An Architecture Study on a Xilinx Zynq Cluster with Software Defined Radio Applications

Christopher V. Dobson

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

Peter Athanas, Chair
Robert McGwier
Patrick Schaumont

June 16, 2014
Blacksburg, VA

Keywords: Zynq, tFlow, Software Defined Radio, Cognitive Radio

An Architecture Study on a Xilinx Zynq Cluster with Software Defined Radio Applications

Christopher V. Dobson

(ABSTRACT)

The rapid rise in computational performance offered by computer systems has greatly increased the number of practical software defined radio applications. The addition of FPGAs to these flexible systems has resulted in platforms that can address a multitude of applications with performance levels that were once only known to ASICs. This work presents an embedded heterogeneous scalable cluster platform with software defined radio applications. The Xilinx Zynq chip provides a hybrid platform consisting of an embedded ARM general-purpose processing core and a low-power FPGA. The ARM core provides all of the benefits and ease of use common to modern high-level software languages while the FPGA segment offers high performance for computationally intensive components of the application. Four of these chips were combined in a scalable cluster and a task assigner was written to automatically place data flows across the FPGAs and ARM cores. The rapid reconfiguration software tFlow was used to dynamically build arbitrary FPGA images out of a library of pre-built modules.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Software Defined Radio (SDR) provides a mechanism through which application-specific radio systems can be implemented on flexible application agnostic platforms. The rapidly increasing level of performance supplied by software platforms has increased the number of applications realizable by this type of system. The addition of rapidly reconfigurable FPGA processing components has supplied performance levels that used to only be attainable by ASICs or fully analog circuits. Embedded systems are an ideal candidate for SDRs due to their low power requirements and portability. The reliability and consistent performance that is a top priority for communications application is also offered by embedded systems.

Cognitive radios are part of an application domain that became more powerful and flexible through the use of software defined radios. With a traditional application-specific platform, cognitive radios are limited to simple tasks like spectrum sensing, where only parameters to hardware components can be changed. Software defined radio enables cognitive systems to radically change configurations and roles depending on a variety of stimuli. These types of systems are especially important in situations where there are finite resources or when the desired behavior is not necessarily known at deployment time.

This work presents a scalable cluster of embedded Xilinx Zynq processors as a platform for software defined radio applications. The hybrid Zynq chip provides both an embedded ARM

general-purpose processing core as well as an interconnected FPGA. These two components provide a combination of ease of programmability and high performance that can be used to realize radios. The cluster possesses the ability to automatically assign radio flows consisting of software and FPGA modules across multiple Zynq boards and form all the necessary interconnections. The FPGAs can be reconfigured to provide any combination of pre-designed components from a library of FPGA modules. This enables the unparalleled ability to build arbitrary software defined radios that make efficient use of FPGA accelerators across a cluster.

1.1 Motivation and Contribution

The motivation for this work came from a project with similar cognitive radio goals but a different hardware architecture. A 64-core Tilera TilePro64 running GNU Radio was the processing core of the system, and a large Xilinx Virtex 6 was used to attach the TilePro64 to the RF front end. The TilePro64 was selected due to the desire to run multiple GNU Radio flowgraphs on one system. The highly parallel and threaded nature of each of GNU Radio flowgraphs made having 64 independent cores to spread the load across seem like an ideal solution at the time, but the low performance offered by each core and communications choke points created bottlenecks that limited the total performance of the system. The Virtex 6 contained some front end processing that all software radios needed, but it was not reconfigurable and all application specific processing had to be done in the TilePro. Section 4 contains more details about this system and how it motivated a platform with better FPGA utilization and more powerful software cores. It also details how these problems are addressed by this work.

The platform presented in this work provides a hardware base for future works in software defined radio. Future research can utilize this platform to implement radio applications without concerns about the specific hardware details. Contributions include:

- A novel scalable embedded cluster, specialized to data flow applications was developed. This provides a transparent target for future research in the areas of software defined radio and cognitive radio.
- A dataflow task assigner was written to automatically divide radio flows across the cluster. This enables the use of a scalable cluster without any consideration from radio designers.
- The first embedded use of tFlow for rapid reconfiguration was performed in this work. The rapid reconfiguration and bit file generation provided by tFlow enables embedded applications to make more efficient use of the FPGA.
- An efficient high speed Zynq DMA engine was developed to provide communication between software running in Linux on the ARM core and FPGA accelerated components. The standard interfaces provided by this system enable easy integration of FPGA acceleration into any Linux application.

1.2 Document Structure

This document begins with an overview of works that have already been performed in related areas. Four such works are presented and explained. It then moves on to describe the Zynq cluster platform in deal, including hardware and software components. An explanation of how all the components fit together is given at the end of this chapter. The chapter detailing the motivating platform and how it was improved upon is presented after that. The next chapter

presents a cognitive FM demodulation application of the platform. A comparison to related works is located after that, which is followed by a conclusion with future work.

Chapter 2
Related Works

Several recent endeavors have already been performed in the areas of reconfigurable FPGA accelerated software defined radios and Zynq based computing. A few have been selected that share at least some key aspects with this work and are presented in this chapter. The first work is a cluster of Zynq boards, but no dynamic reconfiguration is present. The second and third works utilize dynamic reconfiguration in software defined radio applications, but do not support building of arbitrary radio flows. The fourth work provides the ability to build arbitrary FPGA accelerated radios, but relies on an external host for any software processing.

2.1 ZCluster: A Zynq-based Hadoop Cluster

Lin and Chow [1] created a cluster of Zynq ZC7020 boards that use the Hadoop framework to distribute MapReduce applications across the cluster. The ARM cores run Xillinux, which is an Ubuntu 12.04 based operating system released by Xillybus. The proprietary Xillybus Zynq DMA IP core was used to enable transfers between hardware and software. The maximum observed synchronous two way transfer for this bus was 103MB/s. Unidirectional reads of 186MB/s and writes of 162MB/s were observed as well.

The FPGA acceleration consisted of a single bit file that had to be loaded before the processor even started. The sample application in this bit file was a 51 tap fixed-point FIR filter to perform

low-pass filtering which operated at 100 MHz. The benchmark that was run consisted of reading data encoded in ASCII and piping it through the FPGA. They found a 2.4-fold speedup from using the hardware acceleration when compared to a software FIR.

The results from the benchmark indicate that the FPGA is being highly underutilized. Their test consisted of 4 million samples and took 5 seconds when using the FPGA. At 100 MHz, the FPGA should be able to complete filtering in less than 40 milliseconds. This indicates that there is a severe bottleneck somewhere else in their system. Likely causes of this extreme slowdown are the inefficient file format or the fact that data is read from *standard in* instead of standard file I/O. It was hinted that the input file was cached in RAM when the test was run but it is possible that it was stored on an SD card or had to be fetched from 100 MB Ethernet, which could have also negatively impacted performance.

A series of benchmarks were then run using Hadoop to actually distribute the load. They were able to distribute the FIR across a cluster of eight boards. There were some strange results caused by the overhead of the Hadoop protocol. They found that if an input data set is too small, Hadoop can actually increase a benchmark's run time. A file that would have taken about 8 seconds to run on a single board without Hadoop took 108 seconds when spread across the cluster. Increasing the input file size by a factor of 10 presented Hadoop with an easier job and the total execution time was 63 seconds.

2.2 Accelerating Software Radio: Iris on the Zynq SoC

Van de Belt, Sutton, and Doyle [2] provided hardware acceleration for the Iris software defined radio on the Zynq platform. The ARM core runs Xilinx Linux, which provides a basic Linux

environment including GCC, GDB, and binutils. All additional software was cross compiled on a host instead of performing native builds on the ARM. The Iris software radio framework as well as its various dependencies, such as Cmake and Boost, were some of the cross compiled software packages. Iris is an open-source software defined radio framework that provides a series of components that can be connected together to form radios. An XML descriptor file is used to describe all of the desired radio's components and parameters.

Programmable logic acceleration was added to Iris as a component that could just be instantiated in the XML like any other. The profiling tool Perf was used to discover the most intensive sections of the radio, and those sections were replaced with pre-existing hardware accelerators. The system was evaluated by modulating a 800 kbps video stream using OFDM on one ZC702 board, transmitting and receiving it with a pair of USRPs, and demodulating it with a second ZC702 board. Performing both modulation and demodulation on one board only utilized 41% of the ARM system which implies there is plenty of headroom for more complicated radio designs.

2.3 Autonomous System on a Chip Adaptation Through Partial Runtime Reconfiguration

French, Anderson, and Kang [3] had developed a cognitive radio system on a Virtex 4 FX100 that utilizes Xilinx's partial reconfiguration. A PowerPC processor was implemented in the FPGA to provide both a location to run sequential cognitive decision making, as well as control over the programmable logic and its reconfiguration. The PowerPC core ran Linux as its operation system to provide some abstraction from hardware, but it doesn't actually perform any signal processing. Because it was only used to run the cognitive engine and control the reconfiguration, there was no need to build any software defined radio packages and their dependencies.

The FPGA was broken up into a static region consisting of the PowerPC core with a communications decoder and a reconfigurable region. Xilinx's partial reconfiguration flow enables reprogramming of the reconfigurable region, but only with partial bit files generated by its desktop flow. This region was used to perform one of three mitigating filtering techniques before passing the data stream into the decoder located in the static region. The cognitive engine was able to determine which filter was optimal based on an analysis of the spectrum. The cognitive engine was able to detect 97% of changes in the environment, and provide at least 3dB in interference rejection 80.6% of those times.

2.4 A FPGA Partial Reconfiguration Design Approach for Cognitive Radio Based on a NoC Architecture

Delorme, Martin, and Nafkha [4] have implemented a cognitive radio using a pair of Xilinx Virtex 4 LX100s and a network on chip (NoC) architecture. A Microblaze located in a static region of the FPGA was used to manage the reconfiguration. Xilinx partial reconfiguration was used much like the previous work. Instead of a single reconfigurable region, a series of much smaller reconfigurable regions were created that are all connected to a NoC. The final evaluation involved transmitting a video stream from a host PC through one of the LX100s which modulated the data and sent it to the second LX100 which demodulated the data and sent it back to the PC for viewing.

The benefit afforded by the NoC was the ability to build arbitrary radio systems that can have varying numbers of any module in a library. This enabled the building of multiple concurrent radios, whereas the more traditional single reconfigurable region limited the system to a single

radio at a time from a set of pre-built radios. A major downfall was the size in which each of these partial reconfiguration regions was set. Modules would inevitably be too large or too small to fill the entire region and be wasteful. Another problem is that the Xilinx partial reconfiguration flow requires every module in the library to be built for every node in the network that it can be placed in, even if the nodes are identical in the FPGA fabric. This only affects compile time and not run time, so it is more of a nuisance for development than final system performance.

Chapter 3
Platform Overview

The various components of the system are examined in this chapter. It starts by focusing on the hardware aspects such as the Zynq itself and its programmable logic components. It then moves on to more of a software perspective, covering components such as GNU Radio and tFlow.

3.1 Zynq ARM

Each ZC702 board contains a single XC7Z020 Zynq chip with a fairly standard dual Cortex-A9 ARM processor core operating at 866 MHz [5]. As can be seen in Figure 3.1, there are a variety of onboard interface controllers attached directly to the ARM core such as a DDR3 memory controller, a pair of gigabit Ethernet interfaces, and two USB 2.0 buses. Each core has its own 32 KB instruction cache and its own 32 KB data cache, with both cores sharing a 512 KB L2 cache. The optional VFPv3 and NEON extensions are both present in the Zynq ARM implementation, which enable high-performance floating-point operations and SIMD extensions respectively. The ARM architecture and instruction set has been receiving a large amount of attention lately because of its widespread use that ranges from micro desktop computers to embedded control systems. This has resulted in greater and greater support for the ARM ISA in open-source projects, with several mainstream Linux distributions supporting the architecture.

