CRASH: Cognitive Radio Accelerated with Software and Hardware

A Thesis Presented

by

Jonathon Pendulum

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

April 2014
Abstract

The wireless spectrum has become more congested due to the rapid increase in the number of wireless devices. As this trend continues, the increased congestion will lead to further interference among wireless devices, reducing spectrum efficiency and throughput. To counteract this issue, researchers have turned to Cognitive Radio – programmable radios that cooperatively share the wireless spectrum. Two key algorithms in Cognitive Radio are spectrum sensing and the spectrum decision. Before accessing a wireless channel, a cognitive radio must sense or detect channel occupancy and then make a transmit decision based on spectrum policies. The parallelism of spectrum sensing algorithms map well to a Field Programmable Gate Array (FPGA), while the sequential processing of spectrum decision algorithms may be more easily implemented in software. These processing requirements suggest a heterogeneous computing system where parallel algorithms are accelerated by being offloaded to the FPGA fabric.

Recently, FPGA vendors have released System-on-Chip devices that tightly couple programmable logic and a multicore ARM processor. Due to the low latency interconnect, these SoCs show promise as an effective heterogeneous computing system. We have developed CRASH (Cognitive Radio Accelerated with Software and Hardware), a new software and programmable logic framework for Xilinx’s Zynq SoC, to explore their potential in accelerating Cognitive Radio. CRASH provides the framework and interfaces for users to facilitate splitting algorithms between the
Zynq’s ARM processor and FPGA fabric. We implemented CRASH on a Xilinx ZC706 Zynq development board and used an Ettus Research USRP N210 software defined radio as the RF front end. Furthermore, CRASH has been integrated with the GNU Radio software defined toolkit and remains general enough to be integrated with other approaches such as Mathworks’ Simulink.

To demonstrate CRASH, we implemented the spectrum decision in software and offloaded spectrum sensing to the FPGA fabric using our framework. For comparison, we also built a version with both algorithms in software. We determined the performance of each configuration by measuring the latency in sensing unoccupied spectrum and then transmitting in the spectrum. Compared to the purely software implementation, CRASH reduced turnaround time by 2x.

CRASH creates a low latency, high performance cognitive radio platform that simplifies offloading algorithms to programmable logic. This research shows that heterogeneous computing systems, such as CRASH, can provide cognitive radios substantial processing gains without sacrificing programmability.
Acknowledgements

I would like to express my gratitude to my advisor Professor Miriam Leeser for her support and encouragement during my time at Northeastern. This thesis would not have been possible without her advice and constructive criticism. I would also like to thank Professor Kaushik Chowdhury for his insights and guidance on Cognitive Radio.

I want to extend my sincere thanks to Mathworks for their generous support during my graduate studies and Xilinx for their hardware and software donations. As well, I would like to thank the GNU Radio community for supporting my research for a summer semester through Google Summer of Code.

Finally, I want to acknowledge my family, friends, and most importantly my wife Ling. Her patience during my many late nights of studying and encouragement when my confidence wavered helped me to reach one of my lifelong goals.
Contents

1 Introduction ........................................ 13
   1.1 Organization ...................................... 16

2 Background ........................................... 17
   2.1 Heterogeneous FPGA-CPU Devices ............... 18
   2.2 Cognitive Radio .................................. 18
       2.2.1 Spectrum Sensing ............................. 20
       2.2.2 Spectrum Decision ............................. 21
       2.2.3 Universal Software Radio Peripheral ........... 22
       2.2.4 GNU Radio .................................... 22
       2.2.5 Latency in Cognitive Radios ................... 23
   2.3 CRUSH Platform .................................... 24
   2.4 Related Work ..................................... 25
   2.5 Summary .......................................... 27

3 Methodology and Design .............................. 28
   3.1 CRASH Framework Overview ........................ 29
3.2 Hardware ................................................. 29
3.3 HDL Infrastructure ........................................ 30
  3.3.1 AXI Ports ............................................ 31
  3.3.2 AXI-Stream Interconnect .............................. 32
  3.3.3 DMA Block ............................................ 33
  3.3.4 USRP DDR Interface Block ............................ 33
  3.3.5 Spectrum Sensing Block ............................... 35
  3.3.6 USRP Modifications – CRASH DDR Interface Block .... 36
  3.3.7 Resource Utilization ................................. 38
3.4 Software Infrastructure .................................... 38
  3.4.1 CRASH Kernel Module ................................. 39
  3.4.2 LibCRASH .............................................. 40
  3.4.3 Setting USRP Parameters .............................. 42
3.5 CRASH GNU Radio Integration ............................. 42
  3.5.1 Gr-crash .............................................. 42
  3.5.2 CRASH USRP Sink and Source Blocks ................ 43
  3.5.3 Other Blocks .......................................... 43
3.6 Summary .................................................. 45

4 Experimental Setup and Results .............................. 46
  4.1 Experiments: Evaluating Processing Performance and Latency with CRASH .............................. 47
4.1.1 Experiment Design ........................................ 48
4.1.2 Experiment 1: Spectrum Sensing and Spectrum Decision in FPGA Fabric ........................................ 49
4.1.3 Experiment 2: Spectrum Sensing in FPGA Fabric, Spectrum Decision in ARM ........................................ 51
4.1.4 Experiment 3: Spectrum Sensing and Spectrum Decision in ARM ........................................ 52
4.1.5 Measurement of Individual Latencies in CRASH ........................................ 52
4.2 Results .......................................................... 53
4.2.1 Turnaround Time Measurements ........................................ 53
4.2.2 CRASH Processing Performance ........................................ 53
4.2.3 CRASH Latency Breakdown ........................................ 56
4.3 Summary .......................................................... 58

5 Conclusions and Future Work ........................................ 59
5.1 Conclusions .......................................................... 59
5.2 Future Work .......................................................... 60

Appendices .......................................................... 61

A List of Acronyms .......................................................... 61
List of Figures

2.1 Xilinx Zynq System-on-Chip, a Heterogeneous FPGA-CPU Computing System. Image originally published in [8].............. 19
2.2 The Cognitive Radio Cycle. Image originally published in [9]. ....... 20
2.3 Spectrum Sensing Algorithm Overview ......................... 21
2.4 Ettus Research USRP N210 ........................................ 23
2.5 Missing a Spectrum Hole due to Processing Latency .......... 24
2.6 CRUSH Custom Circuit Board ..................................... 25
3.1 The CRASH Framework ............................................. 30
3.2 Hardware used with the CRASH Framework ................. 31
3.3 DMA Block Diagram .............................................. 34
3.4 USRP DDR Interface Block Diagram ............................. 35
3.5 Spectrum Sensing Block Diagram ................................. 36
3.6 CRASH USRP Modifications .................................... 37
3.7 CRASH-USRP Data Transfer Format .............................. 38
3.8 Flow Graph of CRASH USRP Source GNU Radio Block and FFT GUI Sink ........................................ 44
3.9 Flow Graph of Spectrum Sensing and the Spectrum Decision in GNU Radio

