do need to consider the size (resource utilization) of their hardware modules to ensure they will fit in a particular slot size. We envision standard slot sizes will emerge (much like screen sizes), and in our design flow we allow for modules designed for one slot size to always be instantiated in a bigger slot.

App Software

The **app software** that the developer writes will be mostly the same as current apps (*e.g.,* written in Java for Android apps). The only difference is the loading of and interfacing with the hardware module. To load, the app will submit a request to a system service to load the bitstream (*e.g.,* via an intent in Android).

To interface with the module, the app will use the functions in the user-level driver generated by the FPGA vendor's high-level synthesis tool when synthesizing the design (the process which generates the FPGA hardware from the C++ code). This driver is low-level code that runs within the same process as the application and provides functions that can be used to interface with the reconfigurable module. Functionality includes mapping memory regions (*e.g.,* via *mmap()*) that both the reconfigurable module and the app will access. It also provides functions to access the various registers (the char[] variable) through functions like *set_var1()*.

[2] The developer can use a hardware description language, but will then need to manually provide the interfacing hardware and software, which are automatically created with high-level synthesis.

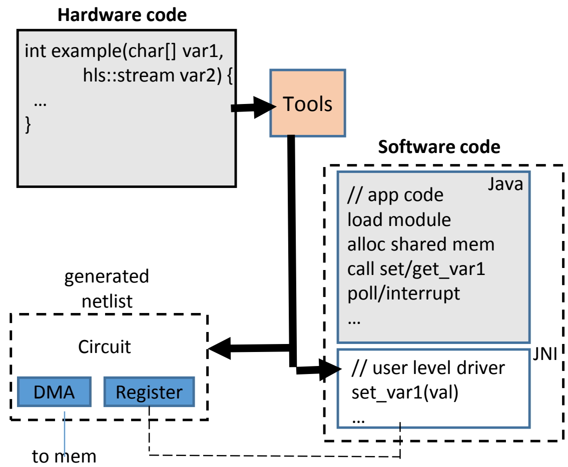

Figure 5: Example app design.

4.4 Cloud Compiler (in the App Store)

The Cloud RTR compiler is responsible for ensuring that an app's hardware module(s) can be loaded into a variety of target devices (smart phones). Rather than working around the limitations of the vendor tools, we work within their constraints, resulting in a practical solution. Recall that the vendor tools have a partial reconfiguration design flow which has the constraints that a module can only be used for a specific static design and target FPGA and for a specific location within that static design. Working within that, the Cloud RTR compiler will simply use the vendor tools to compile the module for every static design variant and for every possible slot within each variant.

The end result is a data structure stored within the app store that looks like the following (where a.bit...e.bit are individual partial bitstreams):

```
[phone 1:
  [slot1:a.bit, slot2:b.bit, slot3:c.bit]]
[phone 2:
  [slot1:d.bit, slot2:e.bit]]
```

When an app is downloaded to a given device, the Cloud RTR system will repackage the application with the set of device-specific bitstreams (possible since the app store has knowledge of a user's device). In Android, for example, apps are packaged in an Android Application Package (APK), which will now include module bitstreams as extra resources for apps that use hardware. To get a rough idea of how this impacts the size of an APK, for the case study we describe in Section 7, the hardware module bitstream is 904KB, the Orbot APK of the version we modified is 5.5MB (before any added hardware), and the latest Orbot release is 11MB.

We show that this brute-force approach is quite practical in Section 6. As such, it provides a general approach that is deployable and usable today. In addition, we also envision a large amount of reuse of both static designs and hardware modules (*e.g.,* by using precompiled libraries). Just as SoCs are oftentimes reused between different mobile devices, there is no need to have a distinct static design for different devices unless a particular device requires some custom technology.

5 Run-time Management of Apps

With the ability to share the FPGA, as provided by Cloud RTR, we now discuss how our system manages the apps' hardware within the Android operating system. This has two aspects – (i) how to dynamically manage the loading and unloading of hardware modules for various apps, and (ii) how to ensure the modules cannot compromise the running operating system and vice versa. We achieve the dynamic management by modifying the Android operating system to include a system service that manages the loading and unloading of modules and enables application software to access these modules. Our secure loading system takes advantage of some hardware security features of modern FPGAs and some hardware in the static design in order to provide the needed security.