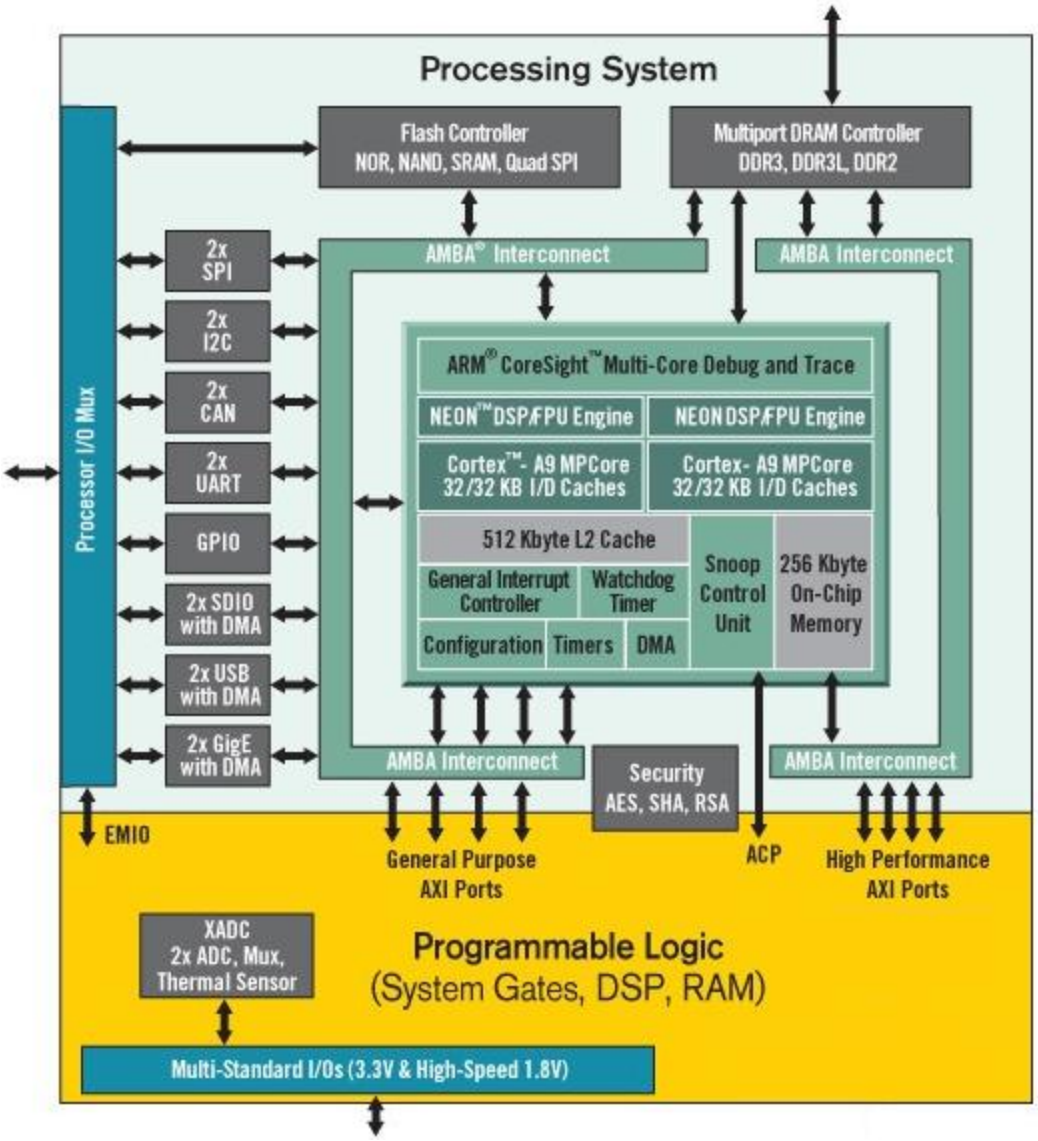

Figure 3.1 : Zynq System Diagram http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm. [Accessed: 07-Jun-2013]. Used under fair use, 2014. [6]

3.2 Zynq PL

The programmable logic present in the XC7Z020 is based on the low-power and low-cost Atrix-7 family of FPGAs. While the performance is also less than Xilinx's other FPGA families, power and cost are both important in embedded applications as well as clusters. The PL contains 53,200 look-up tables, 560 KB of block ram, and 220 DSP slices which puts it in the middle of

the middle of the Zynq lineup capacity-wise [5]. There are also eight AXI links between the programmable logic and the ARM core. Four are general purpose low speed ports that are mapped directly into the ARM's memory, while the other four are high-performance ports that give the programmable logic direct access to the off-chip DDR3 SDRAM. There are also 16 interrupts that tie directly into the ARM core for high speed or timing critical applications.

3.3 DMA

Figure 3.3 contains a block diagram of the DMA system that provides an efficient multi-channel high-speed interface between the ARM core and the PL. Each DMA channel consists of a bidirectional link that utilizes one high performance AXI port, one general purpose AXI port, and two interrupts. Xilinx IP was used that manages the general purpose AXI interface to provide memory-mapped registers. A different Xilinx IP was used that manages the high-performance AXI interface and provides an easier to use IPIC interface. A Linux driver was written that provides a char driver interface for simple use, and an easy control interface for user-space applications. From the programmable logic side, the interface is a FIFO streaming interface.

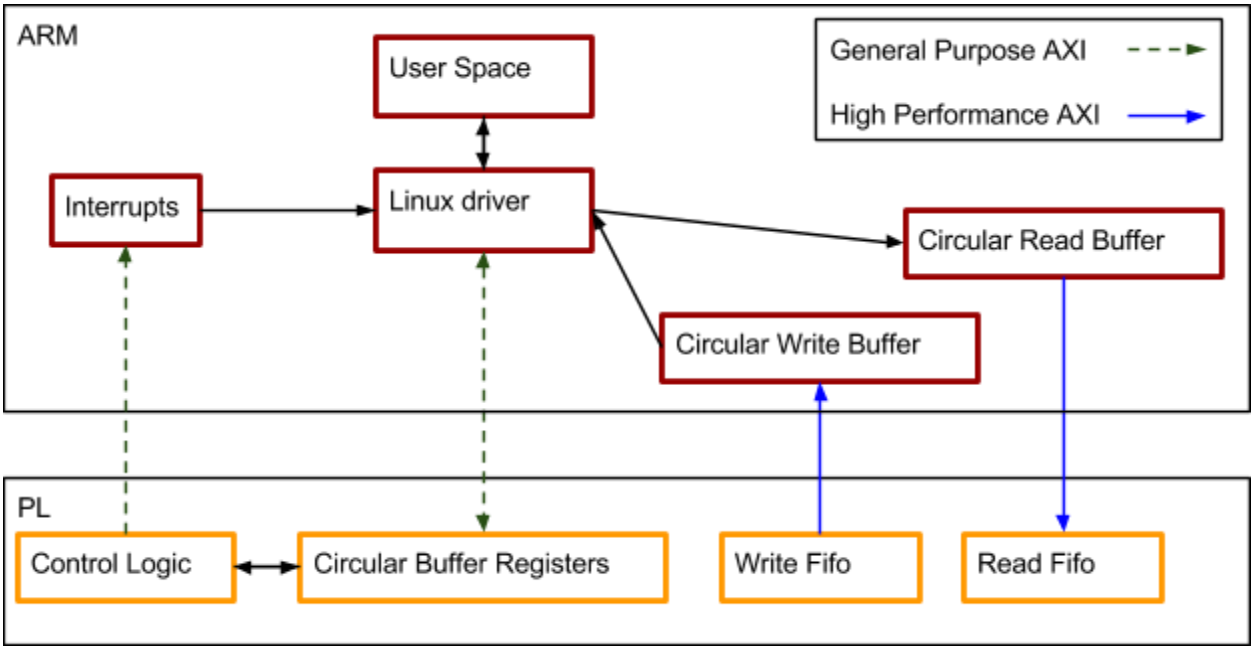

Figure 3.3 : DMA System Block Diagram - Single Channel

The DMA system maintains a pair of circular buffers that are parameterizable in size for each of the DMA channels, one buffer for each direction of data transfer. A series of control registers located in the programmable logic are mapped into the ARM's address space using the general purpose AXI interconnect. The ARM tells the programmable logic the physical addresses of the read and write circular buffers as well as how large they are through this interface. The programmable logic keeps track of how much data there is for the ARM to read, and how much empty space there is to write to. Whenever the ARM reads and writes to the circular buffers, it notifies the programmable logic through the general purpose AXI interface.

The software interface consists of one control char interface and one data transfer char interface for each of the DMA channels. Writing to the control interface can enable, disable, or reset the DMA channels. Writing to the data transfer channels first checks if there is enough room in the

circular buffer, copies to the circular buffer, and notifies the programmable logic about the copy that was just made. If there is not enough room in the buffer to complete the transfer, the driver copies as much data as it can, and notifies the programmable logic to send an interrupt when there is enough room. To prevent unnecessary and inefficient small transfers, interrupts are only triggered when there is either enough room for the entire write or one sixteenth of the buffer is empty, whichever is smaller. Reading from the data transfer interface performs the same series of operations, but it checks for valid data instead of empty space.

The programmable logic is the master of the high performance AXI bus, so it is in charge of actually performing all the reads and writes. Three factors need to be considered when the PL is performing a write operation to memory: the amount of empty space in the circular buffer, the amount of space until the end of the circular buffer, and the amount of data in the FIFO waiting to be written. The smallest of these three numbers is chosen and a write can begin. A similar series of comparisons can be performed for the read operation, but valid data figures are compared.

A few benchmarks were performed to determine maximum transfer rates through the DMA system. These results are summarized in Table 3.3 with results in megabytes per second. Loopback tests were performed where the read and write channels were attached to each other in the FPGA, as well as one way tests. Each test also took two forms, one where the write program generated a counting sequence and the read program check for this counting sequence, and the other where the write program generated nothing and wrote trash while the read program checked for nothing. The baseline performance for dual way transfer was 100 MB/s each way.

Removing the data generation and checking increased throughput by 50%, showing the effect of the compute time used in user space and an extra memory operation. The one way transfers reveal that writing to the FPGA is faster than reading from it.

Transfer Type	DMA Benchmark	Normalized mbw result
Dual way checking	100 MB/s	108 MB/s
One way to FPGA checking	211 MB/s	216 MB/s
One way from FPGA checking	184 MB/s	216 MB/s
Dual way not checking	155 MB/s	144 MB/s
One way to FPGA not checking	367 MB/s	289 MB/s
One way from FPGA not checking	223 MB/s	289 MB/s

Table 3.3: DMA Benchmark Results

A benchmark called *mbw* was run that performed a large copy from one memory location to another that achieved a data rate of 433 MB/s. This effectively is a single read and a single write. The *Dual Way* test consists of four reads and four writes, and the one-way test that consists of four total operations. The normalized benchmark result shows the *mbw* result normalized to the number of copies performed. Considering that the DMA system often performs near or even beats the *mbw* result, it is likely that memory bandwidth is the limiting factor in performance. Changing the driver model to map the circular buffer into user space could remove one read and one write from all operations and increase performance, but the result would be a more difficult interface to use compared to the current file based driver.

3.4 Cluster

A cluster was built by using four separate ZC702 development boards with two parallel methods of interconnectivity. The ARM cores are connected to each other using a gigabit Ethernet switch. This provides an easy method to connect the ARM cores directly to each other at a reasonable bandwidth. The main goal of this link is to provide an interconnect for control purposes and low speed data transfer.

