Heterogeneous Processing in Software Defined Radio: Flexible Implementation and Optimal Resource Mapping

Frank B. Bieberly

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

Electrical Engineering

Allen B. MacKenzie, Chair

Peter M. Athanas

Carl B. Dietrich

February 8th, 2012

Blacksburg, Virginia

Keywords: Software Defined Radio, Heterogeneous Processing, Resource Mapping

Heterogeneous Processing in Software Defined Radio: Flexible Implementation and Optimal Resource Mapping

Frank B. Bieberly

ABSTRACT

The advantages provided by Software Defined Radios (SDRs) have made them useful tools for communication engineers and academics alike. The ability to support a wide range of communication waveforms with varying modulation, encoding, or frequencies on a single hardware platform can decrease production costs while accelerating waveform development. SDR applications are expanding in military and commercial environments as advances in transistor technology allow greater computational density with decreased power-consumption, size, and weight. As the demand for greater performance continues to increase, some SDR manufacturers are experimenting with heterogeneous processing platforms to meet these requirements.

Heterogeneous processing, a method of dividing computational tasks among dissimilar processors, is well-suited to the data flow programming paradigm used in many common SDR software frameworks. Particularly on embedded platforms, heterogeneous processing can offer significant gains in computational power while maintaining low power-consumption, opening the door for affordable and useful mobile SDR platforms.

Many past SDR hardware implementations utilize a partially heterogeneous processing approach. A field programmable gate array (FPGA) is often used to perform high-speed processing (DDC, decimation) near the radio front-end while another processor (GPP, DSP or FPGA) performs the rest of the SDR application signal processing (gain control, filtering, demodulation). A few recent SDR hardware platforms are designed to allow the use of multiple processor types throughout the SDR application's processing chain. This can result in significant benefit to SDR software that can take advantage of the greater heterogeneous processing now available.

This thesis will present a new method of heterogeneous processing in the framework of GNU Radio. In this implementation a software wrapper allows a DSP to participate seamlessly in GNU Radio applications. The DSP can be directly substituted for existing GNU Radio signal processing blocks–significantly expanding the platform's capabilities while maintaining the benefits of the component-based design methodology. A similar approach could be applied to additional processing elements (e.g. FPGAs and co-processors) and to other SDR software frameworks.

As the capabilities of this heterogeneous framework increase users will be required to assign hardware resources to signal processing tasks to maximize performance. To remove this burden, a method of predicting GNU Radio application performance and a heuristic resource mapping algorithm, which seems to perform well in practice, are presented.

Contents

List of Abbreviations

ADC	Analog-to-Digital Converter
BPSK	Binary Phase Shift Keying
CBD	Component Based Design
CBE	Cell Broadband Engine
CMEM	Contiguous Memory
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DDC	Digital Down Conversion
DMA	Direct Memory Access
DSP	Digital Signal Processor

FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GA	Genetic Algorithm
GCC	GNU Compiler Collection
GPP	General Purpose Processor
IF	Intermediate Frequency
JTRS	Joint Tactical Radio System
KUAR	Kansas University Agile Radio
MAC	Multiply and Accumulate
NSF	National Science Foundation
OMG	Object Management Group
OSSIE	Open Source SCA Implementation, Embedded
PPE	PowerPC Processor
RAM	Random Access Memory

RF	Radio Frequency
SCA	Software Communications Architecture
SDR	Software Defined Radio
SIMD	Single Instruction, Multiple Data
SoC	System on a Chip
SORA	Software Radio
SPE	Synergistic Processing Element
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VLIW	Very Long Instruction Word
WARP	Wireless Open-Access Research Platform

List of Figures

List of Tables

Chapter 1

Introduction

Advances in processor technologies enable smaller, mobile, and more capable SDR platforms for research, radio enthusiasts, and mobile commercial and military applications [1]. The technology and techniques have matured to a point where SDR platforms are now common in research environments as well as many commercial uses (e.g. mobile radio base stations, public safety radio) [2]. However, the increased performance (battery-life, bandwidth, waveform complexity) continues in many cases to be limited by the computational power available on the SDR platform. In particular there is a gap in the availability of SDR platforms that combine the performance necessary for participating in current communication standards, low power-consumption, and easy software development. While the upward trend of processor technology continues in general purpose processors (GPPs), initially based on frequency scaling and now by multi-core architectures, additional performance and efficiency can be gained by adding heterogeneous processing

elements to the computing system.