4.1 Measurement of Turnaround Time ............................................ 49
4.2 Measuring Turnaround Time in the Lab .................................... 50
4.3 Dataflow Path for each CRASH Experiment .......................... 50
4.4 Average Turnaround Time for each Experiment .................... 53
4.5 CRASH Variation due to FFT Sampling Window. 1) Too Early 2) Slightly Too Early - Still Above Threshold 3) Below Threshold ......... 55
4.6 Breakdown of Individual Latencies per Experiment ................. 57
List of Tables

3.1 FPGA Resource Utilization of the CRASH Framework on the Zynq 7045 ........................................ 38

3.2 CRASH Kernel Module Functions ........................................ 40

3.3 LibCRASH API ........................................ 41

4.1 Location of Spectrum Sensing and Spectrum Decision Processing for each Experiment ........................................ 47

4.2 Average Latency of Individual Components for each Experiment . . . 58
Chapter 1

Introduction

Many applications employ a mix of algorithms that require both sequential and parallel processing. Previous research has found that a heterogeneous computing system [1–3] can significantly reduce computation time by partitioning algorithms to processing units (CPU, GPU, or FPGA) best suited for the particular workload. For instance, a paper on accelerating cardiac physiological mapping [4] describes the design of a FPGA-GPU-CPU heterogeneous architecture that increased imaging performance by 280x over the baseline CPU implementation.

However, these systems tend to connect the various computation units via system wide buses, which can introduce a large amount of delay. In the case of an application that requires low latency processing, such as cognitive radio, a large bus delay can degrade overall system performance or force the designer to use faster, but sub-optimal algorithms. The release of System-on-Chips from Xilinx [5] and Altera [6] with both FPGA fabric and ARM processors on a single die shows promise in accelerating classes of applications like cognitive radio that use both parallel and
sequential processing but have tight timing requirements.

Using these SoCs with mixed CPU and FPGA processing brings additional challenges. The user must build an infrastructure to efficiently communicate across different architectures with different design methodologies. The FPGA infrastructure must allow a high degree of freedom in adding additional processing blocks while adhering to a standard interface for communication between the FPGA and CPU. The CPU typically runs a modern operating system such as Linux, which brings a wealth of reusable code, but requires writing complex kernel drivers. This thesis presents the design of a new framework, CRASH: Cognitive Radio Accelerated with Software and Hardware, that addresses these challenges on Xilinx’s Zynq SoC. CRASH provides the framework necessary to allow users to partition designs and algorithms between processing blocks in the FPGA fabric and programs running in Linux on the ARM processors. Architectural and software details, such as data transfer from user programs to FPGA processing blocks or accessing control & status registers in the FPGA, are handled by the CRASH framework, allowing researchers to focus on algorithm development.

CRASH targets accelerating cognitive radio – a developing research field in wireless communication that attempts to efficiently utilize our increasingly congested wireless spectrum. Two fundamental steps in a cognitive radio are spectrum sensing and spectrum decision. Since most cognitive radios are implemented on a software defined radio, both of these algorithms execute in software. However, widely used
spectrum sensing algorithms, such as energy detection, have a parallel structure easily exploited on FPGAs. Faster spectrum sensing allows the cognitive radio to more effectively react to changing channel conditions. Conversely, spectrum decision algorithms tend to draw on diverse sources of information, such as spectrum policies, databases, and higher-level algorithms, that are more easily implemented in software than hardware. The CRASH framework on the Xilinx Zynq SoC enables a cognitive radio to utilize both kinds of processing.

This thesis makes the following contributions:

- Introduction of the CRASH framework that exploits the Xilinx Zynq’s heterogeneous computing characteristics to enable low latency processing with both FPGA and CPU resources,

- Implementation of spectrum sensing and spectrum decision algorithms with CRASH,

- Integration of CRASH in the GNU Radio Software Defined Radio toolkit, and

- Quantification of the sources of latency both with and without CRASH.

While the CRASH framework has been demonstrated as a cognitive radio, it is generic enough to be used in other applications requiring a mix of FPGA acceleration and CPU processing.
CHAPTER 1. INTRODUCTION

1.1 Organization

This thesis is organized as follows. In Chapter 2 we present background and related work. Chapter 3 introduces the CRASH framework. Chapter 4 describes the experiments we performed and their results involving processing latency in the spectrum sensing and spectrum decision algorithms using CRASH. These experiments allow us to determine the impact of offloading spectrum sensing to the FPGA fabric. We conclude and discuss future directions in Chapter 5.
Chapter 2

Background

This chapter contains background information necessary for the understanding of this thesis. We discuss state of the art heterogeneous FPGA-CPU processing devices, Cognitive Radio fundamentals, spectrum sensing and spectrum decision algorithms, the GNU Radio software defined radio toolkit, the CRUSH platform, and conclude with related work.
2.1 Heterogeneous FPGA-CPU Devices

Xilinx and Altera have both released System-on-chips, the Zynq SoC [5] and the Cyclone V SoC [6], that pair programmable logic with a dual core ARM Cortex A9 processor. Both families of devices have similar capabilities in processor speed and FPGA resources. Their processor and programmable logic are interconnected via ARM’s Advanced Microcontroller Bus Architecture (AMBA) using the Advanced eXtensible Interface (AXI) [7], an interface designed for high throughput transfers between the CPU and peripheral devices. From a high level perspective, the Zynq and Cyclone SoCs are heterogeneous computing systems with the capability to partition workloads to the CPU or the FPGA fabric to exploit their inherent sequential or parallel processing structure. The CRASH framework is implemented on the Xilinx Zynq SoC. The Xilinx’s Zynq Architecture is pictured in Figure 2.1. CRASH uses AXI ACP and a General Purpose AXI Port for communication between the ARM processors and the programmable logic. While the framework currently only supports the Zynq SoC, due to the highly similar architecture between both devices one could expect similar performance if CRASH were ported to the Cyclone V SoC.