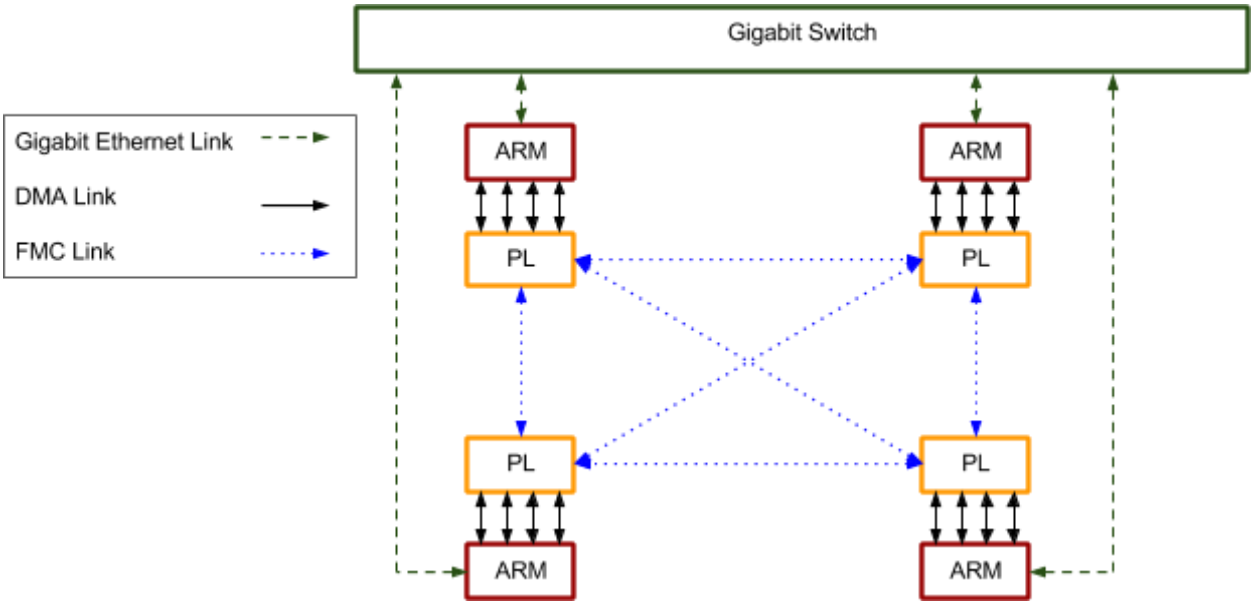

Figure 3.4 Cluster Connectivity

The programmable logic components of the chips are connected by an independent series of links. A breakout board design was found that converts one of the available FMC connectors into 17 SATA connectors. It is possible to create eight one-way links by using two individual SATA cables per link, with each cable containing two differential pairs. Every link has a 32-bit parallel interface attached to a FIFO that is clocked at 10 MHz, while data is moved across the SATA cables at 100 MHz. This provides a speed of 320 Mbps per link. The FMC interconnects

were arranged in a manner that provided a single bidirectional link between every board in the cluster. This fully connected architecture provides the most flexibility for an adaptive system.

Each FMC link is centered around a pair of Xilinx serializer deserializer IP cores. They are peripheral hardware components in the FPGA designed to support inter-chip communication. The serializer core has a parallel input and a serial output that directly drives an output pin, and the deserializer core has just the opposite. There is no protocol built into these components to ensure that the two agree on where to start a word, they just start operating when brought out of reset. There is a capability on the receive side to force it to shift its starting bit by one position. In addition, the FMC breakout board has several channels with inversions between the FPGA's LVDS differential pairs and the SATA twisted pairs. These problems were solved by instituting a training sequence that is automatically transmitted for five seconds after reset, or at the command of the ARM processor. The receiver shifts through all possible combinations of bit shifts and inversion until the known training sequence is found. With a total of about 160 different combinations, the right combination is found very quickly at 10 MHz.

3.5 Linaro Ubuntu

Linaro is a not-for-profit company dedicated to ARM development of a variety of software ranging from the Linux Kernel to the GNU Compiler Collection [7]. In addition to these efforts, they also maintain an Ubuntu based Linux distribution that includes their ARM specific development efforts before they even get pulled into mainstream development lines. This was chosen as the as the operating system for the ARM platform because it provided all of the community support of Ubuntu and the ARM specialization that increases performance. The

kernel was built from source provided by Xilinx that included necessary patches for the Zynq platform in particular.

The biggest advantage provided by using Linaro was the repository of pre-built packages. The standard *apt-get* and *aptitude* tools can be used to painlessly install packages already build for the ARM architecture. While GNU Radio was still built manually because its repository version is out of date, all of its dependencies could just be installed with *apt-get*. While there are only about five non-standard dependencies needed to get the C++ interface working, a multitude more are needed to get the convenient and useful GNU Radio Companion tool working. Being able to *apt-get install* the five utilities needed for the C++ interface was definitely a huge time saver compared to building them, but attempting to get GNU Radio companion to run without the repository would have taken more time that it was worth.

Due to limited options, it was decided to host the root file systems for the boards on an NFS server. Most of the guides and reference designs for this board host the file system on an SD card, and a USB 2.0 port is also available. The SD card was the first option tried, but it suffered from exceptionally low performance. The lack of wear leveling in most SD cards is also a concern when hosting a read write file system on a system undergoing active development. The USB option was ultimately dismissed due to the fact that the Zed boards used in development had a potential timing issue with their USB interfaces. The NFS option was hosted on an SSD over the gigabit network and provided a significantly faster system. An additional benefit to the NFS mount was that the folder containing builds for GNU Radio and tFlow could be shared between boards to ensure they were all up to date and synchronized. This also saved time

because tFlow takes over 10 GB in storage all on its own, and takes several hours to build. A migration back to SD cards could easily be done with relative ease to make the system stand-alone.

Even with the NFS file system, the SD card was still a necessary part of the boot process. At the very minimum, the bootloader needs to be stored on the SD card. The SPI flash likely could have been used, but the SD card provided an easier solution. At first the kernel and device tree were being loaded over NFS to minimize manually changing files on the SD cards, but the bootloader occasionally failed to fetch the files and hung. The solution was to just keep these two files on the SD card with the bootloader.

3.6 GNU Radio

GNU Radio is a free and open-source development kit for the implementation of software defined radios [8]. The software radios are modeled by flowgraphs that represent the movement of data through the components of the radio. Flowgraphs are built by connecting blocks, which are the basic processing units of GNU Radio. Each block's function may vary from something as simple as an adder to something as complex as an entire digital receiver. Blocks have a number of input and output ports that are connected to ports of the opposite type to form radios. When a flowgraph is run, each software block is started in its own thread. This enables blocks to operate concurrently with each other. Input and output ports are connected to a circular buffer that is shared with the data producer and one or more data consumers.

Each block can also have parameters that extend its functionality, with many blocks having parameters that can be changed during operation. The parameters can just be simple variables such as the center frequency or they can make a block more flexible by changing data types and even the number of input and output ports. Using parameters, it is possible to create an adder that is capable of adding any feasible number of inputs to a single output of any data type that supports addition. This is the type of adaptable design that is incorporated into many of GNU Radio's built-in blocks to create a truly flexible tool kit.

A standard GNU Radio install includes hundreds of pre-designed blocks that are ready to use and sufficient for designing many radios. There are a variety of interpolating or decimating filters that can accept either frequency bands as parameters, or user provided taps for additional fine grained control. There are built-in modulators and demodulators for both analog modulation schemes such as AM and FM as well as digital modulation schemes such as FSK, PSK, and QAM. The one area that often requires the design of a new block comes in the form of protocol parsing. This can also be accomplished by piping the demodulated data out of GNU Radio and dealing with the data there.

GNU Radio has an API to build and run flowgraphs from both C++ and Python. While the C++ interface boasts superior flowgraph startup time, the Python interface has a few advantages of its own. The first is the flexibility to easily build a new flowgraph without having use a compiler, which is an especially costly operation on an embedded platform. The second is that there are certain blocks that are only available in Python, such as TCP communication blocks, that make use of standard Python libraries. Once a flowgraph is running, the performance of the two

interfaces is approximately the same. The Python interface actually uses SWIG generated bindings to call and run the same individual C++ blocks.

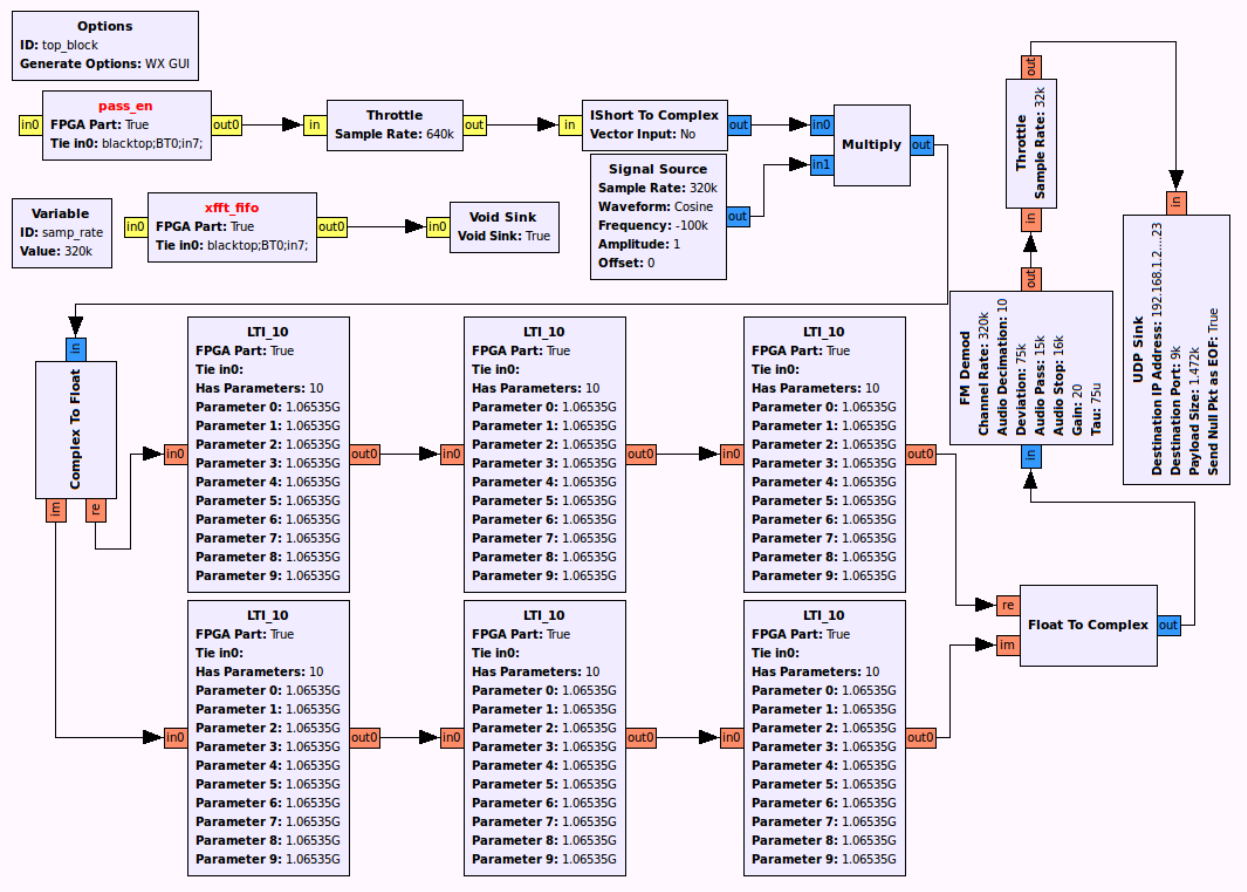

Figure 3.6 : GNU Radio Companion

GNU Radio Companion is a graphical tool that can be used to easily construct flowgraphs by just dragging and dropping blocks onto a canvas and drawing connections between them [9]. Figure 3.6 is a screenshot of a radio in Companion. One of its most useful purposes is in rapid prototyping of radio designs. Companion generates and runs a Python script that contains all the blocks on the canvas. To help in a quick design, Companion includes a number of visual data analyzers such as time domain views, FFT displays, and waterfall plots. There is also support

for sliders and text boxes that can be used to change the parameters of running flowgraphs from a GUI interface for easy experimentation.

The flexible nature of GNU Radio enabled the relatively painless addition of FPGA blocks to accelerate flowgraphs. Hardware acceleration blocks were added into GNU Radio companions library so that they could be used just as easily as software blocks. This enables radio designers to utilize hardware modules without having a background in digital design or FPGAs. Additional support was needed to provide proper interfaces between these two processing domains, which is discussed in greater detail in section 3.11 on the task assigner below.

3.7 tFlow

tFlow is a software toolset that enables the rapid reconfiguration of FPGAs by placing pre-made modules in an empty region of the FPGA [10]. First, FPGA modules are designed in a hardware description language just like any normal FPGA design flow. These modules then need to be implemented in a constrained area. There is a tool in tFlow called the *shaper* that automatically determines the amount of area that a block needs and a place in the FPGA, but complicated designs often need this to be determined manually by the designer. With the area constraints determined and set in a constraint file, the standard Xilinx flow can be used. To make modules more flexible, they are often designed with parameter registers that can be set at runtime.