Heterogeneous processing is a rising trend in the field of high-performance computing, where program execution times are often limited by the computational resources available. This is a method of off-loading computational tasks to processors containing different architectures that offer benefits for certain computational tasks. However, this additional computational efficiency and power comes at the cost of increased complexity of the software development. Additionally, not all tasks are well suited for processing on certain heterogeneous architectures.

Some processing tasks commonly found in SDRs, however, are well-suited for implementation on DSPs or FPGAs. These tasks arise from the data-flow nature of SDRs in which relatively simple computations (e.g. filtering, encoding, scaling) are performed on streams of data. As these tasks can be common to many SDR applications, the additional development costs to incorporate the heterogeneous processors are offset by their frequent use.

The common processing elements seen in SDR platforms are GPPs, FPGAs, and DSPs. GPPs are well suited for highly branching programs and have the shortest development cycle. Unfortunately, they have limited ability to perform parallel computations. However, recent GPPs may have access to some single instruction multiple data (SIMD) instruction sets. DSPs have an architecture that is specifically designed for signal processing tasks. This may include memory architecture designed for streaming data, SIMD operations, very long instruction word (VLIW) instruction sets, and fast multiply and ac-

cumulate operations as well as many other signal processing techniques. DSPs excel at tasks that can be pipelined (a sequence of tasks performed repeatedly for each sample in a buffer/array) and represent a moderate 1.5-2x increase in development time [3]. FPGAs are capable of large-scale hardware parallelism. Unlike GPPs or DSPs which have instruction queues and rely on sequential execution, FPGAs are able to implement many separate logical functions which can run simultaneously. FPGAs are well-suited for tasks that can be spatially separated or use non-standard data-type representations, however, they require significantly greater development time compared to GPPs and DSPs [4].

Another important design concept for SDR development which is found in the two popular SDR software packages that were evaluated in the course of this research (GNU Radio and OSSIE) is that of component-based design (CBD). In this framework, signal processing chains are broken into many component blocks that can then be recombined in arbitrary configurations and with different parameters for quick adaptation of the radio. This promotes code reuse and portability.

By introducing heterogeneous processing abilities and maintaining the CBD architecture, we can realize much of the gains from heterogeneous processing, while avoiding the development costs of using the more complex processing architectures. This thesis focuses on the design and implementation of such a framework. Specifically, making heterogeneous processing available as a component within a framework so that it requires little or no configuration by the end user to utilize.

1.1 Contribution

This thesis will demonstrate a heterogeneous processing framework for use on an embedded SDR platform. In particular, the method will preserve the paradigm of component-based design to allow transparent operation within an SDR software package while providing a significant performance improvement. We will see that heterogeneous processing on the particular hardware and software platforms chosen for this thesis has already been implemented. The novel contribution outlined in this thesis will consist of a new data transfer mechanism which improves upon previous implementations while providing greater flexibility to the user.

Additionally, a method for characterizing the performance of the SDR software is described. This method is designed to profile signal processing tasks on arbitrary hardware platforms without user oversight and with sufficient accuracy to allow prediction of SDR flow graph throughput. Finally, based on the performance predictions, a method for heuristic resource mapping of the heterogeneous processors is described. We evaluate the performance of this method on sample flow graphs, demonstrating performance near that of optimal mapping (which requires brute force optimization). Used together, these methods for performance prediction and resource mapping complete our goal of maximizing the benefit derived from using heterogeneous processing while minimizing development time for users.

1.2 Thesis Organization

This thesis will begin with an overview of SDR software (Section 2.2) and hardware architectures (Section 2.1). Next we will discuss the use of heterogeneous processing in an SDR environment and the concerns involved in implementing a heterogeneous processing framework in Section 2.3. We will look into some of the previous work related to enabling heterogeneous processing on SDR platforms in Section 2.4.

In Chapter 3 I will describe the implementation of my heterogeneous processing method including the development environment and performance as it compares with previous work in this area. Next, in Chapter 4 I will describe a method of profiling and predicting the performance of GNU Radio signal processing tasks. Using the performance prediction results, a method of automatic resource mapping is presented in Section 4.3. Finally, in Sections 5.1 and 5.2 we will conclude with a discussion of the benefits and contributions provided by this implementation as well as several future expansions to this design.