5.1 Dynamic Module Loading Service

To support apps with hardware, there needs to be system support for loading hardware modules into the FPGA. The operating system will have access to our secure loading system (described in the next section) that can take a hardware module compiled using the Cloud RTR system and load it into the FPGA. However, user applications will not have direct access to this system.

User applications will instead submit requests through a privileged *hardware loader* system service. Upon loading and initialization of the app, the service will be provided with the location of the app's hardware module files. The service will then choose an empty slot, select the module compiled for this slot, and use the secure loading module to load the module into the FPGA. In the case where no slots are available, the operating system can create 'virtual' slots by time-slicing existing slots. Given the slot reconfiguration time, we do not expect to swap app hardware as frequently as app software, but we see this is an area for future consideration.

We can implement virtual slots by using the readback capability of FPGAs to store the running configuration of modules, and developers can provide custom unload functionality to aid the readback system in storing difficult to access state (specifically, certain FPGA memory is more difficult to access). Applications that would be disrupted by time slicing can be specifically flagged as unsafe to swap, but the number of these applications running simultaneously should be restricted.

The time to load a hardware module provides an estimate of the time needed to context switch a hardware module. This time is a function of the size of the hardware being written to the FPGA, which we measured to have an average throughput of 37 MiB/s. This leads to a latency of approximately 100 ms for a 4 MB static bitstream, or 27 ms for 1 MB hardware module.

The hardware module is presented as a devfs character device in the Linux /dev directory (when using Android). The hardware loading service will set file permissions to ensure that only the application that requested the loading of the hardware module can access it.

5.2 Secure Loading

We introduce a secure loading mechanism to provide support for protecting both the operating system from app hardware and the FPGA configuration from the operating system (*e.g.,* a rootkit). Our secure loading mechanism is an extension of secure boot technology where (i) we disable the processor's connection to the configuration ports of the FPGA, and (ii) we add a module within the static design to support loading of app hardware. For space purposes we can only sketch the high-level overview of the secure loading.

Threat model and assumptions: We assume that any code running on the CPU, including the operating system, can be malicious. We also assume that even with the support of modules such as the trusted platform module (TPM) to protect the booting of the operating system, malicious code can be executed at run-time that can compromise the operating system. We assume that some reconfigurable modules will be untrusted and potentially malicious. We assume that some reconfigurable modules will be trusted (from trusted sources, such as the phone manufacturer) and will not be malicious. Finally, we assume that the static FPGA configuration is trusted and correct at boot time (through secure boot mechanisms supported by modern FPGAs [64]). With this, we assume the keys of the app store and an additional trusted party are present at boot and cannot be modified.

The secure loading can be summarized as follows:

- Secure boot technology of the FPGA will load a trusted FPGA configuration as well as ensure the operating system is known to be good at boot.
- As part of the secure boot, the processor will have its access to the configuration ports disabled (*e.g.,*

the processor configuration access port, PCAP, and the internal configuration access port, ICAP).

- A hardware secure loading module, part of the static configuration, will accept requests to load a reconfigurable module from the operating system .
- Every module will be signed by the app store. From this the hardware secure loading module will verify the signature in order to ensure that the configuration refers to is the slot to be loaded (preventing the case where an app was modified to include a configuration which overwrites parts of the FPGA other than the allocated area).
- For trusted modules (which will signed by the app store and by an additional trusted party), the hardware secure loading module will verify the signature and if it succeeds, configure the memory access control to allow the module to have access to system memory. Trusted modules will not be able to be unloaded or overwritten without a reboot.
- For untrusted modules (which will be signed by the app store, but not by an additional trusted party), the hardware secure loading module will configure the memory access control to restrict access to memory.

With this process, we get the following properties:

- Modules will be correctly loaded only into the slots for which they are supposed to be loaded and not overwrite any static configuration.
- Code in the operating system cannot load a module in a slot which gets access to system memory (preventing app hardware from helping apps bypass system protections, or otherwise harm the system).
- Code in the operating system cannot overwrite or unload a trusted module (preventing rootkits, for example, from unloading security modules meant to detect the presence of rootkits).