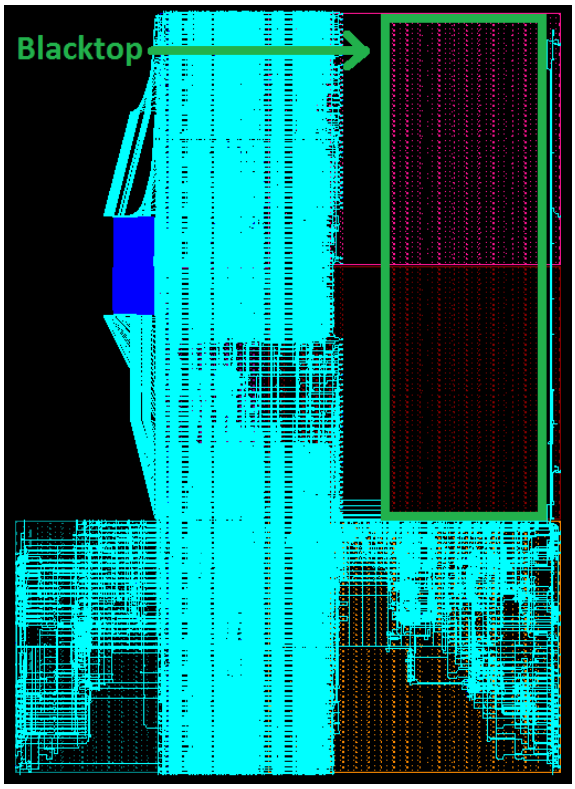

Figure 3.7.1 : Static Image with Blacktop

An FPGA image called a *static*, which can be seen in Figure 3.7.1, is created that contains a special empty reserved region called a *blacktop*. The static also contains necessary logic to connect data ports to and from the blacktop to some useful interface. The static is created using the standard Xilinx toolset with a few tweaks to the normal flow. The first requirement is to prevent the tools from placing any modules in the blacktop, which can be done fairly easily with a few lines in the constraint file. The second requirement is to prevent the tools from using any of the routing resources in the blacktop, which cannot be simply done with the constraint file. The solution used here is to place a blocker that occupies all the routing resources into and out of the blacktop region before the routing stage, thereby preventing the Xilinx tools from using it. The normal Xilinx routing stage then takes place, and the blocker can be removed after that. In

addition, the ports into and out of the blacktop have to be manually placed, preferably in an uncongested area near the blacktop region to prevent routing conflicts later on.

tFlow is instructed which modules to use and how to connect them using a special file called an *EDIF*. With the exception of clocks that are specially detected and treated for performance reasons, all other connections are treated the same. This gives it the ability to connect ports of any size that can contain any protocol. To help with design and compatibility, it is simplest if a standard port size is used. A port width of 32 bits with an additional valid bit was chosen because it supports a single floating point sample, or a complex 16 bit integer sample, which make up most flowgraphs. Complex floating point samples can just alternate between real and imaginary data.

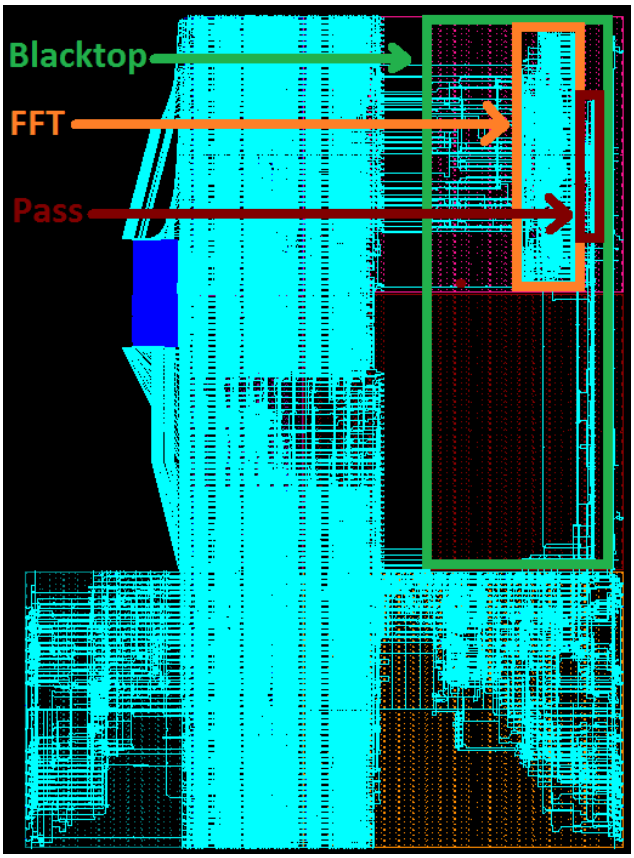

Figure 3.7.2 : Blacktop with FFT and Pass Through Modules

The static generated for the XC7Z020 did require some careful planning to maximize the blacktop region. The static contains four bidirectional FMC channels and four bidirectional DMA channels. The four FMC channels enable the cluster to be fully connected with one channel left over to receive data as an input. The four DMA channels ensure that there is plenty of connectivity between the ARM and the blacktop. All in all, this creates eight ports into and out of the blacktop. There is an additional restriction on modules that requires them to be exactly one clock region tall, and the XC7Z020 is a three clock region tall device. Due to the fact that the FMC pins are located on both sides of the FPGA, and the blacktop cannot have any routing inside of it, the blacktop is limited to being two clock regions tall or else it would cut the chip in half. The location of the ARM DMA ports and the supporting logic then limits the blacktop's

width to about third of the chip. The resulting blacktop is large enough to contain about four placement locations for small to medium sized modules, and two placement locations for larger modules.

tFlow takes approximately 60 to 90 seconds to generate a bit file from an EDIF. To mitigate this run time, a caching system was put in place that saves previous outputs of tFlow so they can be rapidly reused. The outputs are cataloged by combining an md5sum of the entire EDIF input file and the filename. While this doesn't detect EDIF files that are functionally identical but superficially different, it is sufficient that any radio that is rerun can skip tFlow all together.

3.8 Block Parameterization

Just like software blocks, it is important that hardware blocks can be parameterized as well. This however presents more of a challenge than software blocks because these values cannot just be fed into tFlow's EDIF file like the software variables can be included in the Python flowgraph. This required the creation of an interface that is expandable to large numbers of registers, but easy for tFlow to hook up.

Addressing individual parameter registers using a traditional addressed bus is not optimal for a number of reasons. First off, that would require routing address and data pins to every block in the blacktop. Fan-out and routing concerns could become an issue for large numbers of blocks, and tFlow doesn't need its job made any harder. In addition, there is no easy way to assign the modules addresses to listen on. Assigning hard addresses to each module doesn't work because there are often duplicates that would be listening on the same addresses. The solution that solved

both of these problems was a single wire serial interface that used daisy chaining to deal with the addressing problem. Blocks with multiple parameterizable registers can just include the same daisy chained design within the HDL module.

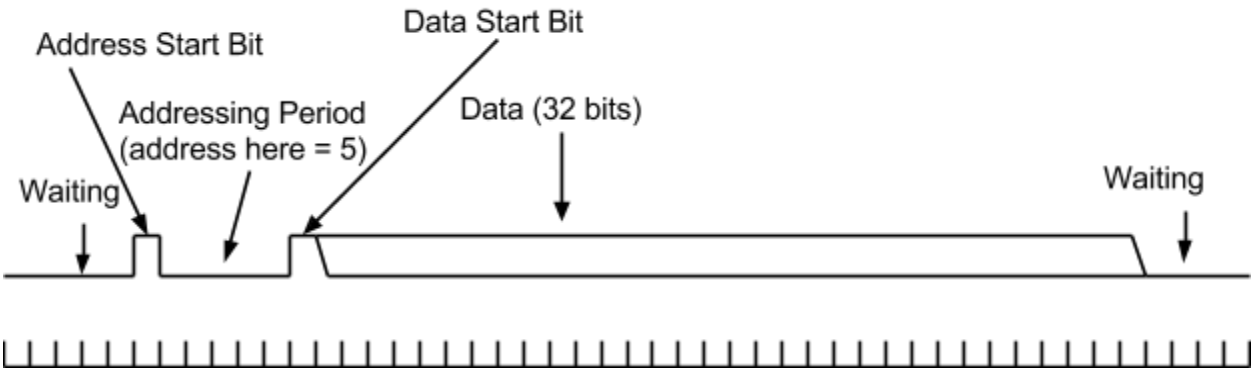

Figure 3.8 : Parameter Protocol

Figure 3.8 shows a timing diagram of the protocol, with the address being five in this case. The protocol, like most serial protocols, begins with a start bit which indicates the start of a 32-bit data word. The bus is held low when there is no data, with the start bit being high. After the start bit, the addressing segment of the signal occurs. Registers are addressed by their position in the chain. While a number corresponding to a module's position was first considered, it was rejected because modules would either have to be trained or decrement the number. Both of these operations would require each register to have an addition adder, which is inefficient in space for modules with many registers.

Instead, for each register to be skipped, the protocol just holds the data line low for one clock cycle after the start bit. The register strips off this low bit, and forwards the rest of the data stream on. The speed penalty paid in cases with large numbers of registers is a worthy trade off

for the area savings. Due to the fact that parameters are not rapidly changing and the clock operates in the mega-Hertz range, the time it takes to set a register at the end of the chain is still small. After the signal has been held low the desired number of cycles, the signal strobes high to signify the end of the addressing segment of the signal. The next 32 clock cycles contain the data word itself.

In addition to the register component that must be instantiated for every parameterizable module, a controller was written to interface between software and the serial interface. The controller is attached to the ARM system through one of the general-purpose AXI interfaces. The controller's inputs are a 32-bit data word, a 31-bit address, and a single start bit. The controller detects a rising edge on the start bit and uses that to signify the beginning of a transaction. These input signals add up to a pair of memory-mapped registers, so setting a parameter register just takes three writes from the ARM: one to clear the start bit, one to set the data, and one to set the address and start bit.

3.9 Cognitive Engine

The cognitive engine is a framework for the development of reactive radio systems. The first stage consists of an energy detector that is attached to a FFT located in the programmable logic. The hardware FFT is a 1024 point Xilinx Coregen part that operates on 16-bit complex samples. The Coregen part is then wrapped in a tFlow module and placed in the blacktop. Using core affinity commands it is possible to ensure that the FFT is always present on a specific core and connected to the data source as an input. The energy detector can read the transformed data in

through a DMA channel. The energy detector first looks for peaks above the noise floor, and then determines each peak's bandwidth.

Format:
Center Frequency, Bandwidth, Priority, Resource utilization, Name, GRC file

Sample CSV File:

```
0,1000,2,1.0,fm0,../grc/fm0.grc
100000,1000,1,1.0,fm1,../grc/fm1.grc
```

Figure 3.9 : CSV format and Sample CSV file

The cognitive engine then takes these frequencies and bandwidths and compares them to a database of signals. The database of signals is stored in a CSV file that contains the center frequency, bandwidth, priority, resource utilization, and a GRC flowgraph. Figure 3.9 shows the file format and a sample file containing two signals. First, all of the detected peaks are compared to the database, and all matches are recorded. If there are multiple matches, the cognitive engine can be configured by the database to either demodulate just one flowgraph, or start multiple flowgraphs. The resource utilization parameter is used to ensure that too many flowgraphs are not all started at once. If too many signals are detected to all be demodulated concurrently, then the priority parameter is used to determine which flowgraphs are the most import and should be included first.

3.10 GRC Parsing

To facilitate easy radio design, it was decided that the flowgraphs read in by the cognitive engine would be stored in the GNU Radio Companion's XML GRC file format. First, all of the FPGA accelerators were added to the companion library of blocks by creating a special XML file for

each. This enabled both hardware and software blocks to be available in the simple drag and drop design environment. The GRC file itself included an entry for every block in the flowgraph, and every connection.

The XML file associated with each software block contains all of the information needed to generate the necessary Python scripts. The name of the module in Python as well as the location of its arguments in the constructor are present in a "make string" field. The arguments themselves are stored in the GRC file. The XML also contains the Python import statements that are needed for each block.

The nature of FPGA modules, however, requires a different set of information. The first addition was a FPGA module flag that lets the XML parser know that it isn't a software block. This prevents it from looking for information that isn't present, such as the make string or import statements. Beyond this flag, all of the XML files contain the name of the module used by tFlow, and the number of ports. In addition, a field was added for each parameterizable register located within the module. These values are taken and sent to the boards where the module is located. A field used to tie a block's inputs to blacktop ports is also present in GRC.