Chapter 2

Technology Overview

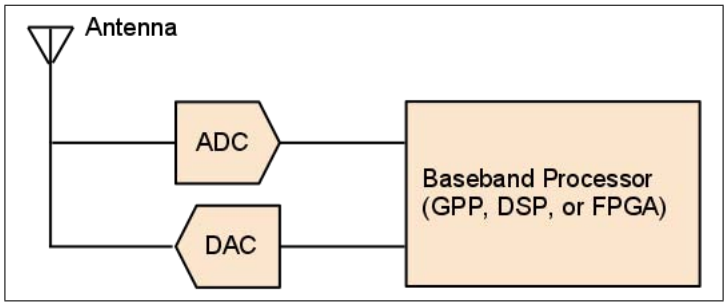
In this chapter we review current SDR hardware and software architectures and identify their strengths and weaknesses. Next, we will describe previous work in heterogeneous processing techniques on several different hardware/software platforms. Finally, we will give justification for, and a description of, the hardware that was chosen for this research.

2.1 SDR Hardware

All SDR platforms require a hardware front-end from which digital samples of the RF spectrum are received. Figure 2.1 illustrates the ideal implementation of an SDR. An analog-to-digital converter (ADC) takes samples directly from an antenna and passes the samples to a signal processing unit. Similarly, to transmit, samples from the signal pro-

cessing unit are sent to a digital-to-analog converter (DAC) which synthesizes the analog signal sent to the antenna. In practice, due to performance limitations of the ADCs, there must be RF hardware between the antenna and ADC to assist in recovering the signal. Typically, this RF hardware consists of low noise amplifiers, filters, and mixers to bring the signal of interest to an intermediate frequency (IF) which is sampled by the ADCs. In this thesis we will be focusing on the hardware that is responsible for performing the signal processing on samples after they have been collected by the ADC.

Figure 2.1: Ideal SDR implementation



There are two common architectures for SDR hardware platforms and one emerging trend that shows promise. In the past 10 years the majority of SDR platforms have been centered around either GPP or FPGA-based designs.

2.1.1 GPP-based

Examples of GPP-centric design can been found in the Software Radio (SORA) project designed by Microsoft Research [5], the Universal Software Radio Peripheral (USRP) line of products designed by Ettus Research (now part of National Instruments) [6], and the

FUNcube dongle SDR designed in conjunction with the FUNcube satellite program [7].

In this design, the samples from the ADC may be fed into an FPGA which will perform digital signal processing common to almost all SDR applications (e.g. digital down conversion, decimation, data-type conversion from the ADC output of 12 or 14 bits to the GPP input of 16-bit shorts). Samples are then fed from the FPGA to the GPP for the majority of the signal processing (filtering, demodulation, decoding, etc.) that will be adaptable from one application to another.

This architecture has several key advantages, the primary being that the development cycle for these platforms can be quick due to the flexibility, code re-usability, and user familiarity of programming for GPPs. This is perfect for system prototyping where a rapid development cycle can greatly accelerate the process. One drawback can be the need for more powerful and less mobile processing machinery in order to implement more complicated waveforms. Typical waveforms implemented on GPP-based SDRs result in data rates that range from a few hundred kilobits per second up to a few megabits per second. There have been some implementations of wireless communication standards (802.11b 1 Mbps, 802.15.4 Zigbee). The SORA board has successfully implemented the 802.11g wireless standard up to 54 Mbps, however, it requires the computational resources of a dual-core 2.66 GHz processor to enable it [8].

2.1.2 FPGA-based

There are several well-established entrants in the FPGA-based SDR architecture. Particularly, we will mention research platforms. The Kansas University Agile Radio (KUAR) developed by the Information Technology and Telecommunications Center at Kansas University [9], the Wireless Open-Access Research Platform (WARP) developed by Rice University [10] [11], and the Japanese National Institute of Information and Communications Technology (NICT) SDR platform. This design architecture replaces the GPPs of the GPP-centric design with FPGAs to do almost all of the signal processing needed by the radio. Higher level logic for applications, network layer or medium access functions are often carried out by soft cores implemented in the FPGA fabric allowing these platforms to handle GPP-like branching logic.