6 Evaluation

There are two main questions to answer, which we discuss in this section:

Is there value in apps with hardware?

There's general acceptance that hardware will be faster than software[3]. The question we seek to answer here is whether the same performance benefits are retained when we consider it within a system (*e.g.,* does crossing the hw-sw boundary make things worse).

[3]That's not really the focus of our paper – we take the stance that there are places where each wins (FPGA, CPU, GPU) and that heterogeneous architectures are good, and more importantly open programmability is what drives innovation.

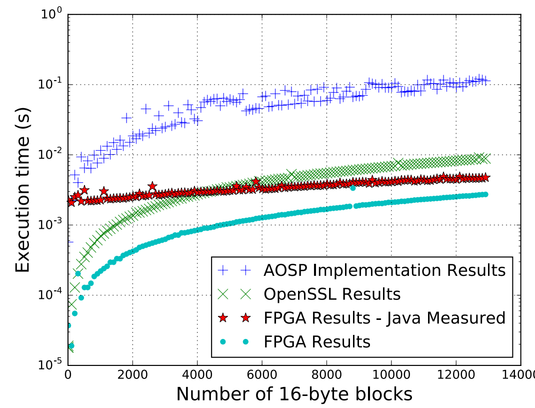

Figure 6: The execution time to perform an AES encryption for a range of data sizes – from 10 to 13000 contiguous 128-bit (16-byte) segments of memory.

Is the cloud compilation of Cloud RTR practical?

As mentioned, rather than continuing the path of runtime reconfiguration research, which leads to creative, but impractical solutions, we aimed for a solution which was highly practical and deployable today. This resulted in a brute force approach. Here, we ask whether this is itself practical by examining the processing required to support the app market ecosystem.

6.1 Application Performance Acceleration

Performance acceleration is one of the benefits of using an FPGA. Of course, we also believe that new applications are now enabled, such as our hardware-based memory scanner. We focus on performance here as a concrete demonstration with a quantitative evaluation.

We focus on three key application domains that are enabled. Each of these applications consists of a hardware module written in C++ using high-level synthesis and an Android application that interfaces with the module. We run the hardware module through the Cloud RTR compilation platform targeting the static design for our development board. The end result is an APK that can be loaded into our demonstration board. Each application fit within our slots, which we defined at 12% of the overall FPGA area – a number resulting from dividing the remaining area after what is needed for the static design by six available slots.

For our experiments, we protoyped a mobile device using the Zedboard development board, which integrates a Xilinx Zynq 7020 FPGA [25] that has an embedded dual-core ARM Cortex-A9 CPU. We based our Android services on the Android 2.3 and 5.0.2 operating systems that were already ported to our device.

6.1.1 Cryptography: AES

In Figure 6, we compare three different AES implementations using our development board, and running Android as the operating system on the CPU[4]:

- **FPGA (ours)** – an AES FPGA reconfigurable module accessed by an Android application in native code using the the Java native interface (JNI).
- **OpenSSL** – the OpenSSL AES implementation which we interfaced directly from a C application.
- **Android AES** – the AES implementation provided to Android applications by the AOSP.

This figure shows the execution times of the AES implementation (which we derived from [1], though we also experimented with versions from Apple [2] and NIST [3], which had identical performance) for a range of data sizes to be encrypted, varying in size from 10 to 13000 contiguous 128-bit segments of memory. It can be seen that the FPGA implementation is on average three times faster than the OpenSSL implementation, and is approximately 12 times faster than the AOSP. However, the execution time of the FPGA module as measured by Java (marked in red circles) and executed using the JNI is longer than the execution time of the same module when executed directly by a C program (marked in blue diamonds). This is likely due to overhead entailed in copying memory to the JNI function call and transferring control to the JNI. This can potentially be alleviated using Java direct byte buffers passed directly to the JNI function, but is deferred to future work.