2.2 Cognitive Radio

The wireless spectrum continues to become increasingly crowded as more devices gain wireless connectivity. Cognitive radios seek to alleviate the congestion by efficiently utilizing the limited available spectrum through either cooperative spectrum sharing
Figure 2.1: Xilinx Zynq System-on-Chip, a Heterogeneous FPGA-CPU Computing System. Image originally published in [8].

among several nodes or opportunistically using a channel when the primary user is not present. In [9], the authors present the cognitive cycle, shown in Figure 2.2, breaking the functions of a cognitive radio into discrete parts. Cognitive radios use spectrum sensing to search for available spectrum; the spectrum decision is the choice to transmit on available spectrum. Spectrum mobility and spectrum sharing implement the cognitive radio’s ability to avoid interference with a channel’s primary user and other nodes, respectively. This thesis focuses on spectrum sensing and the spectrum decision implemented with the CRASH framework.
2.2.1 Spectrum Sensing

Spectrum sensing is the fundamental method cognitive radios use to detect *spectrum holes*: sections of empty spectrum available for transmission. Spectrum sensing encompasses many different algorithms with time-domain and frequency-domain based energy detection being the most popular [10]. The downside to energy detection in the time domain is that the algorithm does not provide any information on the distribution of the energy across frequency. For instance, the transmitting device could be occupying a subset of the spectrum available. With that information, the cognitive radio could avoid those frequencies. In this thesis, we perform frequency-domain based energy detection by applying the Fast Fourier Transform (FFT) with thresholding to the radio sample data. Figure 2.3 provides an overview of the algorithm.
2.2.2 Spectrum Decision

After determining that unused spectrum exists, the cognitive radio needs to decide whether or not to transmit. This process is called the spectrum decision. The decision can involve information from many sources: the spectrum sensing data, network topology and routing metrics, adaptive machine learning algorithms, and even regulatory policies. In fact, the FCC has released rules [11] requiring that radios operating in the TV white space bands must access a spectrum database of valid transmission frequencies. In this case, a cognitive radio’s spectrum decision would include drawing information from a database over the Internet. Beyond the binary decision of whether or not to transmit, the cognitive radio must also set the parameters of the transmission, such as data rate, modulation, and error correction, to best utilize the spectrum. CRASH implements a simple spectrum decision: transmit only if all frequency bins in the sensed spectrum do not exceed a set threshold.
2.2.3 Universal Software Radio Peripheral

Cognitive radios learn and adapt to changing channel conditions, which require a high degree of reconfigurability. Software defined radios provide such capability by moving as much signal processing as possible from fixed hardware to software or reconfigurable hardware. The Universal Software Radio Peripheral (USRP) \cite{12} family of software defined radios are widely used for implementing experimental cognitive radios. The USRP hosts the transmitter and receiver circuitry along with a small FPGA to send and receive sample data over Ethernet. A typical system will pair the USRP with a powerful desktop computer running a software defined radio library, such as GNU Radio, to implement the digital transmitter and receiver functions. A drawback to this system is the large latency associated with Ethernet communication between the USRP and the host computer. CRASH uses the USRP N210 with the WBX \cite{13} daughterboard as its RF front end. The USRP N210 has a receive and transmit bandwidth of 100MHz and can tune from 50-2200MHz. CRASH avoids the USRP’s Ethernet latency by using a high speed serial interface as discussed later in Section 2.3.

2.2.4 GNU Radio

For software implementation of a cognitive radio, many researchers utilize the GNU Radio software defined radio toolkit \cite{14}. GNU Radio provides a library of signal processing blocks, the programming infrastructure to connect blocks to create a sig-
nal processing chain (called a flow graph), a run-time environment to execute the flow graph, and the ability to interface with radio hardware to transmit and receive waveforms. GNU Radio includes a graphical user interface tool called GNU Radio Companion to easily create flow graphs. The popularity of GNU Radio in the cognitive radio community stems from its ease of creating new algorithms, rapidly modifying existing algorithms, and reuse of signal processing code. This thesis shows how CRASH integrates with GNU Radio to provide the ability to accelerate signal processing blocks by offloading them to the FPGA fabric.

2.2.5 Latency in Cognitive Radios

The process of spectrum sensing and the spectrum decision involves substantial computation, but the overall cognitive radio decision to utilize spectrum is highly sensitive to latency. Any delay in the spectrum decision process can cause poor spectrum utilization and possible interference with other users. Figure 2.5 shows how the spectrum sensing algorithm in a cognitive radio could miss the beginning of a spectrum hole.
due to latency. Therefore, cognitive radios must not only focus on quickly processing sample data, but also avoid architectures that incur delays due to the communication interface with the radio front end.

![Figure 2.5: Missing a Spectrum Hole due to Processing Latency](image)

2.3 CRUSH Platform

Cognitive radios that use the USRP face a latency penalty due to the USRP’s Ethernet interface. The USRP uses the TCP/IP protocol over Ethernet, which introduces additional header and routing information to the data stream. Even for a point-to-point connection, this overhead can add up to a millisecond or more latency. The CRUSH (Cognitive Radio Universal Hardware Software) [15] platform reduces the interface latency with the USRP by eschewing Ethernet for a direct high speed serial interface to a FPGA board. It allows the USRP to stream its Analog to Digital Converter (ADC) samples at full rate with no overhead. This high speed connection
allows CRUSH to accelerate spectrum sensing with the FPGA fabric. The CRASH framework employs CRUSH with the USRP for the same advantage. CRASH further extends the interface by adding the capability to stream Digital to Analog Converter (DAC) samples, i.e. to transmit as well as receive. Figure 2.6 is an image of CRUSH’s custom circuit board that converts Matched Impedance Connector (MIC-TOR) to FPGA Mezzanine Card (FMC) standard, allowing a USRP to interface with an off-the-shelf FPGA development board.

2.4 Related Work

Two recent papers discuss heterogeneous computing with the Zynq SoC. The first pairs the USRP with the Zynq SoC to make a software defined radio [16]. The paper focuses on porting their SDR library Iris to the ARM processor, as well as methods to identify blocks for potential acceleration in the Zynq’s programmable logic. The second paper [17] presents improvements to OmpSs, a programming model
that supports heterogeneous execution, to simplify offloading processing tasks to the Zynq SoC’s programmable logic. While both these papers propose frameworks similar to CRASH, neither presents a completed framework with components running on both the FPGA fabric and the CPU.

Beyond the Zynq SoC, several platforms use both a processor and FPGA processing for cognitive radio research. WARP from Rice University [18] uses a Virtex 4 FPGA with an embedded PowerPC processor. WARP was used by Gupta et al. [19] to make a cognitive radio testbed called WARPnet. Microsoft’s SORA platform [20] is a software defined radio that pairs a commodity PC with a Radio Control Board and RF card communicating to the host processor over PCI-Express. The Radio Control Board features a Virtex 5 FPGA, which controls the RF cards and can offload some signal processing tasks such as sample filtering. Bajaj et al. [20] used the system to perform energy detection based spectrum sensing in the WiFi 2.4 GHz bands, but it had very long sensing times (4 seconds per channel) which would not expose any latency issues related to PCI-Express communication. The previously mentioned CRUSH platform pairs a USRP with a Xilinx ML605 FPGA development board via a high-speed serial interface to perform low latency spectrum sensing. CRUSH transfers the results of spectrum sensing over Ethernet to a host computer incurring a moderate delay (on the order of 100 $\mu$s).