There are a few minor issues that require special attention. GRC uses a "sizeof" string to represent the size of data types and passes these as arguments to certain blocks. The "sizeof" strings for complex float and bytes, however, are not defined in Python, while all the other "sizeof" strings are. This requires searching for these strings and replacing them with a different string. All members of the filter class also require a special prefix to be attached to the name of

the block when searching for the XML file associated with them, but the blocks are not marked as filters in the GRC file. The solution is to just tack the prefix onto the name of the block if finding the XML file fails the first time. There is also a minor issue where having certain GNU Radio Python import statements in the wrong order causes a memory leak that always results in an eventual operating system segmentation fault.

All of the blocks with their parameters are placed in one vector and all of the connections are placed in a second vector. Both of these vectors are then passed onto the task assigner. A wrapper was written that takes these two vectors and puts them in the format that the task assigner expects where blocks have pointers to all of their neighbors.

3.11 Task Assigner

The task assigner takes a flowgraph that can contains both hardware and software blocks and divides the blocks between the four boards in the cluster. The main objective is to minimize the number of connections between the boards, in both software and hardware. In hardware, there is only a single two-way FMC link between two boards, so over utilization results in a flowgraph that cannot physically exist. In software, passing data over Ethernet costs CPU time, so there is a tradeoff between the benefit of spreading out the load and the cost of network communication.

The task assigner breaks up the flowgraph into two separate lists of blocks, with software blocks in one and hardware blocks in the other. The hardware blocks are placed first with no considerations about the software. This is due to the fact that while using Ethernet is not preferred there are no limitations on the total number of Ethernet connections. The hard limits

on the number of FMC links between boards take priority. A multiplexing scheme could be used to enable multiple data links to share a single FMC link, but there are some difficulties preventing that. Any link shared between two flowgraphs can only give each a fraction of the total data rate, so two high rate connections that share a link could be throttled and slow down the flowgraph. Keeping track of the bandwidth of each link would be possible, but it would require relative rate information to be known about every block that doesn't currently exist.

The task assigner works by first breaking the flowgraph up into even chunks and placing each on one of the boards. Each block then has a value calculated using Equation 3.11. Any connection that exists to a block on the same board is worth a positive point, and any connection to a block on a different board is worth a negative point. This formula could be extended to include information such as the size of an FPGA block or the CPU utilization of a software block. Unfortunately, this information doesn't currently exist in the GNU Radio block library. While the area of blocks could be used as a decent metric to not overfill an FPGA, the actual number of placements for a block can be unrelated to the blocks area. Currently the number of blocks on each board is just balanced.

$$value = C_L - C_N \qquad (3.11)$$

Blocks with a value that is less than one, called bad blocks, are considered for trade. First the bad block's value is calculated on each board that it has a connected neighbor on, as if the bad block were to be moved to that board. Then, all of the low-value blocks on the neighboring board are considered as if they were to be swapped. The move with the best value and the swap with the best value are then considered. If the move has a better value and the board to move the bad block to has fewer blocks than the board the bad block is currently on, then the block is

moved. Otherwise the best swap takes place. If neither a profitable swap nor a profitable move is found then the bad block is not moved. Figure 3.11.1 displays the thought process behind a move and a swap. All of the low-value blocks are continuously considered until all of the low-value blocks are considered in a row and no blocks move. At this point the placing is considered complete. After the hardware blocks are placed, software blocks that are connected to hardware blocks are pinned to the board where their hardware neighbor is located. These pinned blocks are never considered for swapping or moving. After that, the exact same placement method is run on the software blocks.

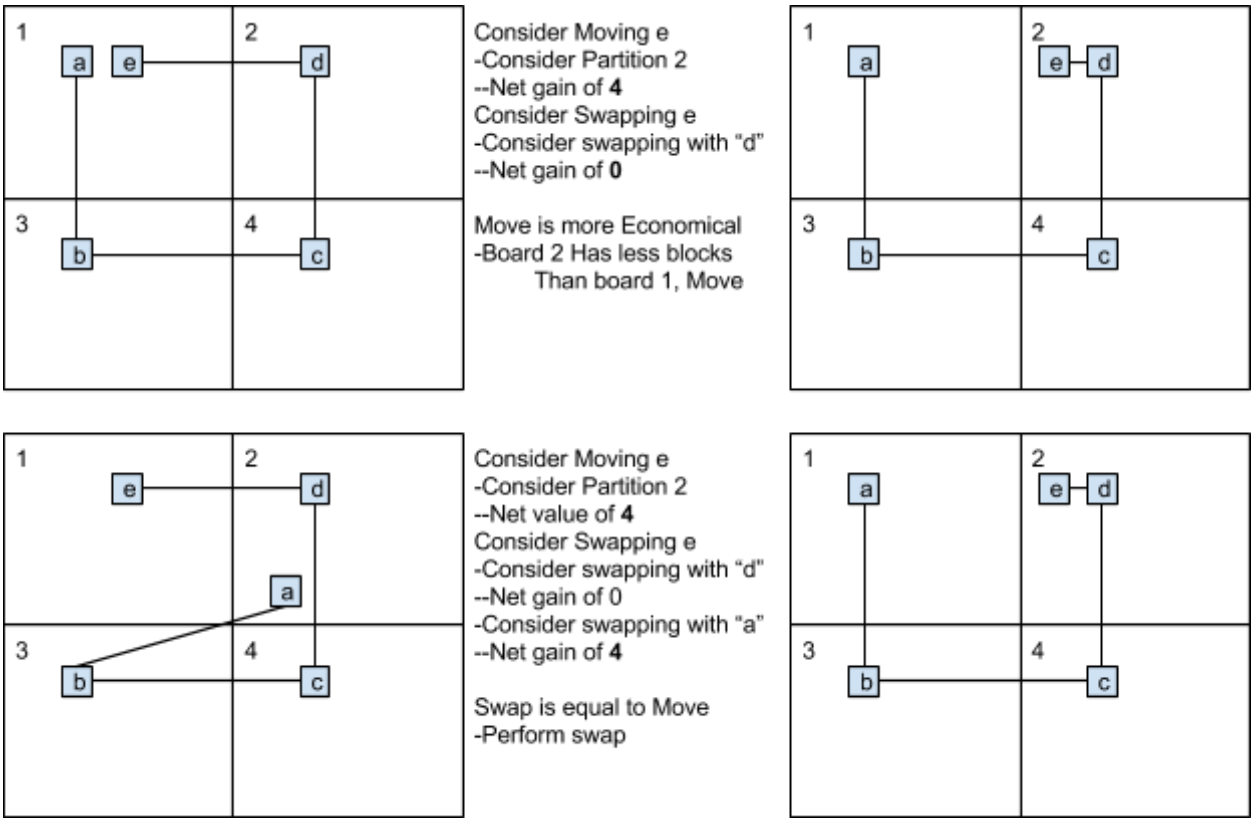

Figure 3.11.1 : Sample Moving and Swapping

Connections between boards and between hardware and software require special attention. A connection between two blocks on the same board is as simple as a single line in the EDIF, or a

single line in Python. A connection between two hardware blocks on different boards requires reserving a FMC link, and connecting to the proper ports on the blacktop to use that link. A connection between software blocks on different boards requires creating a TCP source on one, and a TCP sink on the other. This can be a little tricky because one runs as a server, and one runs as a client. If the server is not already present when the client is constructed, the client throws a Python exception and gives up. A retry mechanism had to be added that repeats the construction of the client until it connects.

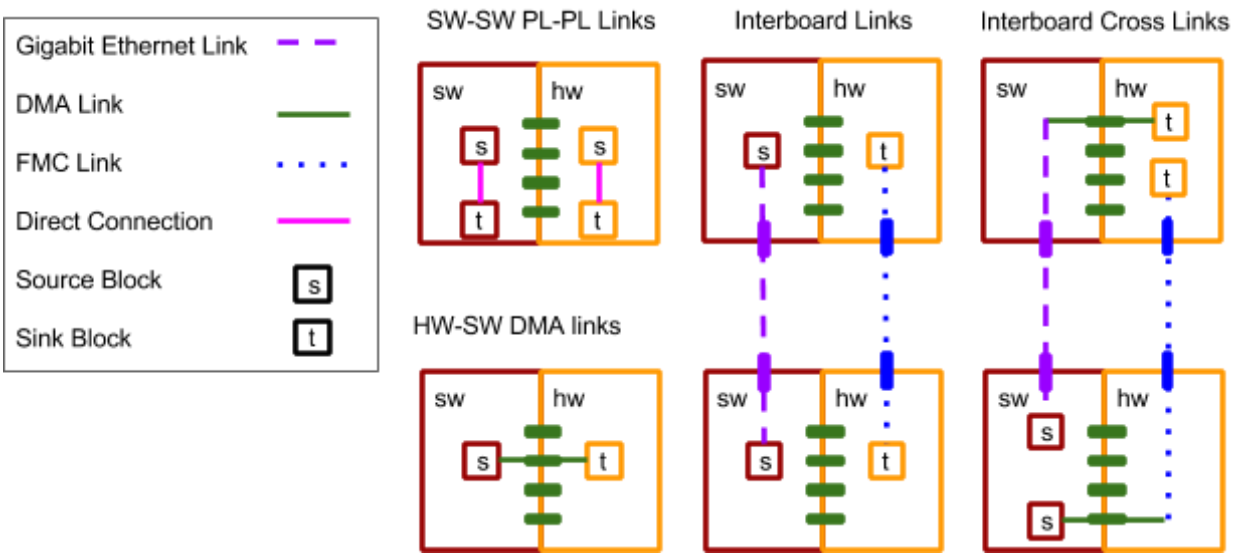

Figure 3.11.2 : Various Block connections

A connection between hardware and software on the same board first requires reserving a DMA channel. The FPGA block is then connected to the corresponding blacktop port in the EDIF, while the software blocks requires the creation and connection of a GNU Radio file sink or source block that writes to or from the driver interface file for that DMA channel. Connections between software and hardware on different boards are a little tricky, though generally avoided. First the task assigner checks if there is a free FMC link between the two boards, and routs the

connection through the DMA system on the software block's board and over the FMC link. If there is no free FMC link, the software block is connected to file source or sink on the hardware block's board though the use of a TCP link.

Due to the nature of the cluster, an incoming data stream may be present on all or just a few of the boards' FMC links. It is also possible that the data is coming into the FPGA from a source that is not part of its assigning. These problems are solved by module pinning and input tying. The affinity field that is already present in all GNU Radio blocks has been repurposed to indicate the target board in the cluster instead of the core in a CPU. The loss of pinning a software block to a specific CPU core is no real loss because the flowgraph maker doesn't know which blocks end up on which boards, so they cannot efficiently set core affinities. The first FPGA block in a flowgraph can be pinned to the core where the data is streaming in to ensure that the input is properly connected. In addition, the input to this FPGA block needs to be tied to the proper blacktop input port. A parameter was added to all FPGA blocks that, when populated, makes a connected from a manually specified location in the blacktop to the input port of the block. These two GNU Radio parameters are necessary and flexible enough to cope with any data source that can be feed into the blacktop. If the data streaming is coming in from a different location such as Ethernet or a USRP, this would be handled by using a software block as the source.

3.12 Fitting It All Together

Figure 3.12 depicts the total operational flow of the system. The system is started by running the cognitive engine, and giving it a list of signals. The cognitive engine can be run on any of the

boards. The first thing it does is give the netlist parser an initial GRC file. In this case, it instances a spectrum sensing application that contains only a hardware FFT and a connection to software. Because this is the same every time the cognitive engine is start, tFlow never has to be run here and the cached result is used. The energy detector is then run until signals of interest are found. The best possible netlist is generated by considering signal priority and reading in all of the GRC files to be run, and this is passed to the task assigner.