The advantage of using FPGAs to implement the majority of the signal processing is that many of the DSP tasks scale well onto the FPGA, allowing the user to get the most out of non-standard data-types (12 or 14-bit ADC/DACs) and the inherent parallel processing ability of many DSP tasks (e.g. filtering). Additionally, these platforms may be much more power efficient, with reduced size and weight footprints, making them a better choice for mobile devices. Most of the SDR platforms that were built to implement current wireless standards use the FPGA-based design as they needed greater computational performance than could be offered by the GPP-based designs. Depending on the RF front-end hardware used, these FPGA-based platforms are often capable of fully

implementing many common wireless communication standards including 802.11a/b/g, WiMAX, wCDMA and more.

A disadvantage with this design is that the development cycle on FPGAs can be greater than on GPPs for an equivalent amount of code. Additionally, this code is more platform specific than GPP code would be.

2.1.3 Heterogeneous

In February 2011, Ettus Research announced the USRP E100 SDR platform. This platform is similar to its predecessors in that it uses an FPGA for high-speed signal processing close to the radio front-end. It then relies on an embedded GPP for other signal processing needs. In addition, it contains a fixed-point digital signal processing (DSP) chip which can be utilized for specific signal processing tasks. Another successful example of the heterogeneous architecture is the Lyrtech Small Form Factor (SFF) SDR. This product, composed of an ARM GPP, Texas Instrument C64x+ DSP, and Xilinx Virtex 4 FPGA, has been demonstrated to be capable of implementing many wireless standards. On these platforms the GPP and DSP are contained on a single die and can share a portion of the embedded processors random access memory (RAM). This close coupling of heterogeneous processors will allow both the DSP and GPP to be active components in the SDRs processing chain to maximize the advantages of each processor.

The hardware platforms listed as examples in this section are designed primarily for

research purposes. GPP-based platforms, because they depend on host computer's GPP, are not designed to meet the requirements of a specific communication standard. Instead, they are often designed to offer an easy development environment for waveform prototyping. The FPGA-based platforms, on the other hand, are often capable of implementing common communication standards and interacting with hardware radios. In addition to the examples given above, there are commercial SDR platforms (e.g. designed for use with military or public safety radios systems) that are available as well. Unsurprisingly, these radios commonly employ either FPGA-based or heterogeneous platforms in order to meet the requirements for the communication standards they are designed to support.

In this thesis we will be examining the heterogeneous architecture in more detail. In particular, we will look at how heterogeneous processing can be implemented on the Ettus Research E100 in such a manner that increases the platform's performance without increasing the difficulty of use.

2.2 SDR Software

There are many SDR software development packages available. There are software frameworks designed for amateur radio enthusiasts, commercial SDR application development and communications research. Often each SDR hardware platform has unique requirements and software tools. As mentioned previously, our goal is to explore heterogeneous processing in conjunction with component based design. As such, we evaluated software

frameworks that use the CBD architecture and are operable on our target hardware (USRP E100). The user community and licensing of the software were considered as well.

Of the several signal processing packages that are available, two popular packages have previously been implemented on embedded processors. Both GNU Radio and the Open Source SCA Implementation, Embedded (OSSIE) have been ported to the OMAP embedded processor found on the E100 SDR [12][13]. In this section we will discuss some specifics regarding these two software packages, the advantages and roadblocks presented by both, and finally the reasons for choosing GNU Radio as the development package.

2.2.1 OSSIE

OSSIE is an open-source SDR software platform, developed at Virginia Tech, based on the Software Communications Architecture (SCA). SCA was first described as a core part of the Joint Tactical Radio System (JTRS) program proposed by the United States Department of Defense [14].

One of the important aspects of the SCA is the use of middleware to manage much of the inter-component interaction (including data transfer) within the SDR. The middleware used in SCA (and OSSIE) is the Common Object Request Broker Architecture (CORBA) standard defined by the Object Management Group (OMG). CORBA performs all method calls and data-type consistency/translation between components ensuring

compatibility between code written in different languages or running on different operating systems or hardware. This is a valuable asset to the OSSIE framework as it allows relatively quick expansion of the OSSIE framework to distributed or heterogeneous systems.