CHAPTER 2. BACKGROUND

2.5 Summary

This chapter provided a background on heterogeneous processing devices and Cognitive Radio. We discussed CRASH’s base hardware: the Zynq System-on-Chip and the USRP N210. The spectrum sensing and spectrum decision algorithms were overviewed along with sensing and decision latencies’ effect on spectrum utilization. We detailed how CRASH utilizes CRUSH’s hardware development to create a low latency interface with the USRP. Finally, we compared and contrasted CRASH to related projects. The next chapter discusses the methodology and design of the CRASH framework.
Chapter 3
Methodology and Design

In this chapter, we introduce the design and implementation of the CRASH framework. CRASH is a software and HDL infrastructure that allows users to offload algorithms to the Zynq’s FPGA fabric. The framework consists of both a HDL and software component. The framework targets cognitive radio by adding processing blocks for spectrum sensing and interfacing with RF hardware, but remains flexible enough to accelerate many other applications. We discuss the HDL and software design behind CRASH and our spectrum sensing implementation in programmable logic. The chapter closes with an overview of integrating CRASH into the GNU Radio software defined radio toolkit.
3.1 CRASH Framework Overview

The CRASH framework exploits the heterogeneous computing capabilities of the Zynq SoC to accelerate applications requiring low latency sequential and parallel processing. It consists of a HDL and software component. The HDL component provides the programmable logic interface to allow algorithms implemented in the FPGA fabric to communicate with software programs running on the CPU. The software component runs under Linux and uses a kernel module to facilitate data transfer between user programs and FPGA processing blocks. It is important to note that CRASH does not automate the HDL implementation of software algorithms; the user is responsible for writing the algorithm in HDL. Figure 3.1 shows a high level overview of the CRASH framework.

3.2 Hardware

While the CRASH framework can be built on any Zynq SoC device, we implemented CRASH on Xilinx’s ZC706 development board [21] along with the USRP N210 as the RF front end. CRASH uses CRUSH’s custom interface board and a MICTOR cable to incorporate a low latency, high speed serial interface between the USRP and the ZC706. By using off-the-shelf components such as the USRP and the ZC706, CRASH can upgrade to the latest hardware without a significant redesign. Figure 3.2 shows a picture of the hardware.
CRASH’s HDL infrastructure provides users the ability to offload algorithms to the Zynq’s programmable logic to increase processing performance. To accomplish this, the infrastructure includes an AXI-Stream interconnect to allow processing blocks to communicate with each other, a DMA block to oversee data transfers to/from the CPU via AXI ACP, and interface code to the general purpose AXI ports to allow control/status register access. The infrastructure also includes two blocks related to cognitive radio processing: a USRP interface block and a spectrum sensing block.
3.3.1 AXI Ports

Data transfers between the ARM processors and FPGA fabric on the Zynq occur via AXI Ports. The nine AXI ports use the AMBA AXI interface standard and can be seen in Figure 2.1: four AXI General Purpose (GP) ports, four AXI High Performance (HP) ports, and one AXI Accelerator Coherency Port (ACP). CRASH uses one AXI GP port and AXI ACP as shown in Figure 3.1.

The AXI GP ports are low performance ports useful for access to memory mapped devices. The ports are 32-bits wide and split into two slave and two master ports. CRASH uses one AXI GP port to read/write access to status and control registers in processing blocks. Each processing block has an address space of 256 32-bit words. Our experiments show that control and status registers have an access time of approximately 150 ns from software running in Linux.
The four AXI HP ports allow high throughput data transfer between the ARM processors and FPGA fabric. Similar to the AXI GP ports, they are split into two master and two slave ports. The AXI HP ports can directly read and write to the CPU’s RAM via the multiport DRAM memory controller. Each port is 64-bits wide and capable of operating at 150 MHz. This results in a peak 1.2 GB/sec transfer rate for both reads and writes.

AXI ACP has the same features as a slave AXI HP port, except that it can directly access the CPU’s cache. This maintains cache coherency which reduces data access latency for software executing on the CPU. Due to this advantage, CRASH uses AXI ACP for bulk data transfers, such as transmit and receive sample data and the output of the Spectrum Sensing block. While the theoretical peak transfer rate is 1.2 GB/sec, our experiments have found approximate 80% theoretical (or 960 MB/sec) to be a more realistic expectation due to protocol overhead and bus utilization by other devices. To minimize CPU overhead when executing data transfers, CRASH uses Direct Memory Access (DMA) on the AXI ACP interface.

### 3.3.2 AXI-Stream Interconnect

In the programmable logic section of Figure 3.1, we can see that CRASH allows the user to place processing blocks in the FPGA fabric which are connected to a common AXI-Stream interconnect. The interconnect is based on Xilinx’s AXI-Stream Interconnect IP [22] and can support up to 16 processing blocks in a full crossbar configuration. Each processing block can transfer data to any other block at up to
1.2 GB/sec. This value is derived from the interconnect clock rate of 150 MHz times the interconnect data bus width of 64 bits. All processing blocks in the FPGA fabric operate at the 150 MHz clock rate. Each block’s interface is standardized to include both the AXI-Stream interconnect ports and an interface to memory mapped I/O control and status registers.

3.3.3 DMA Block

The Direct Memory Access (DMA) block is used to shuttle data from other processing blocks to the CPU over the AXI ACP interface and vice versa. As shown in Figure 3.3, the DMA block is based on Xilinx’s DataMover IP core which provides the DMA functionality of transferring data from processing blocks to the CPU. It is important to note that processing blocks interface with the AXI-Stream interconnect, but the AXI ACP interface uses the AMBA AXI bus standard. The DataMover IP core provides the bridge between the buses. To setup DMA transfers between the programmable logic and the CPU, user programs configure the DMA block’s control registers. Once configured, the DMA block will initiate memory reads and writes without further intervention by the processor.

3.3.4 USRP DDR Interface Block

CRASH sends and receives sample data with the USRP via the USRP DDR Interface block. Figure 3.4 shows a block diagram of the USRP DDR Interface block. The USRP’s ADC and DAC both sample at 100 million-samples/sec and use complex
Figure 3.3: DMA Block Diagram

data. Since CRASH runs at a different clock rate than the USRP, this block includes logic and FIFOs to synchronize data between the clock domains. As well, the block includes programmable decimation and interpolation filters (based on Xilinx IP cores) for increasing or reducing the sample rate of transmit and receive samples. This is an important preprocessing stage in any cognitive radio that the FPGA can implement more efficiently than the processor due to highly parallel and pipelined digital filter structures. Decimation and interpolation filters in the USRP’s FPGA firmware could have been used in place of the filters in this block. However, we built our own filtering chain in order to have more control over both the length of the filters (which affects latency) and the filter responses. Finally, the block features by-passable fixed point
to single precision floating point conversion blocks for both the transmit and receive sample data paths. Performing the conversions in the FPGA is another time saving method as most signal processing algorithms implemented on a CPU are in floating point.