6.1.2 Software-defined Radio: QAM

This application can process a signal stored in a contiguous memory region and produce an output signal that is stored into another contiguous region. In a live smart phone, the static design would place the signal off of the antenna into buffers in memory, notify the Android application of a buffer being full, and the application would pass this data to the QAM module. The number of samples the QAM block can process determines the sample rate of the radio application. We implemented this module by modifying (to be compatible with our Cloud RTR system) a reference Xilinx project [55], which comes with C++ code that can be executed in software or run through high-level synthesis to produce hardware.

As shown by Figure 7, the hardware implementation is several orders of magnitude faster than the software implementation. The hardware implementation achieves an average throughput of approximately 5 Msps (megasamples per second), while the software implementation only achieves and average of approximately 500 samples/s. The Xilinx application notes claim a throughput value of 50 Msps [55], which is likely achievable due to the fact that the hardware device is intended to process

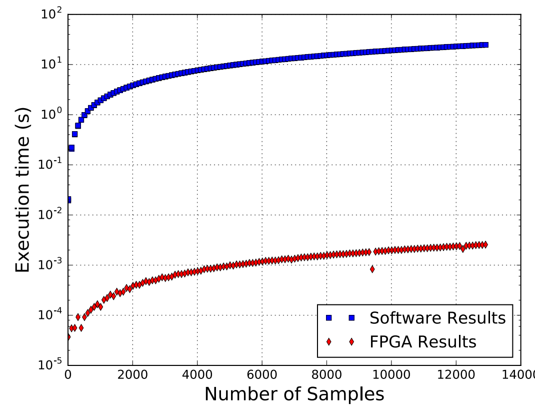

Figure 7: The execution time to process a different number of samples with our QAM application.

data received directly from an analog-to-digital converter (ADC), whereas our implementation has been retrofitted to stream data from system memory.

6.1.3 Memory Scanner

Our final application is a simple implementation of a hardware memory scanner that searches our device's address space for occurrences of a 16 byte string. We wrote custom C++ code that we ran through high-level synthesis for this implementation.

Using the LMbench testbench [50], we instituted a memory benchmark that measured the throughput of our device's memory while under normal operation and while the hardware memory scanner was executing (which is constantly reading from memory). Using this benchmark, we measured a 2.7% reduction in performance for read operations and a 5.5% reduction in performance for write operations.

6.2 Cloud Compilation Resources Needed

We propose performing compilation of the reconfigurable module in the cloud as part of the process to upload to the app store. To understand the feasibility of this, here we evaluate the amount of computing resources needed to sustain an 'apps with hardware' ecosystem.

The metric of interest is how long it takes to compile a single reconfigurable module for a given static design. Recall that a static design is the base design that roughly corresponds to a system on chip used for a given smart phone. These static designs have open areas (slots) for placing reconfigurable modules.

For all experiments, we used a server with an Intel Xenon CPU (2.1 GHz, 6 cores, 48 GB RAM).

6.2.1 Single App for a Single Static Design

An app with hardware uploaded to the app store must have its hardware modules compiled for each slot that it

[4] We also performed the OpenSSL benchmark in Ubuntu Linux to confirm that the Android OS does not institute a performance penalty.

# of slots	Compilation Time (min)	Throughput (apps/day)
2	11.92	121
3	14.93	96
4	19.02	76
5	24.21	59
6	28.23	51

Table 1: Compilation time and number of apps a single server could service per day.

6 Slots Requirement	% of Jan 2016 Apps that Use Hardware		
	0.1	1	10
	# of Apps Uploaded per Day		
	5	52	520
# of Static Designs	# of Machines Required to Compile RMs		
1	1	1	10
10	1	10	102
100	10	102	1020
1000	102	1020	10200

Table 2: Number of servers required to support the compilation requirements of Cloud RTR, assuming designs with six slots.

can be placed into. However, certain steps in the process do not need to be performed for each slot – *e.g.,* synthesis needs to be performed once for each module, then for each slot, the synthesized module needs to be placed and routed. For this evaluation we used an FFT module, which is a highly regular structure and enabled us to adjust its parameters to effectively alter its size to fill up any slot size we experimented with.