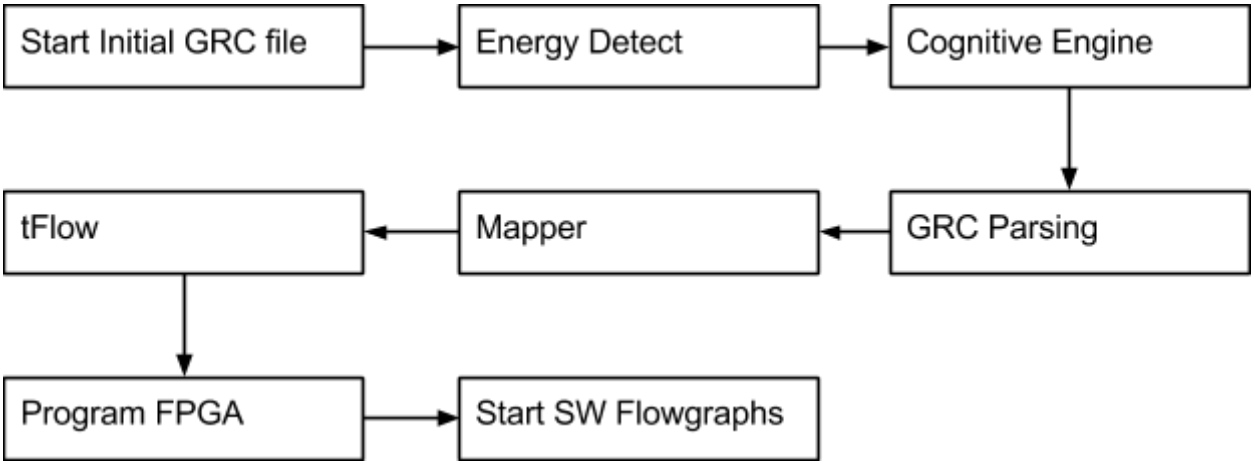

Figure 3.12 : Total flow

The task assigner is then called directly by the cognitive engine, meaning that it will run on the same ARM core. After the EDIF and Python flowgraphs are generated by the task assigner, they are all copied to the boards in which they will be used. From there, each ARM runs tFlow independently to generate a bit file for its own FPGA. This is a natural and efficient way to spread the load around. If the EDIF has been seen before, then the cached result is used instead. After the boards are done with tFlow, they are all programmed at the same time. Also, the Zedboard being used as a clock and data source has its FMC training sequence initiated at the

same time. As soon as this is over and while the FMC training sequence is still active, the parameterizable hardware blocks are configured and the Python flowgraphs are started.

Chapter 4
Motivating Platform and Improvements

The project that motivated the creation of this research will be summarized in the beginning of this section. The system had a similar architecture where there was a general-purpose processor accompanied by an FPGA. They were two separate devices that were connected through a PCI Express link. The system also used GNU Radio for its software processing components, just like this work. After the overview, the original system's limitations and how they are addressed in this work are discussed. Finally, a few benchmark results are presented that compare the performance between the ARM core and the TilePro64.

4.1 Old Platform Overview

The TilePro64 is a 64 core multiprocessor designed by Tilera [11]. As can be seen in Figure 4.1, each core is connected to six different switched networks that together form the Tilera iMesh. This interconnectivity manages communication to external resources such as SDRAM and Ethernet controllers as well as communication between cores in the form of cache coherency and message passing. This mesh network presents a cache-coherent memory architecture that is both uncommon on platforms with a large number of independent processor cores and is easy to develop on.

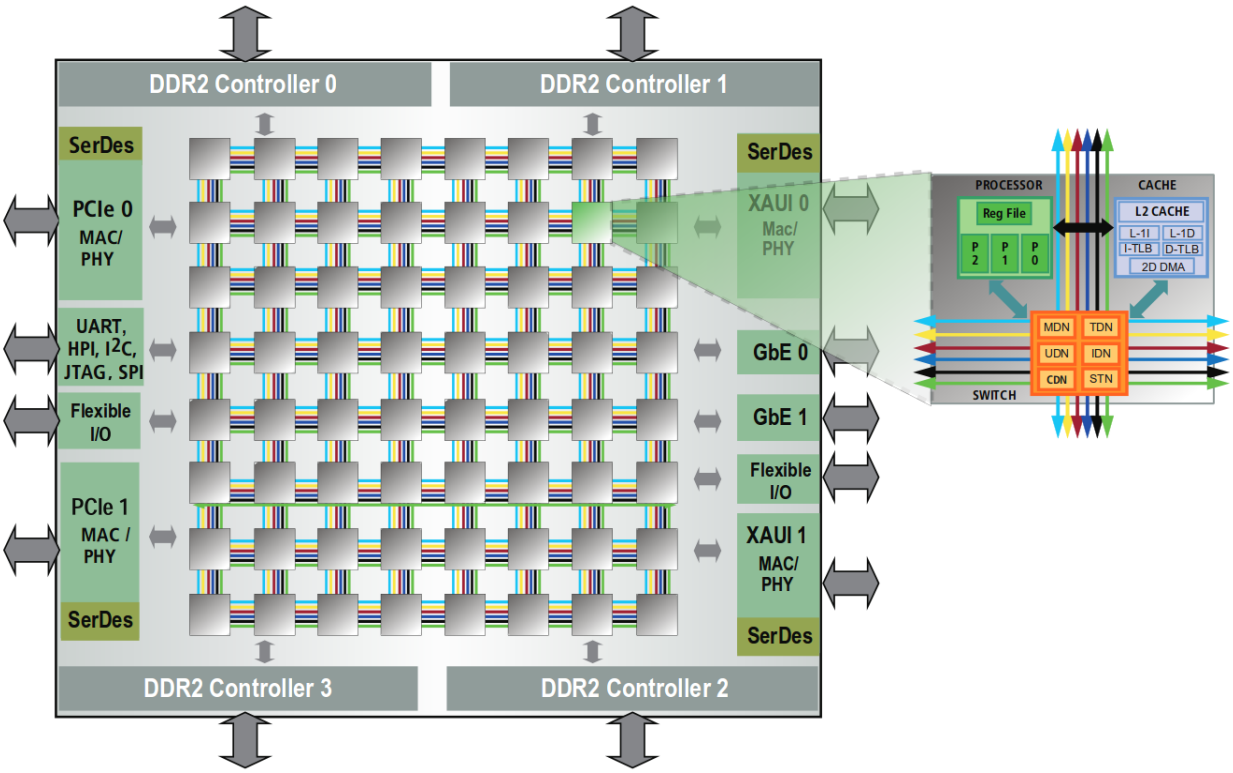

Figure 4.1 : Tilera Architecture
http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf. [Accessed: 06-Jun-2014]. Used under fair use, 2014. [11]

The cores themselves have a RISC architecture with three parallel execution pipelines, though each pipeline is limited in the functions that it can perform and only two can be used at a time in most cases. Each core has 64 KB of L2 cache, and the caches from multiple cores can be automatically combined and shared as a virtual L3 cache. The dedicated cache coherency network is what enables this virtual L3 cache sharing mechanism to operate at a high performance level. The chip itself uses about 23 watts when clocked at its lower speed of 700 Mhz, which puts it in the right power budget for a variety of embedded applications.

The TilePro64 is able to run SMP Linux and includes a standard GCC compiler and tool chain that make it compatible with many non-platform specific programs and libraries. The binary

version of GCC provided by Tilera is Version 4.4.3 which was released in 2010 and is a little antiquated, but TilePro support has been added to the mainline GCC release so it should be possible to build an up to date version independently. It was possible to build GNU Radio and all its dependencies such as boost and Python with some cross-compiling work, though there are some pretty significant drawbacks to this development path.

The Xilinx ML605 [12] evaluation board was used as the FPGA in this system. The ML605 interfaced both with an external ADC card over FMC and the TilerPro64 through PCIe. The FPGA contained the glue logic to attach to the ADC, the glue logic to attach to the PCIe bus, and a channelizer to break up the radio signal into frequency bins. To get an arbitrary sized bin at an arbitrary frequency, the frequency bins had to be chopped up into smaller pieces with an additional software channelizer, or combined with a software synthesizer.

The interface between the FPGA and software consisted of a single circular buffer that was read by many separate processes. Bins from the channelizer that were needed by software flowgraphs were tagged and copied over to the Tilera's memory. The processes then picked through the buffer and took data from tags they were interested in. This made it easy for processes to share incoming data, which was necessary due to the use of the fixed channelizer.

4.2 Old Platform Limitations and Improvements

The main limitation of the TilePro64 is its overall performance. While it does have many cores, each one is a relatively simple RISC design. Radio applications are only easily parallelizable to a certain extent. Putting individual GNU Radio blocks into their own threads is a natural way to

parallelize a flowgraph, but all it takes is one intensive block, such as a channelizer, that maxes out one of the small cores to slow down the entire system. GNU Radio flowgraphs often have uneven loads that lump large amounts of the CPU utilization into individual blocks. This caused repeated difficulties in previous development work. The solution is for the programmer to perform heavy optimizations and manually break these complex parts of their application up into small "tile" sized pieces that can run in parallel [13, Sec. 1.1]. This is frequently a daunting task that is not seen as often in architectures with fewer, more powerful cores. It is especially difficult when the bottlenecks are present in dependencies developed by someone else. Each ARM core is significantly more powerful, as the benchmark section later on discusses.

An additional limitation of the TilePro's weak cores is the need for dedicated driver cores. Because interfacing with hardware peripherals pushes the limit of a single core's performance, certain interfaces require Linux be restricted from using the core where the peripheral is located. While attempts were made to minimize the number of driver reserved cores, 5 still had to be reserved. Two were needed for Ethernet, two were needed for the PCIe driver interface, and one was needed for memory profiling. Switching to the ARM processor enables demand based switching of driver task like most other computer architectures.

The cores also lack hardware floating-point support, which is especially crippling due to the fact that GNU Radio has almost no fixed-point signal processing elements. GNU Radio support for integer operations is limited to basic math such as addition and multiplication. Where integer arithmetic can easily produce numbers that are too big and overflow or so small that they underflow, floating-point arithmetic manages a coefficient to ensure that no accuracy is lost.

This makes floating-point communication libraries much easier to program, debug, and maintain. This ease of use makes floating point the logical choice for GNU Radio. The ARM architecture contains SIMD floating-point support that enables each ARM core to have roughly 10 times the floating-point performance of a Tilera core, as is shown later in the benchmarks section.

An additional concern is the lack of community support. While Tilera does develop a number of important utilities such as GCC and GDB, most performance minded programs and tools are architecture specific or at least architecture aware. Many of these can be forced to compile a generic platform independent version of themselves, but others tools such a *valgrind* cannot be ported without intimate knowledge of the hardware and a large amount of assembly work. One major side effect of this is that there are no Linux distributions available with repositories of pre-built software. This means that just about every tool, library, or application that is needed to run on the TilePro architecture has to be built by the system developers, which is an inefficient development path. The ARM architecture on the other hand has wide support from community developers. A good example of this is that the Debian project is able to build over 97% of all its approximately 37,000 packages for ARM, with only x86 and PowerPC having higher build rates [14].

The Tilera also has poor task switching performance. The Linux scheduler normally moves tasks around as needed depending on demand and core utilization. These normal context switches cause major problems when the Tilera system is heavily loaded, with some tests seeing as much as a 100% improvement when manually setting each thread to its own core. The solution that was used required manually specify affinity for every task when starting execution.

If there are more jobs than cores, then dynamic control is necessary to keep everything running at top performance, basically overwriting the Linux scheduler. The ARM architecture does not suffer this heavy task switching penalty, meaning no special attention needs to be paid to it. The task assigner actually evolved from a process assigner written to set core affinities to deal with this issue.

The tag system used in the shared PCIe buffer caused some major problems with flow management. The shared circular buffer located in memory was constantly written to by the FPGA. This meant that any process that wanted to read data from the buffer had to get its data out before it was overwritten. In the case of single consumer on a lightly loaded system, it didn't really matter because if it couldn't consume data fast enough. Data would have to be dropped somewhere anyways. The problem became prevalent in a heavily loaded system with multiple readers. More readers resulted in more data being transferred from the FPGA, and as a result that data was stored in the buffer for smaller amounts of time. With the system under heavy load, even a process reading a single channelizer bin could miss out on incoming data. There was also a race condition where a thread could read a tag and think that it was going to be reading valid data but the FPGA over wrote that location in the meantime with a different tag data, causing the thread to read data from a different channel. These issues are not present in the ARM DMA system because each channel gets its own circular buffer and it isn't overwritten until it has been read. Any data dropping that might occur happens in the FPGA with this work.

The channelizer system in the FPGA was a powerful tool that provided the entire spectrum for software to use, but it was really only making limited use of the FPGA's processing capabilities.