Figure 3.4: USRP DDR Interface Block Diagram

### 3.3.5 Spectrum Sensing Block

The last processing block, Spectrum Sensing, implements the cognitive radio spectrum sensing and spectrum decision algorithms. Figure 3.5 illustrates how the spectrum sensing and spectrum decision algorithms are implemented in the Spectrum Sensing block. All processing in this block is in single precision floating point and many blocks are based on customized Xilinx IP cores. The spectrum sensing algorithm uses a variable size FFT and magnitude calculation pipeline for frequency-domain based energy detection. The spectrum decision algorithm consists of threshold detection where the output of the spectrum sensing algorithm is compared against
a threshold. When the magnitude of the FFT output falls below (or exceeds depending on the block configuration) the threshold, the block can cause an interrupt in the processor or strobe an output signal. This mechanism allows the block to automatically trigger other blocks when a transmit decision has been reached. Most stages in the Spectrum Sensing block can be bypassed to customize the block’s output. For instance, the FFT’s result can be forwarded to the block’s output allowing the user access to the raw frequency spectrum data.

![Figure 3.5: Spectrum Sensing Block Diagram](image)

### 3.3.6 USRP Modifications – CRASH DDR Interface Block

As mentioned in Section 2.3, CRASH incorporates CRUSH’s USRP HDL modifications to provide a high speed serial interface to receive and transmit sample data. The modifications are contained in a block called the *CRASH DDR Interface* added to the USRP HDL firmware. Figure 3.6 provides a block diagram. The CRASH DDR Interface block connects with the USRP’s ADC and DAC data buses, which operate at 100 MHz with complex 14-bit and 16-bit samples respectively. Due to a limited
number of pins on the USRP to ZC706 interface (via the MICTOR connector and custom circuit board) and the need to use LVDS signaling to improve signal integrity, the block transfers ADC and DAC data at half the bit width (7-bits for ADC, 8-bits for DAC) at quadruple the clock rate (200 MHz DDR). The need to send the data at quadruple the clock rate arises from the fact that the samples are complex, effectively doubling the number of samples. Figure 3.7 illustrates the USRP-CRASH transfer format for ADC and DAC data.

The CRASH DDR Interface block also includes a UART to allow CRASH to control the block’s mode. The modes affect the source of the ADC or DAC data (raw versus DC filtered) and the selection of test modes. To remain minimally intrusive, the CRASH DDR Interface block includes a bypass mode, allowing the USRP to operate as if it were loaded with unmodified firmware.

![Figure 3.6: CRASH USRP Modifications](image-url)
3.3.7 Resource Utilization

Table 3.1 provides CRASH’s FPGA resource utilization on a Zynq 7045 device. These values include the Spectrum Sensing and USRP DDR Interface blocks.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Available</th>
<th>Percent Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Registers</td>
<td>38,270</td>
<td>437,200</td>
<td>8%</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>29,772</td>
<td>218,600</td>
<td>13%</td>
</tr>
<tr>
<td>DSP48s</td>
<td>216</td>
<td>900</td>
<td>24%</td>
</tr>
</tbody>
</table>

3.4 Software Infrastructure

In the CRASH framework, the Zynq’s dual core Cortex A9 ARM processor runs Linux as the operating system. This provides access to a large amount of Linux based software including software defined radio libraries such as GNU Radio. Linux
employs memory protection with virtual memory, preventing unprivileged user programs from directly accessing specific memory address. However, user programs need such addressing capability to access the processing blocks embedded in the FPGA fabric. To provide this functionality, CRASH uses a custom written kernel module, as shown in the processing system section of Figure 3.1, as the middle-man between the user program and the FPGA processing blocks facilitating communication. Directly interfacing with a kernel module can be cumbersome, so we have written an API called LibCRASH, which resides between the user software and the kernel module.

### 3.4.1 CRASH Kernel Module

The CRASH kernel module, called `crash-kmod`, provides the gateway for user programs to access processing blocks in the programmable logic. Crash-kmod performs the kernel level configuration required for user software to communicate with processing blocks. It allocates contiguous memory for DMA buffers, enables interrupts to signal the completion of DMA transfers, and maps control/status registers into user accessible memory.

Every device in Linux is treated as a file. To support this paradigm, we extend this treatment to individual processing blocks in the programmable logic. From the user perspective, this means to access their processing block they must use file I/O system calls: `open`, `close`, `mmap`, and `ioctl`. Table 3.2 describes these function’s responsibilities in crash-kmod. As required for all Linux kernel modules, crash-kmod includes the `init`, `exit`, `probe`, and `remove` functions. These functions are called directly
by the kernel at boot up and perform tasks such as informing Linux about the module’s capabilities and registering interrupt callbacks.

Table 3.2: CRASH Kernel Module Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>Allocates contiguous memory for DMA buffers</td>
</tr>
<tr>
<td>close()</td>
<td>Deallocates DMA buffers</td>
</tr>
<tr>
<td>mmap()</td>
<td>Maps DMA buffers and control/status registers to user space</td>
</tr>
<tr>
<td>ioctl()</td>
<td>Performs read and write DMA transfers.</td>
</tr>
</tbody>
</table>

The functions outlined in Table 3.2 require the user to have detailed knowledge of the kernel module code. For instance to perform a DMA transfer, the user must know to use `ioctl` with a control word that specifics the direction of the DMA, number of bytes, and destination processing block. To reduce the complexity of using CRASH in software, we provide an API library called LibCRASH that wraps the calls to crash-kmod with easier to use functions.

### 3.4.2 LibCRASH

*LibCRASH* includes functions and macros to simplify DMA transfer setup and accessing control/status registers in processing blocks. Table 3.3 provides a list of the available LibCRASH functions.

To interact with a processing block, the user must open it with `crash_open` with the block’s identification number as an input. LibCRASH will allocate a DMA buffer for the processing block, map the buffer and the block’s control/status registers to user space, and return a `pblock` structure. The pblock structure contains informa-
Table 3.3: LibCRASH API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crash_open()</td>
<td>Opens a processing block; Allocates DMA buffer and maps control/status registers</td>
</tr>
<tr>
<td>crash_close()</td>
<td>Closes a processing block; Deallocates DMA buffer and unmaps control/status registers</td>
</tr>
<tr>
<td>crash_read()</td>
<td>Perform DMA read from processing block</td>
</tr>
<tr>
<td>crash_write()</td>
<td>Perform DMA write to processing block</td>
</tr>
<tr>
<td>crash_reset()</td>
<td>Global reset of programmable logic</td>
</tr>
<tr>
<td>crash_read_reg()</td>
<td>Reads control/status register</td>
</tr>
<tr>
<td>crash_write_reg()</td>
<td>Writes value to control register</td>
</tr>
<tr>
<td>crash_get_bit()</td>
<td>Reads a single bit from control/status register</td>
</tr>
<tr>
<td>crash_set_bit()</td>
<td>Sets a single control register bit</td>
</tr>
<tr>
<td>crash_clear_bit()</td>
<td>Clears a single control register bit</td>
</tr>
</tbody>
</table>