Table 1 shows the total time to compile a single application's reconfigurable module for static designs with two to six slots (each slot is defined as 12% of the overall area of our FPGA, as previously mentioned), as well as the extrapolated throughput (the number of apps that could be compiled per day on one server given this compilation time). We chose up to six slots (in contrast to the 60-100 slots in [38]) as we believe each to be big enough to implement a reasonable module within a single slot.

6.2.2 Compiling All Apps

Using the calculated throughput, we can now estimate the amount of computing resources needed to service the entire app ecosystem.

First, we need to know how many apps are uploaded each month. The company AppFigures provided us with the Google Play Store application upload figures for the entire year of 2014, with a total of 1.43 million apps at the end of 2014, and an average monthly growth of 6.10%. While we are unsure if this growth rate persisted over the past year, we use it to estimate the current needs. Using this average monthly app growth, for the month of January 2016, 177,521 applications are predicted to be uploaded into the Google Play Store.

Table 2 shows the number of machines required to service monthly demands for compiling apps with hardware, for six slots as an example (2-5 slots would be proportionally less). Each table varies the number of apps with hardware uploaded each day based on the percentage of applications that require hardware (0.1%, 1%, and 10%), as well as the number of static hardware variants – for the sake of illustration, we assume from one to 1000 variants, with each interval increased by a factor of ten (we expect the number of variants to be on the low end, as static bitstreams can be reused between devices, just as phones today use a small set of SoCs, and not every device will require a new static bitstream).

The cloud provider will easily be able to support the lower end of the spectrum internally. On the upper end, the cloud provider might look to relieve the computation burden by offloading to the phone manufacturers to compile for their own variants.

7 Case Study: Orbot Tor Client

Developing our demonstration applications from scratch allows us with to design it to use an FPGA natively. Here we explore modifying an existing, complex application to make use of a hardware module to understand the inefficiencies that may result.

We chose to modify the Orbot [17] Tor [37] client for Android. Tor is an anonymization network that allows for a user to access the internet without disclosing their source IP address, making identifying and tracking their internet traffic very difficult for third parties. A Tor client creates a circuit through this network and encrypts their traffic separately for each node along the path to prevent eavesdropping during transmission. Because of this extensive use of encryption and based on notes by the Orbot developers mentioning that AES is one of the areas to optimize Orbot [21], we see this as an ideal case study.

Our AES module implements the CTR (counter) mode of operation on top of a standard AES block cipher that is an equivalent to the OpenSSL CTR implementation used by Tor. In order to integrate our AES accelerator with Tor, we replaced all calls to OpenSSL AES encryption with calls to the FPGA accelerator, which proved to be a fairly minor modification. We also needed to ensure that all data that was to be encrypted was located in a contiguous memory region with a known physical address, which required us to replace all *malloc()* calls with calls to a custom memory allocator, and leverage a memory region that we reserved from the kernel.

With this, the application is able to make use of the FPGA resources and operate correctly. However, there are inefficiencies remaining due to (i) the overhead required to allocate memory in the reserved region, (ii) the overhead in accessing this memory, as it is implemented using memory-mapped I/O, and (iii) the fact that certain memory system calls (*e.g., malloc()*, *memcpy()* and *memset()*) are incompatible with the current memory mapped implementation – which would require more extensive modifications to the code to resolve. Even so, this provides us with great insight into how apps should be designed to capitalize on the FPGA resources and is an area for future improvements.

8 Related Work

Although there are no existing systems that implement all of the functionality of our Cloud RTR system in mobile devices, there has been much work done in reconfigurable computing in other contexts, including several different attempts with Android.

Of note from previous reconfigurable computing research is the BORPH system [62], which attempts to create operating system extensions in Linux for FPGA operations, and uses Berkeley's BEE2 system [35], and the more recent Connectal framework [47], which can automatically generate HW/SW interfaces during hardware development. These systems, however, do not address application distribution or FPGA resource sharing.

In terms of mobile systems research, some proposals have been made, such as the rSmart system [63] and the work from Smit et. al. [61]. Smit et. al. proposes a similar hardware architecture to the Zynq-7000 architecture, but does not present an operating system integration or a deployment system. The rSmart system only presents a high-level sketch of a system similar to ours, but no details on implementation or integration are provided. Our system builds upon this research to create a general system that is deployable using existing technology.