Initial research into OSSIE as an avenue for heterogeneous processing showed that use of the CORBA middleware incorporates a per-component overhead that can quickly overwhelm the diminutive GPP on our embedded platform [15] [16]. To overcome this limitation several components can be grouped into a single larger component to save on overhead. However, this will negatively impact our component based design goal and possible code-reuse while increasing the development time for SDR applications.

2.2.2 GNU Radio

GNU Radio, also an open source signal processing software package, enjoys widespread popularity, a large contributing user base, and a relatively shallow learning curve. GNU Radio SDR applications, called flow graphs, are programmed in Python, however, the underlying signal processing is written in C++ for performance reasons.

GNU Radio is a prime example of component-based software design. Each signal processing task is a separate block, run in a separate thread, with the data flowing from block to block via shared memory buffers. Because of this fine granularity, almost any desired SDR application can be constructed from the more than one hundred blocks that

come with GNU Radio.

GNU Radio lacks a mechanism to integrate heterogeneous processing components into a flow graph. Lacking any middleware to manage components results in less inter-block overhead generated by GNU Radio as compared to OSSIE. However, in order to perform heterogeneous processing in GNU Radio, an interface for the heterogeneous component must be added to GNU Radio's architecture.

The factors that led to the selection of GNU Radio over OSSIE are based on the lighter program overhead, the expansive user community, and the close integration of GNU Radio and the Ettus Research product, USRP E100, that is the intended hardware platform for this thesis.

2.3 Heterogeneous Processing for SDR Platforms

On an SDR platform, because the signal processing is performed in software, certain capabilities of the radio (response times, modulation complexity) are limited by the performance of the computational elements of the radio. As communications standards (particularly mobile cellular radio) continue to push for increased radio capabilities, the demands on the processors of SDRs increase as well.

Even with the ever-increasing advances in processor power, the bottleneck for current GPP-based SDR platforms is often their processing capability. We will see that SDR

processing chains can have several similarities to high performance computing problems. As such, we will identify certain aspects of the SDR computational tasks that have a high likelihood of benefiting from using heterogeneous processing.

Within a common SDR processing chain we find several instances of tasks that are well-suited for GPPs, DSPs, and FPGAs. By fully exploiting the specific advantages of each processor, we can simultaneously reduce power consumption while increasing the capabilities of the radio.

Figure 2.2: BPSK signal processing tasks from GNU Radio flow graph

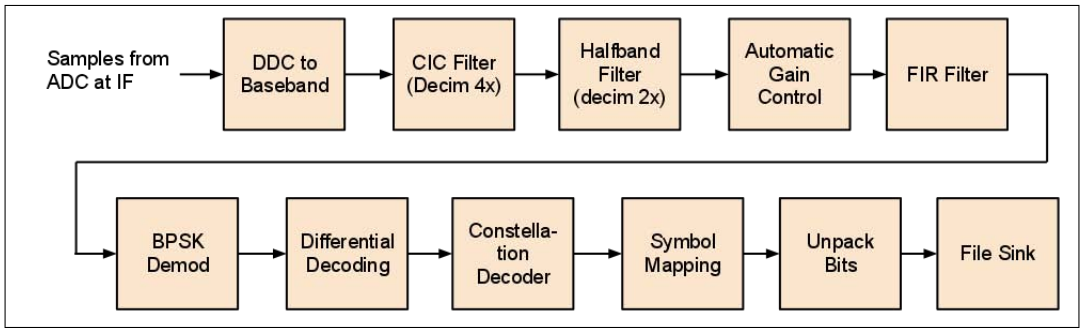


Figure 2.2 illustrates the components involved in receiving a binary phase shift keying (BPSK) signal. In this example, filtering, scaling, and demodulation are signal processing tasks that are commonly available in DSP libraries or FPGA logic. Specifically, an efficient, hand-tuned FIR filter method is provided by Texas Instruments for use on the DSP, while the FPGA logic for automatic gain control and interpolation/decimation are distributed with GNU Radio.

An additional boon for heterogeneous processing comes from the data-flow programming architecture utilized by many SDR software packages. This programming paradigm

represents the signal processing as a graph of connections between independent tasks and can be well-suited for parallel processing and heterogeneous processing. However, there are some additional concerns that a developer must consider when designing a heterogeneous processing system, in particular: data-type representation/consistency, data transfer, and concurrency.