A function describing the processing block along with pointers to its DMA buffer and control/status registers. Each processing block must be opened individually and can be opened multiple times. Opening a block multiple times is useful for blocks that may require several DMA buffers, for example for the USRP DDR Interface block’s transmit and receive samples. Once opened, the user can initiate DMA transfers with the `crash_read` and `crash_write` functions. The various register helper macros (`crash_read_reg`, `crash_write_reg`, etc.) allow the user to safely manipulate specific registers without overwriting other registers in the same bank. For instance, `crash_set_bit` reads the entire register, sets only the specified bit, and writes the resulting register value back. To close the processing block, the user calls `crash_close`. 
3.4.3 Setting USRP Parameters

The USRP serves as CRASH’s RF front end and has several parameters for tuning the center frequency and gain for the transmitter and receiver. CRASH lacks a mechanism within the framework to set these parameters. Instead the user must execute a separate program (or GNU Radio flow graph) that sets them through the USRP’s native control library, libUHD. In the future, we plan to include the ability to configure the USRP in CRASH.

3.5 CRASH GNU Radio Integration

GNU Radio is a popular software defined radio toolkit that is widely used for cognitive radio research. It includes a library of signal processing blocks that can reduce algorithm development time. We added support for CRASH in GNU Radio to both bring FPGA acceleration to GNU Radio and to further simplify using CRASH for cognitive radio development.

3.5.1 Gr-crash

GNU Radio adds additional functionality through modules, which contain one or more GNU radio processing blocks. For instance, the module gr-filter contains blocks for FIR and IIR filters. We created a module called gr-crash that contains four CRASH specific blocks and one general purpose block: CRASH USRP Sink, CRASH USRP Source, CRASH Spectrum Sense Source, CRASH Triggered Transmit Sink, and Spectrum Decision. These blocks use our LibCRASH API to simplify interfacing
3.5.2 CRASH USRP Sink and Source Blocks

The CRASH USRP Sink and CRASH USRP Source gr-crash blocks interface with the USRP DDR Interface block as described in Section 3.3.4. These blocks provide a low latency path for receive and transmit sample data and support sample rates up to 100 million-samples/sec. This exceeds the capabilities of the USRP’s native Gigabit Ethernet interface, which has a maximum rate of 25 million-samples/sec for both receive and transmit.

The sink block has an adjustable interpolation rate and writes transmit samples to the USRP DDR Interface block via DMA. The source block has an adjustable decimation rate and reads in receive samples. Figure 3.8 demonstrates the CRASH USRP Source GNU Radio block receiving USRP sample data at 100 million-samples/sec and using a GNU Radio FFT GUI Sink to display the 100 MHz of spectrum. The center frequency is tuned to 90 MHz and the various peaks are local FM radio stations.

3.5.3 Other Blocks

The remaining gr-crash blocks, CRASH Spectrum Sense Source, CRASH Triggered Transmit Sink, and Spectrum Decision, work together to produce a GNU Radio flow graph (shown in Figure 3.9) that performs FPGA accelerated spectrum sensing with CRASH and triggers a transmission if the spectrum is empty. This flow graph provides a starting point for researchers to both explore FPGA acceleration in GNU
CHAPTER 3. METHODOLOGY AND DESIGN

Figure 3.8: Flow Graph of CRASH USRP Source GNU Radio Block and FFT GUI Sink

Radio and develop more complex spectrum decision algorithms.

The flow graph starts with the CRASH Spectrum Sense Source block. It receives FFT data via DMA from the FPGA accelerated Spectrum Sensing block (discussed in Section 3.3.5) and passes the data to the Spectrum Decision block. The Spectrum Decision block performs thresholding and makes a decision: transmit if the spectrum

Figure 3.9: Flow Graph of Spectrum Sensing and the Spectrum Decision in GNU Radio
does not exceed an adjustable threshold. A true decision triggers the final block, CRASH Triggered Transmit Sink, to transmit a tone by performing a DMA transfer of transmit samples to the USRP DDR Interface block.

3.6 Summary

This chapter overviewed two of the three thesis contributions: the CRASH framework and our implementation of spectrum sensing and the spectrum decision within the framework. We described CRASH’s HDL and software infrastructure that allows users to offload processing algorithms to the Zynq’s FPGA fabric. In the next chapter, we discuss our experiments using the CRASH framework to offload spectrum sensing to programmable logic to increase overall system processing performance.
Chapter 4

Experimental Setup and Results

This chapter describes our experiments to quantify CRASH’s effect on the latency and processing performance, specifically for spectrum sensing and the spectrum decision. We use a metric called turnaround time to measure how quickly CRASH can detect a spectrum hole and transmit. This metric provides a basis of comparison for how CRASH’s heterogeneous computing capabilities can reduce latency and improve processing performance versus a pure software implementation. We discuss our experimental results and provide a quantitative analysis of the sources of latency in CRASH.
4.1 Experiments: Evaluating Processing Performance and Latency with CRASH

CRASH provides a framework for applications to implement their algorithms both with the Zynq’s ARM processors and the FPGA fabric. However, data transfers between the programmable logic and the ARM processors adds delay. Ideally, the increase in processing performance outweighs the penalty incurred due to data transfer. To explore this, we implemented the spectrum sensing and spectrum decision algorithms in three configurations:

1. Both algorithms executing in programmable logic,

2. Using CRASH to offload spectrum sensing to the FPGA fabric, and

3. Both algorithms executing in software.

Table 4.1 summarizes the configurations. The following sections provide a description of the experiment design, how we measure CRASH’s performance, and the three experiment configurations.

Table 4.1: Location of Spectrum Sensing and Spectrum Decision Processing for each Experiment

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Spectrum Sensing</th>
<th>Spectrum Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>FPGA</td>
<td>CPU</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
4.1.1 Experiment Design

Measuring the effect of offloading algorithms to programmable logic on latency and processing performance presents a challenge. The chosen metric must be consistently measurable across configurations, composed of several latencies (including the communication delay) to allow comparison, and provide insight into how well CRASH performs as a cognitive radio. We decided to treat the system as a black box and use the *turnaround time* between providing a spectrum hole and the cognitive radio transmitting. The turnaround time is:

1. A "black box" measurement and independent of the experiment configuration,

2. The aggregation of many latencies including delay through the USRP’s analog receiver and transmitter circuitry, transferring data from the USRP to CRASH, spectrum sensing, the spectrum decision, and transferring data between the FPGA fabric and the ARM processors, and

3. A valuable indicator of the responsiveness of CRASH as a cognitive radio.

Expanding on the last point, turnaround times on the order of 10-30 $\mu$s indicates the system could meet 802.11 MAC layer timing requirements [23] – a desirable trait in a cognitive radio.