There has also been recent advantage in reconfigurable cloud platforms. For example, Microsoft's Project Catapult makes use of of FPGA peripherals in data centers to accelerate web searches [58] and neural networks [54], and Intel's acquisition of Altera [13] is leading to x86 CPU architectures coupled with FPGAs [40]. Microsoft's solutions, however, are only single-application hardware accelerators, whereas our system allows for usage in general applications. Intel's system is more general, but has not been released publicly, but does claim to use OpenCL [16] as the software interface.

Our work is complimentary. For example, OpenCL can be used on mobile devices with support from major hardware manufacturers, such as ARM, Intel and Qualcomm, [14] [20] [8] [12] [18], and can even be used with our system by using a compatible hardware module. OpenCL's main limitation is its focus on parallel acceleration, which does not enable new architectural enhancements, such as our SDR or security applications.

Reconfigurable Android devices and systems have also been proposed, such as Google's Project Ara [11], among others, including various other modular phones [19] [9] [15]. These modular phone systems allow for reconfiguration and upgrading of smart phone *physical* components, similar to how personal computer components can be upgraded. However, these modular architectures can only be reconfigured manually by the user replacing the physical modules, whereas our system allows for dynamic and custom reconfiguration by software.

Finally, the Android OS has been ported to the Zynq-7000 in several projects, such as the work of Barbareschi, et. al., among others [32] [7] [6] [24]. However, with the exception of the work of Barbareschi, et. al., these projects only port the OS to a new device. The work of Barbareschi, et. al. only extends this work to create an Android-compatible custom accelerator to address a single problem, whereas our system allows for any general software to create their own custom hardware modules.

9 Conclusions and Future Work

In this paper we presented the concept of 'apps with hardware' where, with the introduction of an FPGA into a smart phone we can enable app developers to innovate in the architectural (hardware) space, as they can today in the software space. Our Cloud RTR cloud based compilation mechanism overcomes past limitations of using the FPGA in a general way and does so without requiring any modifications from the vendors (making it deployable today). Our Android-based run-time application management system enables the dynamic management of the execution of apps (and their use of the available hardware), and provides a secure loading mechanism.

There is a great deal of possible future work. First, performing a thorough power analysis in a fair manner will provide great insight into both the benefits and needed system support, and building more apps to capitalize on the new capabilities would likewise provide great insight. Second, we wish to work with additional tools from other vendors (*e.g.,* Altera) and operating system platforms to explore implementation differences and with a partner to develop an actual prototype smart phone system (rather than the Zedboard) to further understand its viability. Finally, we wish to further investigate memory management techniques for better optimization.

10 Acknowledgments

This research was supported in part by NSF SaTC grant number 1406192. We would also like to thank Phil James-Roxby and Derek Woods for their guidance, and Xilinx for their hardware and software donations.

References

[1] http://programmablelogicinpractice.com/?p=87.

[2] http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-55010/Source/libtomcrypt/src/ciphers/ltc_aes/aes.c.

[3] http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[4] 40Gbit AES Encryption Using OpenCL and FPGAs. http://www.nallatech.com/40gbit-aes-encryption-using-opencl-and-fpgas.

[5] Altera socs. https://www.altera.com/products/soc/overview.html.

[6] Android 4.2.2 on zynq getting started guide. http://www.wiki.xilinx.com/Android+4.2.2+On+Zynq+Getting+Started+Guide.

[7] Android on zynq getting started guide. http://www.wiki.xilinx.com/Android+On+Zynq+Getting+Started+Guide.

[8] ARM Mali OpenCL SDK. http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/.

[9] Fairphone. https://www.fairphone.com/.

[10] FPGA System Smokes Spark on Streaming Analytics. www.datanami.com/2015/03/10/fpga-system-smokes-spark-on-streaming-analytics/.

[11] Google Project Ara. http://www.projectara.com/.

[12] GPGPU OpenCL API. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html.

[13] Intel Altera Acquisition. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

[14] Intel OpenCL SDK. https://software.intel.com/en-us/intel-opencl.

[15] LG G5. http://www.lg.com/us/mobile-phones/g5.