2.3.1 Data-type Consistency

Any time multiple architectures, operating systems, or programs are used to manipulate a common set of data, data-type representation and consistency between processes is an important concern. A brief example of trade-offs that might be found when choosing data-types for SDR processing can be found by examining GNU Radio.

In GNU Radio data can be transfered from a USRP to the host computers GPP as either 16-bit fixed point or 32-bit floating-point values. While floating-point calculations offer better precision when compared to fixed-point calculations, the computational demand to perform floating-point arithmetic is greater than for fixed-point.

However, GNU Radio is typically run on desktop/laptop computers containing dual or quad-core CPUs, most of which will support floating-point SIMD instruction sets. Because of this, it can be advantageous to use floating point data-types for most of the signal processing. In fact, the majority of signal processing blocks in GNU Radio's library use floating point data-types.

On the embedded platform a SIMD floating point unit (FPU) is available, however, the DSP and FPGA use fixed-point representations. As such, a method for fixed-to-float conversion (and the reverse) is needed to interface between the GPP, DSP, and FPGA. This type conversion will incur an additional processing load and will need to be considered when deciding when to use the heterogeneous components for signal processing tasks.

2.3.2 Data Transfer

The efficiency of the data transfer mechanism in heterogeneous processing systems, particularly for real-time applications like SDRs, has a great impact on the usability of the heterogeneous services. As the time or computational processing required for data transfers increase, the amount of data processed per function call or the benefit derived from the heterogeneous element must likewise increase.

In the hardware platform that is explored in this thesis which uses the OMAP3530, the DSP and GPP share 128 MB of RAM but do not share caches. Memory is allocated for the DSP during system start-up and remains off-limits to the GPP. The processor manufacturers, Texas Instruments, provide a library called DSPLink for interfacing with the DSP from the GPP. Within this library there are several methods for GPP-DSP data transfer. An overview of the transfer methods and their performance is provided in [17].

2.3.3 Resource Mapping

Resource mapping is the task of matching up tasks and resources that may be shared between the processing elements in order to meet a particular goal (e.g. minimum execution time). This can apply to multi-core processors, distributed processing networks, or, as in our case, heterogeneous processors. Often there are trade-offs to be made, for instance between execution speed, memory size, latency requirements, and network loads. In the case of SDRs, additional requirements may be placed on the radio by the particular modulation/MAC schemes that are implemented (e.g. acknowledgment timeouts).

The resource mapping for heterogeneous processing is made somewhat easier with the data-flow programming paradigm (versus an interpretive or functional programming design) that is common in SDR software environments. This is a result of the fact that the programs are divided into many independent processing tasks. There are many approaches to resource mapping on heterogeneous systems. In Section 4.3 we will further describe the considerations of resource mapping on SDR platforms and outline the strategy we developed in this work.

2.4 Previous and Related Work

This thesis is not the first to explore heterogeneous processing for either the OMAP3530 system on a chip (SoC) or for the GNU Radio software platform. I will briefly describe

these previous implementations, their results, and my goals for specific improvements that can be made.

2.4.1 GNU Radio on Cell Broadband Engine

The Cell Broadband Engine (CBE) is a multi-core processing platform that combines one PowerPC processor (PPE) and eight to sixteen Synergistic Processor Cores (SPEs) which are capable of a combined 6.4×10^{12} single precision floating-point operations and half as many double precision operations per second. The cores are connected via a high-bandwidth bus interface.

In porting GNU Radio to run on the CBE [18], a new scheduling method was designed to replace the multi-threaded scheduler that resides in current implementations of GNU Radio. The PPE provides the overhead and schedule management aspects of the flow graph while the SPEs poll a queue of available tasks. When a SPE retrieves a task, it receives a task description, DMAs the data and arguments to its local memory, completes the task, and finally DMAs the data back to the PPE and signals job completion. This represents a significant departure from the scheduler used in a non-heterogeneous GNU Radio flow graph. Additionally, the requirements to utilize the DMA at maximum speeds require careful management of the data blocks that are transfered.

This method of heterogeneous processing allows near linear performance improvements when adding SPEs to a given task, up to the max of 16 SPEs on a IBM QS22 Cell

blade. This is an excellent example of the benefits that can be gained with heterogeneous processing. This thesis will attempt to realize similar gains without the changes to the core GNU Radio architecture/scheduler that users are familiar with or the limitations to data transfers or task definitions.