The oscilloscope display depicted in Figure 4.1 provides a diagram of our test setup. We inject a pulsed sinusoid into the USRP’s receiver. The receive samples from the USRP travel over the USRP-CRASH interface. The experimental configuration
(described in Sections 4.1.2-4.1.4) determines the specific next steps, but from a high level perspective, CRASH will sense the spectrum to detect the presence of the sinusoid. Once detected, CRASH waits until the sinusoid pulse ends. This simulates a *spectrum hole*. Once the spectrum hole is detected by CRASH, it immediately transmits a waveform. The delta time between the end of the input pulse and the beginning of CRASH’s transmission is the turnaround time. Figure 4.2 is an in lab picture of measuring the turnaround time during an experiment.

4.1.2 **Experiment 1: Spectrum Sensing and Spectrum Decision in FPGA Fabric**

In this experiment, spectrum sensing and the spectrum decision occur in the FPGA fabric. The blue path in Figure 4.3 shows the processing path. The USRP DDR Interface block receives USRP samples at 100 million-samples/sec. The samples are converted to floating point and routed via the AXI-Stream interconnect to the Spectrum Sensing block. In this experiment, the Spectrum Sensing block performs
both spectrum sensing and the spectrum decision. When the block determines the spectrum is unoccupied, it triggers the USRP DDR Interface block to send a buffered
transmit waveform to the USRP.

4.1.3 Experiment 2: Spectrum Sensing in FPGA Fabric, Spectrum Decision in ARM

Similar to Experiment 1, sample data enters CRASH through the USRP DDR Interface block and is transferred to the Spectrum Sensing block. However, the output of the FFT and magnitude calculation pipeline is transferred to the ARM processor via the DMA block for thresholding. This moves the *Spectrum Decision* to the ARM processor while keeping the computationally expensive *Spectrum Sensing* in the FPGA fabric. Once the spectrum decision software running on the ARM processor determines the spectrum is unoccupied, it triggers the USRP DDR Interface block to transmit the waveform. The spectrum decision software is a short C program that initializes the CRASH infrastructure then continuously executes DMA transfers of FFT magnitude data from the Spectrum Sensing block. The program performs thresholding on the FFT magnitude data and makes the spectrum decision: transmit only when the spectrum does not exceed the threshold. GNU Radio could have been used for this purpose; however, we found the program introduced an intolerable amount of delay (on the order of milliseconds), which would unfairly skew the results. In our future work, we propose investigating the source of this delay, as GNU Radio would be a valuable resource in these types of experiments.
4.1.4 Experiment 3: Spectrum Sensing and Spectrum Decision in ARM

Building on Experiment 2, this experiment performs both spectrum sensing and the spectrum decision on the ARM processor. The FPGA fabric simply interfaces with the USRP and transfers the samples to the ARM processor. This experiment provides data to compare the processing time of a purely software implementation versus the FPGA fabric.

4.1.5 Measurement of Individual Latencies in CRASH

Three methods are used to breakdown the turnaround time into the individual component latencies: Clock counting, program timers, and RX/TX loopback. The clock counting approach uses Xilinx’s Chipscope to allow observation of the internal FPGA signals and count number of clock cycles certain events take. This method works well for determining the latency of components within or exposed to the FPGA such as the FFT in the Spectrum Sensing block and AXI ACP transfers in the DMA block. However, some latencies, such as spectrum sensing execution time in the ARM and the delays associated with the USRP interface, require other methods. For spectrum sensing in the ARM processor, reading a timer in the FPGA fabric allows for accurate measurement of the execution time. To determine the USRP delay, the receive sample data is looped back to the transmit path and measured using the same technique as the overall system delay.
4.2 Results

4.2.1 Turnaround Time Measurements

For each of the three configurations listed in Section 4.1, turnaround times were measured while varying the FFT size. Figure 4.4 plots the average turnaround times and the standard deviation. It is important to note that the standard deviation is approximately the same between experiments but appears to become smaller due to the logarithmic axis scaling.

![Figure 4.4: Average Turnaround Time for each Experiment](image)

4.2.2 CRASH Processing Performance

From Figure 4.4, we can gain several useful pieces of information. First, as FFT size increases so does the turnaround time. This agrees with theory, as the FFT execution time grows at $O(\log N)$ in the parallel implementation and $O(N \log(N))$ with sequential processing. Second, the latency increases as less work is performed
in the FPGA. Performing spectrum sensing and the spectrum decision in the FPGA has the best performance – 130% faster than experiment 2 and 300-400% faster than experiment 3. Offloading spectrum sensing to the FPGA fabric (experiment 2) versus executing both algorithms in software (experiment 3) is approximately 120% faster. Finally, the larger turnaround time in the third configuration relative to the other two illustrates the performance degradation when performing spectrum sensing in software versus reconfigurable hardware.

The standard deviation is an artifact of the employed spectrum sensing algorithm: frequency-domain energy detection. The signal generator in our experiment transmits the RF pulse independently of CRASH’s FFT sample window. Therefore, as shown in Figure 4.5, the sensed spectrum energy will vary depending on when CRASH began buffering samples relative to the RF pulse. Scenario 1 shows the spectrum hole being missed due to sampling too early, scenario 2 shows the sampling slightly too early so the FFT energy is still above the threshold, and in scenario 3, the hole is detected. This affects all experiment configurations in the same way, resulting in similar standard deviations. To avoid this phenomenon, we could employ time-domain energy detection, but we would lose frequency information as discussed in Section 2.2.1.

As discussed in Section 4.1.1, a turnaround time of less than 30 $\mu$s would be desirable to meet 802.11 MAC layer timing requirements [23]. From Figure 4.4, we can see that all three experiments can meet that goal for a FFT size of 64, but only
experiment 1 and experiment 2 can meet that goal for FFT sizes of 128 and 256. This result shows that a cognitive radio utilizing the CRASH framework could implement a sophisticated spectrum decision algorithm in the ARM processor and still make strict timing requirements.

Finally, we employed a simple spectrum decision: transmit if the spectrum does not exceed a threshold. Most spectrum decision algorithms are more complicated. For instance, recent FCC regulations concerning the use of TV white spaces [11] requiring cognitive radios to access a spectrum database. Accessing a database over the Internet is a task well suited for a CPU which suggests a software implementation of the spectrum decision. CRASH could allow the user to access the database and perform the spectrum decision in software while accelerating spectrum sensing in programmable logic.

Figure 4.5: CRASH Variation due to FFT Sampling Window. 1) Too Early 2) Slightly Too Early - Still Above Threshold 3) Below Threshold
4.2.3 CRASH Latency Breakdown

Table 4.2 and Figure 4.6 present a breakdown of the individual component latencies. It is important to note that for Figure 4.6 the percentages are relative to total latencies for each experiment and FFT size.

The USRP RX & TX Path delays show a lower bound of approximately 600 ns for turnaround time of this system – where the RF input to the USRP is looped back for immediate transmission.