[16] OpenCL. https://www.khronos.org/opencl/.

[17] Orbot. https://guardianproject.info/apps/orbot.

[18] PowerVR SDK. https://community.imgtec.com/developers/powervr/.

[19] Puzzlephone. http://www.puzzlephone.com/.

[20] Qualcomm Adreno GPU SDK. https://developer.qualcomm.com/software/adreno-gpu-sdk/tools.

[21] Tor source code hacking documentation. https://gitweb.torproject.org/tor.git/tree/doc/HACKING.

[22] Universal Software Radio Peripheral (USRP) by Ettus Research. http://www.ettus.com/.

[23] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.

[24] Zedroid - android (5.0 and later) on zedboard. http://www.slideshare.net/noritsuna/zedroid-android-50-and-later-on-zedboard.

[25] Zynq-7000 all programmable soc. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[26] Zynq UltraScale+ MPSoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[27] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[28] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In *Proceedings of IEEE MSE*, 2007.

[29] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, 2008.

[30] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with Wi-Fi like connectivity. In *Proc. SIGCOMM*, Aug. 2009.

[31] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, 2009.

[32] M. Barbareschi, A. Mazzeo, and A. Vespoli. Network traffic analysis using android on a hybrid computing architecture. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing - Volume 8286*, ICA3PP 2013, pages 141–148, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[33] G. Brebner. Circlets: Circuits as applets. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998.

[34] G. J. Brebner. A virtual hardware operating system for the xilinx xc6200. In *Proc. International Workshop on Field-Programmable Logic (FPL)*, 1996.

[35] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Des. Test*, 22(2), Mar. 2005.

[36] O. Diessel and G. Wigley. Opportunities for operating systems research in reconfigurable computing. Technical Report ACRC99018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.

[37] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. USENIX Security Symposium*, 2004.

[38] 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.

[39] S. A. Guccione and D. Levi. XBI: A java-based interface to FPGA hardware. In *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 97–102, Nov. 1998.

[40] P. K. Gupta. Xeon+fpga platform for the data center. *The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, June 2015.

[41] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[42] 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.

[43] 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). Technical Report WUCS-01-13, Dept. Comput. Sci., Washington Univ., Saint Louis, MO, 2001.

[44] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proc. Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[45] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, Oakland, CA, May 2015. USENIX Association.

[46] E. Keller. Jroute: A run-time routing api for fpga hardware. In *IPDPS Workshops, ser. Lecture Notes in Computer Science*, volume 1800, 2000.

[47] M. King, J. Hicks, and J. Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 13–22, New York, NY, USA, 2015. ACM.

[48] E. Lechner and S. A. Guccione. The java environment for reconfigurable computing. In *Proc. International Workshop on Field-Programmable Logic and Applications*, Sept. 1997.

[49] 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.

[50] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[51] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2003.

[52] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. Netfpga: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2008.

[53] S. Neuendorffer and C. Epifanio. Generic partially reconfigured processor systems applied to software defined radio. In *Proc. of the Software Defined Radio Forum (SDR)*, 2007.

[54] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[55] A. Paek and D. Mackay. Implementing carrier phase recovery loop using vivado hls. `http://www.xilinx.com/support/documentation/application_notes/XAPP1173-carrier-loop.pdf`.

[56] C. Patterson, P. Athanas, 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.

[57] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security Symposium*, 2004.

[58] A. Putnam, A. Caulfield, E. 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 *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[59] T. Rinta-aho, M. Karlstedt, and M. P. Desai. The click2netfpga toolchain. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 77–88, Boston, MA, 2012. USENIX.

[60] 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.

[61] G. J. M. Smit, P. J. M. Havinga, L. T. Smit, P. M. Heysters, and M. A. J. Rosien. Dynamic reconfiguration in mobile systems. In *Proc. International Conference on Field-Programmable Logic and Applications (FPL)*, 2002.

[62] 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), Jan. 2008.

[63] N. Soundararajan. rSmart: The Reconfigurable (Real) Smartphone. *Provocative Ideas session of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013.

[64] S. Trimberger and J. Moore. Fpga security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, Aug 2014.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2012.