There are a plethora of useful operations that FPGAs excel at performing, such as convolution and full digital radio demodulators. Many of the Tilera's shortcomings were exacerbated by the large amounts of data that were being passed into it. A properly designed FPGA accelerator has no consumption rate issues and can be used to reduce that input data from a torrent of noise to a stream of desired data. This work makes better use of the FPGA through the use of tFlow's construction of arbitrary radio specific bit files.

4.3 Benchmark and Platform Comparisons

Two benchmarks were run on the Tilera and Zynq platforms to gage their raw integer and floating-point performance. Both benchmarks were multi-threaded through the use of *pthreads* within one process. In each of the charts below the Zynq and Tilera plots represent the data directly from the benchmarks, while the Zynq Cluster plot represents the aggregate performance of the entire cluster even though it cannot directly run the benchmarks. The plots below have their horizontal axes normalized to the total number of cores present in the system. When the horizontal axis is one, the TilePro64 is running 59 threads, the single Zynq board is running two threads, and the Zynq cluster is running eight threads. At this point on the horizontal axis, there is one benchmark thread running on every core in the system.

4.3.1 Linpack

The Linpack benchmark is a test of a systems floating-point arithmetic performance. The benchmark was parallelized by running multiple identical instances concurrently. The computation consists of solving a dense *n* by *n* series of linear equation, which is represented by an *n* by *n* matrix. The default matrix size of 200 was used, and the computation is repeated until

a total run time of 10 seconds is achieved. This ensures that random variations in the performance and imprecise timers are not an issue.

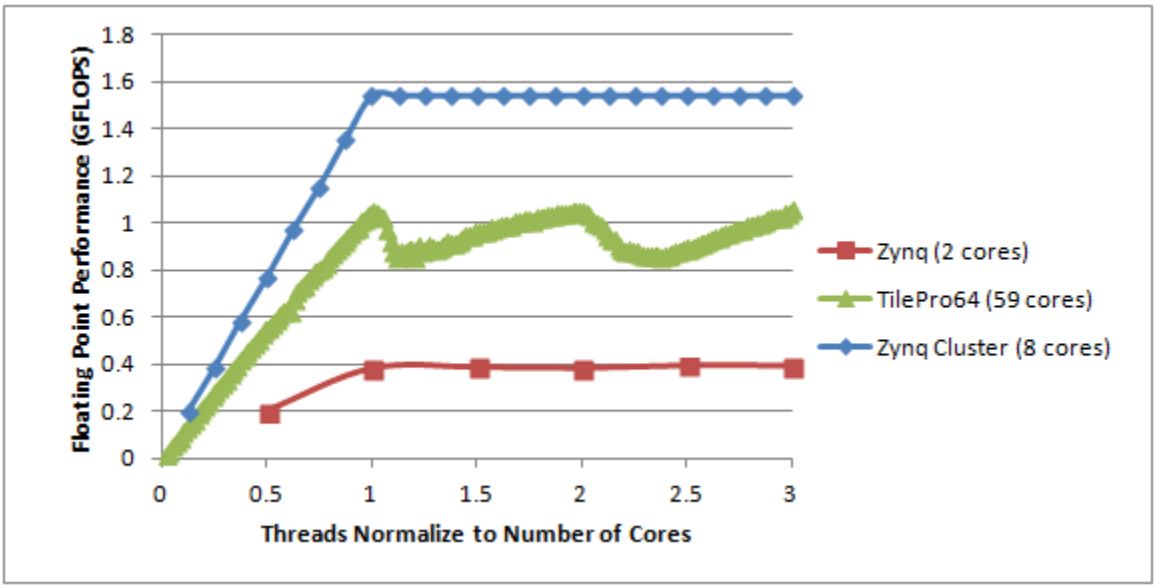

Figure 4.3.1 : Linpack Performance

In Figure 4.3.1, the horizontal axis represents the number of concurrent threads normalized to the total number of cores on the processor and the vertical axis represents the aggregate performance of all the running threads combined in FLOPS. A single thread running on an ARM core performs about 10 times better than a single thread running on a Tilera core, with hardware floating point support likely being the reason for the performance difference. In both cases, performance scaled roughly linearly with the number of cores on the chip. The aggregate floating-point performance of the entire cluster using the ARM processors only is roughly 45% greater than the performance of the Tilera chip.

There is a regular pattern in the performance of the Tilera as it exceeds the total number of cores on the chip. The performance rises consistently until the thread count is equal to the number of

cores, and then it dips back down. It then returns to climbing to the same peak performance level which it reaches at twice the number of cores. This dip between multiples of the core count is repeated for higher multiples as well. The lowest dip occurs when 71 threads are present, which is 12 more than the number of cores. The performance here is about 80% of the peak performance. This pattern is likely due to the poor context switching of threads between cores. At even multiples, the cores can just alternate between an integer number of threads, while uneven multiples requires moving threads around to give each a fair CPU percentage. This unusual pattern was not observed on the Zynq platform.

4.3.2 Convolution

The convolution benchmark is a test of a system's integer arithmetic performance. Two vectors that are 16,384 integers long are generated and then a series of convolutions are performed. Because convolution is a common operation in communication libraries, this benchmark is a good indicator of fixed-point software defined radio performance. Just like the Linpack benchmark, the computation is looped through until a run time of over 10 seconds is achieved. The total memory utilization of the vectors themselves is 128 KB per thread.

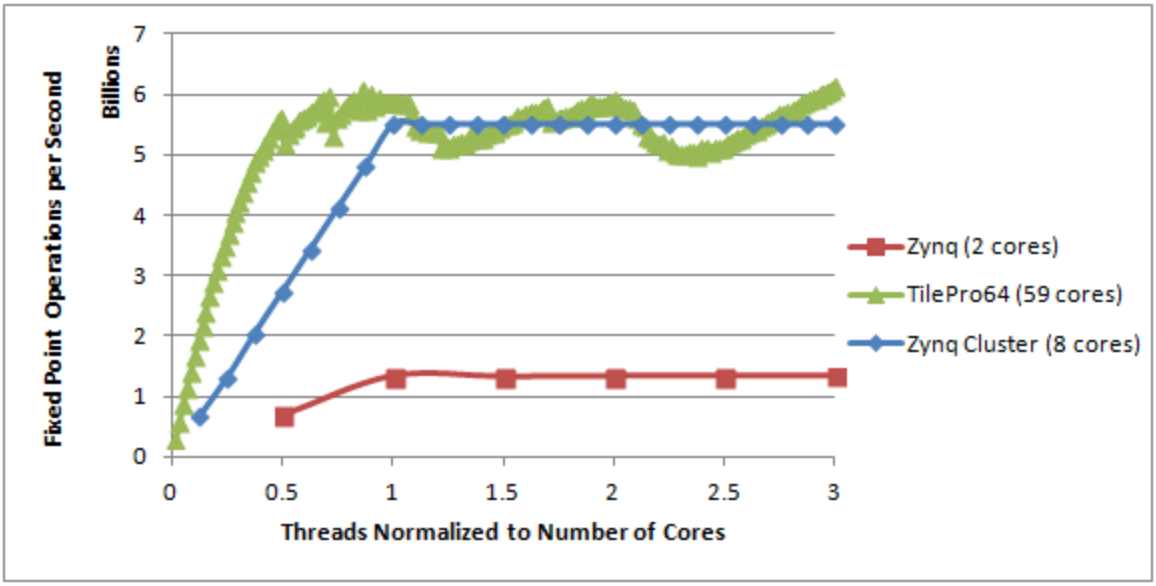

Figure 4.3.2 : Convolution Performance

As before, the horizontal axis represents the number of running threads normalized to the number of cores on the chip. The vertical axis represents the combined performance of all the running threads which is measured in integer operations per second. A single thread running on an ARM core performs about 2.25 times better than a single thread running on a Tilera core. Just like the Linpack benchmark, the performance on the Tilera chip peaked around multiples of the core count, and dipped in between.

While the performance on the ARM cores scaled linearly, the performance on the Tilera did not. The rapid tapering off of performance as the thread count is increased can be easily seen in Figure 4.3.2. If performance had scaled linearly, then the maximum aggregate performance would have been three times higher than what was observed. The reduction in performance is likely occurring because the cores are running out of cache space. At 59 threads, about 7.4 MB of memory are in use by the vectors, which is larger than all of the caches on the Tilera chip combined. A benchmark was run on an ARM core with 177 threads, which requires about 20

MB of memory, and there was only a 5% drop in total performance. The Tilera's 66% drop in performance indicates that the caching and memory system can be easily overwhelmed. This tapering in performance resulted in the Tilera and the ARM cluster having similar aggregate integer performance in this benchmark.

Chapter 5
FM Tuner Cognitive Application

This demonstration application is an FM tuner that searches the spectrum for stations of interest and demodulates them. All stations that should be examined need to be put in the database so the cognitive engine knows about them. Also, preferred stations are given higher priority so they will be tuned into when multiple stations are present.

The flowgraph itself, shown in Figure 5.1, makes suboptimal use of hardware and software blocks. The first noticeable inefficiency is that there are five separate crossings between hardware and software. An ideal flowgraph would only have one or two of these crossings. In addition, there is no decimation that occurs in the PL before data is transferred to software. While this flowgraph still functions due to the relatively low sample rate, a higher bit rate would most likely saturate the software blocks. Due to the fact that all the signals are the same, the only difference between each signals flowgraph is the frequency of the mixer. These suboptimal properties may not be ideal for a software radio, but provide a good test for the system.

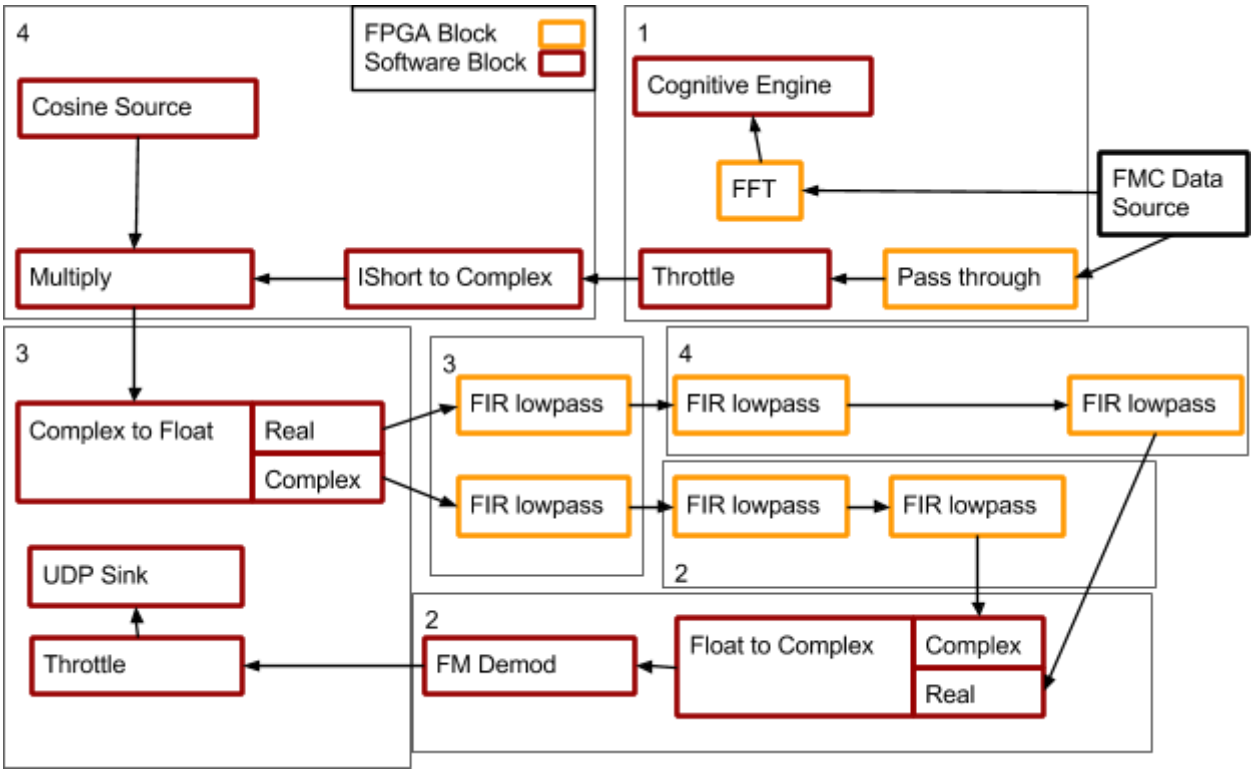

Figure 5.1 : Assigned Flowgraph

The software blocks in the flowgraph consists of a block to convert the data from interleaved shorts to complex float, followed by a mixer with the frequency adjusted to put the signal of interest at baseband. After FPGA processing, the FM demodulator included in GNU Radio is used to perform the actual demodulation. This is then piped over the network to a computer with speakers for listening. The sample rate is at 320k until the FM demodulator's built-in decimator reduces it to the audio rate of 32k. The most CPU intensive block is the FM demodulator, which takes up about 30% of one ARM core.