For the three experiments, the majority of the overall delay (60-95%) comes from spectrum sensing and the spectrum decision. Conversely, in experiments 2 and 3 the DMA transfers time only contributes 5-25%. This is a favorable outcome as we want to maximize the time performing computations and minimize the time spent transferring data.

The performance of experiment 3 suffers the most compared to the first two. Even with the use of the FFTW [24] FFT library with specific ARM Processor optimizations, spectrum sensing in experiment 3 contributes an overwhelming 70-80% of the overall latency. This exemplifies the need to accelerate certain functions in the FPGA fabric. By offloading spectrum sensing with the CRASH framework, the time spent spectrum sensing is reduced to 25-40%.

It is also interesting to compare the execution time for spectrum decision in experiments 2 versus 3. Both execute the same algorithm on the ARM processor, but a large gap exists across all FFT sizes. We speculate the issue may involve compiler
optimization. In experiment 2, the spectrum decision loads spectrum sensing data from the Zynq’s L1 cache placed there by the Spectrum Sensing block in the programmable logic. Experiment 3 reads receive sample data from the cache (placed by the USRP DDR Interface block) and executes an FFT. Then, for each FFT bin the magnitude is calculated and thresholded. These two steps occur one FFT bin at a time. In the case of experiment 3, the compiler could be storing the thresholding result in a CPU register allowing faster access by the spectrum decision code versus accessing the cache in experiment 2.

![Figure 4.6: Breakdown of Individual Latencies per Experiment](image-url)
Table 4.2: Average Latency of Individual Components for each Experiment

<table>
<thead>
<tr>
<th>Latency Component</th>
<th>Exp. 1</th>
<th>Exp. 2</th>
<th>Exp. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>USRP RX &amp; TX Paths</td>
<td>0.6 µs</td>
<td>0.6 µs</td>
<td>0.6 µs</td>
</tr>
<tr>
<td>USRP - CRASH Interface</td>
<td>0.7 µs</td>
<td>0.7 µs</td>
<td>0.7 µs</td>
</tr>
<tr>
<td>Spectrum Sensing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Point FFT</td>
<td>2.6 µs</td>
<td>2.5 µs</td>
<td>13 µs</td>
</tr>
<tr>
<td>512 Point FFT</td>
<td>18 µs</td>
<td>18 µs</td>
<td>96 µs</td>
</tr>
<tr>
<td>4096 Point FFT</td>
<td>149 µs</td>
<td>149 µs</td>
<td>747 µs</td>
</tr>
<tr>
<td>Spectrum Decision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Point FFT</td>
<td>0.8 µs</td>
<td>3.3 µs</td>
<td>2.0 µs</td>
</tr>
<tr>
<td>512 Point FFT</td>
<td>3.4 µs</td>
<td>25 µs</td>
<td>15 µs</td>
</tr>
<tr>
<td>4096 Point FFT</td>
<td>28 µs</td>
<td>203 µs</td>
<td>121 µs</td>
</tr>
<tr>
<td>FPGA-CPU DMA Delay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Point FFT</td>
<td>N/A</td>
<td>2.3 µs</td>
<td>2.2 µs</td>
</tr>
<tr>
<td>512 Point FFT</td>
<td>N/A</td>
<td>6.3 µs</td>
<td>6.6 µs</td>
</tr>
<tr>
<td>4096 Point FFT</td>
<td>N/A</td>
<td>34 µs</td>
<td>41 µs</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Point FFT</td>
<td>4.7 µs</td>
<td>9.4 µs</td>
<td>19 µs</td>
</tr>
<tr>
<td>512 Point FFT</td>
<td>23 µs</td>
<td>51 µs</td>
<td>119 µs</td>
</tr>
<tr>
<td>4096 Point FFT</td>
<td>178 µs</td>
<td>387 µs</td>
<td>910 µs</td>
</tr>
</tbody>
</table>

### 4.3 Summary

Overall, experiment 2 shows that using our CRASH heterogeneous computing framework can improve processing performance by 2x over the software only processing in experiment 3. However, we increase latency by 2-3x versus FPGA only processing in experiment 1. For the loss in performance, we gain the flexibility and ease of programming algorithms for a CPU versus a FPGA.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis we introduce CRASH, a versatile heterogeneous computing framework for the Xilinx Zynq SoC. Our experiments show that CRASH can successfully segment two cognitive radio algorithms, spectrum sensing and the spectrum decision, between the Zynq’s FPGA fabric and ARM processors with up to a 2x increase in processing performance versus a purely software approach. More importantly, MAC layer timing specified in the 802.11 specification can be met while providing the designer with flexibility in implementing important processing components.

While the CRASH framework has been demonstrated on cognitive radio, it is generic enough to be used in other applications requiring a mix of FPGA acceleration and CPU processing. For instance, real time avalanche detection requires sampling a seismic sensor, filtering the signal, and feeding the reduced data to a machine learning algorithm [25]. In this case, the radio interface blocks could be replaced with blocks interfacing to the sensor. The filtering and FPGA-CPU data transfer blocks could
remain in place. CRASH supports this and other applications with a platform for researchers to quickly program and deploy new detection algorithms on the ARM processor while maintaining high performance processing in the FPGA.

5.2 Future Work

In the future, we are interested in studying the effects of more sophisticated spectrum decision algorithms on latency. We also plan to exploit the platform’s low latency processing capabilities to study cognitive radio algorithms when operating with other 802.11 wireless devices. Finally, we want to expand the platform to support easily moving processing blocks between software and reconfigurable hardware, possibly incorporating run-time reconfiguration, and interface with other software packages such as Mathworks’ Simulink.
Appendix A

List of Acronyms

ADC Analog to Digital Converter
ACP Accelerator Coherency Port
AMBA Advanced Microcontroller Bus Architecture
API Application Programming Interface
AXI Advanced eXtensible Interface
CPU Central Processing Unit
CRASH Cognitive Radio Accelerated with Software and Hardware
CRUSH Cognitive Radio Universal Software Hardware
DAC Digital to Analog Converter
DDR Double Data Rate
DMA Direct Memory Access
DRAM Dynamic Random Access Memory
FIFO First In First Out
FFT Fast Fourier Transform
**FFTW** Fastest Fourier Transform in the West

**FM** Frequency Modulation

**FMC** FPGA Mezzanine Card

**FPGA** Field Programmable Gate Array

**GPU** Graphical Processing Unit

**GUI** Graphical User Interface

**HDL** Hardware Description Language

**MAC** Media Access Control

**MICTOR** Matched Impedance Connector

**RF** Radio Frequency

**SDR** Software Defined Radio

**SIMD** Single Instruction Multiple Data

**SoC** System-on-Chip

**UART** Universal Asynchronous Receiver Transmitter

**USRP** Universal Software Radio Peripheral

**UHD** Universal Hardware Driver

**WARP** Wireless Open Access Research Platform
Bibliography