The FPGA blocks in the flowgraph consist of a FFT for use by the cognitive engine, a pass-through for the first link to the software, and six real floating-point FIR filters. Because the FIR filters are real and not complex, the complex samples are split up in software and the I and Q

channels are processed in parallel, with each passing through three filters. Due to this configuration, the FIR filters are not useful for much more than being low-pass filters. That is why this flowgraph mixes the signal down to baseband in software. A few complex FIR filters would be more efficient, but this does create a good challenge for the system.

Figure 5.1 shows how the task assigner distributed the load across the cluster. Because the FFT and pass-through blocks were both pinned to Board 1 to ensure that they were attached to the ZED source (which is only present there), the remaining six modules were placed on the other three boards to keep things even. The FIR modules are assigned into groups that minimize inter-board connectivity utilization. The connection between the second FIR filter on Board 4 and the software block on the ARM on Board 2 is passed through the FMC to the blacktop of Board 2 and then through one of the DMA channels.

Step	**Time Taken**	**Accumulated Time**	**Cached Time**	**Cached Accumulated**
1) Cognitive Engine Initialization	0.125 S	0.125 S	0.125 S	0.125 S
2) Generate Initial Netlist + Assign	<0.125 S	0.125 S	<0.125 S	0.125 S
3) Cached tFlow + program	3.00 S	3.125 S	3.00 S	3.125 S
4) FMC Training	5.00 S	8.125 S	5.00 S	8.125 S
5) Energy Detect + Decision Making	<0.125 S	8.125 S	<0.125 S	8.125 S
6) Generate FM flowgraph + Assign	<0.125 S	8.125 S	<0.125 S	8.125 S
7) tFlow Board 1	47.75 S		1.00 S	
tFlow Board 2	80.50 S		1.00 S	
tFlow Board 3	85.50 S	93.625 S	1.00 S	
tFlow Board 4	76.25 S		1.00 S	9.125 S

8) Program	2.00 S	95.625 S	2.00 S	11.125 S
9) Start Python Flowgraphs	<0.125 S		<0.125 S	
FMC Training	5.00 S	100.625 S	5.00 S	16.125 S

Table 5.2 : Time Taken By Step

Figure 5.2 shows how much time each step in the flow takes. The total process from starting the cognitive radio to the flowgraph running is about 100 seconds. The biggest influence is tFlow at 85% of the runtime, with FMC training coming in second at 10%. Much of the remaining 5% is taken up by waiting for SSH connections to be made, which could be avoided through a system of manager programs. The FMC training time could be reduced by the creation of a management program that runs on the receiving end of each link in the ARM that notifies the source of the training sequence when training is complete. The cached column shows how long the flow takes when tFlow doesn't have to be run because cached results are used. The cached run took about 84 seconds less than the non-cached run.

tFlow takes about 85 seconds to finish placement, routing, and bit file generation in the longest case. Board 1 only contains the FFT and pass through blocks which are both quite small, which can be seen in Figure 3.7.2. All of the other boards have two FIR filters located in their programmable logic, which together take up nearly all the resources. As can be seen in Figure 5.1, Boards 2 and 3 have four 33-bit connections to make while Board 4 only have three. This is likely why Board 4 routes a full 4.25 seconds faster.

Chapter 6

Results

This chapter provides a comparison between this work and several of the similar projects from Chapter 2. Projects [2], [3], and [4] were chosen because they share common software defined radio applications with this work. Table 6.1 summarizes the key points that differentiate the different platforms.

Property	**This Work**	**Iris on Zynq [2]**	**Autonomous System on a Chip [3]**	**NoC architecture [4]**
Processor	Hard ARM	Hard ARM	Soft PowerPC	Soft Microblaze
Integrated SDR Processor	Yes	Yes	No	No
Included SDR Framework	GNU Radio	Iris	No	No
Dynamic Reconfiguration	tFlow	Xilinx Partial Reconfig	Xilinx Partial Reconfig	Multiple Xilinx Partial Reconfig
Arbitrary Dynamic Reconfiguration	Yes	No	No	Yes
Scalable Cluster	Yes	No	No	No

Table 6.1 : Feature Comparison Table

One key differentiator is whether or not an integrated software defined radio processor is included. This work and [2] both utilize Zynq ARM processors with the ability to perform communication computation in software, whereas [3] and [4] perform all their computations in

the FPGA. Part of this reason is due to the lack of a high performance hardware processor. The PowerPC soft core in [3] is only used to run cognitive radio applications, whereas the Microblaze present in [4] was only used for basic control of the partial reconfiguration with an external host being required to give it commands.

Both this work and [2] provide a software framework to make use of both the hardware and software components of the system. Because the hardware decoder used in [3] is actually located in the static region, the system is not flexible enough to include multiple radio receivers and there would be little benefit in including a SDR framework. The system provided by [4] requires an external host to operate and has the flexibility to be integrated into a SDR framework, but it wasn't. Both GNU Radio and Iris provide similar plug-in architectures that enable the integration of heterogeneous computing systems. GNU Radio provides a graphical front-end to expedite the development of radio applications, while Isis requires manually editing XML files. GNU Radio is also an official GNU project that has been released under an open source license since 2001, whereas Iris wasn't publicly released until 2013, resulting in GNU Radio currently being a more mature framework with greater community support.

While this work uses tFlow to provide arbitrary and dynamic placement and routing of resources within the FPGA, all of the other works utilize the standard Xilinx partial reconfiguration flow. In the case of [2] and [3], there is only one reconfiguration region. This means that only entire radios that were foreseen and created using the Xilinx tool flow can be used. tFlow in comparison has the ability to place and route any combination of modules from a library in an FPGA, providing the ability to build arbitrary radios. The multiple reconfiguration regions

interconnected by a network on chip provided by [4] does provide the potential to build arbitrary radios, but it isn't discussed in the paper. One huge benefit provided by this flexible approach is the ability for a cognitive engine to entirely customize radios depending on environmental factors. For example, the cognitive engine could skimp on filtering in a low-noise environment to provided more programmable logic space for additional decoders.

The only related work summarized in Chapter 2 to build a scalable cluster was [1], which was excluded from the feature comparison table because it lacked SDR specialization. The key differences between this work and [1] are the load distribution systems and the interconnect systems. The Hadoop system used in [1] utilizes a MapReduce framework for generic computation, while the task assigner utilized in this work is specialized for streaming data applications such as software defined radios. This work utilizes both Gigabit Ethernet for communication between ARM cores and FMC communication links for FPGA to FPGA transfers, while [1] only utilizes 100 megabit Ethernet. The general-purpose computing applications of [1] would likely not benefit greatly from the inclusion of inter-FPGA communication, but using Gigabit Ethernet would provided a noticeable boost to any inter-ARM communications. With the exception of the different Ethernet speed, the differences here are largely driven by separate application types.

Chapter 7
Conclusion

This work presented an embedded scalable cluster platform with software defined radio applications. The hybrid half ARM processing core and half FPGA Zynq chip used combines the ease of programming that a general-purpose processor provides with the high-performance of an FPGA. tFlow provides a reconfiguration flexibility unseen in other systems that makes arbitrary data paths in the FPGA possible. The task assigner presents a mechanism through which GNU Radio flowgraphs consisting of both hardware and software blocks are automatically placed on boards in the cluster. This provides a capability that is convenient for radio design and debugging, yes provides profound abilities in the area of cognitive radio systems. A cognitive radio without the ability to make these changes is either limited to the lower performance of a software only system, or limits its flexibility to a few specific applications.

7.1 Future Work

There are several areas where this work could be expanded to increase its capabilities. The DMA system currently only provides a single read buffer and a single write buffer per physical high performance AXI interconnect used. Due to the fact that the memory bandwidth limit of the system can be reached with a single high performance AXI channel, the programmable logic could be extended to support multiple buffers per channel and save area in the FPGA for a larger

blacktop. This could easily be done without making any changes to the Linux driver. As discussed in the DMA section of the platform overview, there are also optimizations to the driver that could be performed to attain a 100% increase in DMA bandwidth.

If a database of the resources utilized by all of the hardware and software GNU Radio blocks were created, the task assigner could be modified to provide a more even and efficient load distribution. The current system of just keeping even numbers of blocks on each Zynq means undesirable inter-board communications are used when they could be avoided, and a single Zynq could be overloaded with intensive or large blocks and slow down the system even when the total needed resources are less than what the cluster provides. The task assigner also currently doesn't check if too many FMC links are in use, it just minimizes the utilization. This works well for a fully connected cluster, but adjustments would have to be made to support other connectivity patterns.

The current radio design flow requires creating a radio in GNU Radio companion and then running it through an external tool to realize the radio on the cluster. Better integration with companion could use the existing generate and run buttons to perform this flow automatically. It would provide no benefit to cognitive applications, but would be beneficial to radio design and testing. GNU Radio provides a library of mathematical kernels that utilize SIMD instructions for the benefit of software blocks called VOLK. Kernels have only been written for various x86 architectures, but speedups of well over 100% were observed for many of the basic math operations [15]. Populating VOLK with kernels for the ARM architecture could provide a noticeable performance boost when using built-in GNU Radio blocks.

Bibliography

[1] Z. Lin and P. Chow, "ZCluster: A Zynq-based Hadoop cluster," in *Field-Programmable Technology (FPT), 2013 International Conference on*, 2013, pp. 450–453.

[2] J. van de Belt, P. D. Sutton, and L. E. Doyle, "Accelerating software radio: Iris on the Zynq SoC," in *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*, 2013, pp. 294–295.

[3] M. French, E. Anderson, and Dong-In Kang, "Autonomous System on a Chip Adaptation through Partial Runtime Reconfiguration," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, 2008, pp. 77–86.

[4] J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot, "A FPGA partial reconfiguration design approach for cognitive radio based on NoC architecture," in *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, 2008, pp. 355–358.

[5] "Zynq-7000 All Programmable SoCs." [Online]. Available: http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-combined-product-table.pdf. [Accessed: 07-Jun-2013].

[6] "Zynq-7000 Silicon Devices." [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm. [Accessed: 07-Jun-2013].

[7] "Linaro About." [Online]. Available: http://www.linaro.org/about/. [Accessed: 07-Jun-2013].

[8] "Welcome to GNU Radio." [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki. [Accessed: 07-Jun-2014].

[9] "GNU Radio Companion." [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion. [Accessed: 07-Jun-2014].

[10] A. Love, W. Zha, and P. Athanas, "In pursuit of instant gratification for FPGA design," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013, pp. 1–8.

[11] "TILEPro64_Processor_PB019_v4(1).pdf," *TILE Pro 64 Processor: Product Brief*. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf. [Accessed: 06-Jun-2014].

[12] “Virtex-6 FPGA ML605 Evaluation Kit.” [Online]. Available: http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf. [Accessed: 07-Jun-2014].

[13] Tilera Corporation, “Multicore Development Enviroment Optimization Guide,” 06-Apr-2011. [Online]. Available: http://www.tilera.com/scm/docs/UG105-Optimization-Guide.pdf. [Accessed: 07-Jun-2014].

[14] “Debian Build Stats.” [Online]. Available: https://buildd.debian.org/stats/. [Accessed: 01-Jun-2014].

[15] T. Rondeau, N. McCarthy, and T. O’Shea, “SIMD Programming in GNU Radio: Maintainable and User-Friendly Algorithm optimization with VOLK.” [Online]. Available: http://gnuradio.org/redmine/attachments/download/422/volk.pdf. [Accessed: 07-Jun-2014].