2.4.2 OSSIE on OMAP3530

In [13] we see an example of the OSSIE SDR software platform operating on the OMAP processor. As previously mentioned, OSSIE uses the CORBA middleware to interface between distributed or heterogeneous processors. The efficiency of this method has been described by [15] and [16] which demonstrate that the middleware introduces a processing overhead that increases as the number of components in the signal processing chain grows, regardless of whether or not they are spread over heterogeneous processors.

While OSSIE would likely require the least effort to implement a heterogeneous processing environment as the difficulties inherent with heterogeneous processing would be handled by CORBA, we will endeavor to create a framework that can avoid much of the inter-component overhead.

2.4.3 GNU Radio on OMAP3530

There have been two recent explorations of heterogeneous processing with the OMAP3530. These works are particularly relevant to this thesis as they provide the initial research into

heterogeneous processing on the hardware platform we have chosen for our research. One is Carlo Rinaldi's work [17] to evaluate the performance of various data transfer methods for heterogeneous processing on the OMAP processor using the Texas Instruments provided DSPLink data transfer architectures [19]. Another is the use of one of those transfer methods for heterogeneous processing with GNU Radio to implement an SDR performed by Al Fayez [20].

[17] describes the performance of each of the DSPLink data transfer mechanisms under various configurations. The GPP overhead, latency, and throughput of each method is detailed. [20] builds on this work by choosing the Channel (CHNL) DSPLink component to construct a heterogeneous interface. In both cases, the benefits of heterogeneous processing result in a significant performance increase. However, several limitations were identified. In particular, there is significant latency in the data transfer mechanism.

In this thesis we will be improving upon the data transfer mechanism–significantly expanding the utility of our heterogeneous processing implementation at the cost of moderately increased complexity. The data transfer mechanism described in this thesis avoids the overhead present in the DSPLink implementations by using shared memory between the GPP and DSP. This architecture allows faster transfer of data regardless of data block size. Section 3.2.2 goes into detail on this mechanism.

2.5 Hardware Setup

The hardware platform chosen for this thesis was the Ettus Research USRP E100. However, for the majority of the research process, the E100 was unavailable to the public. In its place, I utilized an OMAP3530 development board and used a USB 2.0 connection to interface with a USRP1. This setup allowed me to run GNU Radio with actual USRP data on the same processor that is now used in the E100. This ensured minimal changes required to employ the heterogeneous framework upon availability of the E100.

The OMAP3530 is comprised of three elements: an ARM Cortex-A8 CPU, a C64x+ fixed point DSP, and a Single Instruction Multiple Data (SIMD) floating point co-processor.

The ARM Cortex-A8 is a 600 MHz CPU with 256 MB of RAM and 256 MB of Flash memory. An SD card provides the storage memory. The TMS320C64x+ DSP is a 430 MHz fixed-point digital signal processor. It uses a 256 bit very long instruction word (VLIW) instruction set and 32 bit data word length. It is capable of performing four 16-bit multiply and accumulates (MACs) per cycle or up to eight 8-bit MACs.

2.6 Summary

In this chapter we have described three hardware architectures, identifying several strength and weakness of each. In particular, we note that heterogeneous SDR platforms have the capability, through the use of component based design, to bridge the gap between ease of

use and performance.

Additionally, we have reviewed two well-known SDR software frameworks and the attributes of the GNU Radio framework that led to its use in this thesis.

Finally, we have described previous work done in the area of heterogeneous processing for SDR platforms. The limitations of each is presented and from them we have defined specific goals for a new heterogeneous implementation.

Chapter 3

Implementation Method

Our goal was to create a method of heterogeneous processing for GNU Radio that would expand upon the work of [17] and [20], most notably, to overcome the limitations of the data transfer mechanisms employed therein. Additionally, care was taken to maintain the component-based design of GNU Radio and to introduce the use of heterogeneous processing elements in a way that would be essentially transparent to GNU Radio and the user.

There were three guiding principles while developing this heterogeneous processing architecture.

1. Open source software: As our desire is to expand the usability of current SDR platforms for universities, radio enthusiasts, and commercial organizations, open source software was chosen over proprietary when possible.

2. Transparency to GNU Radio: GNU Radio is actively in development and experiences frequent updates. In order to create a heterogeneous framework that is easily adoptable to current and future versions of GNU Radio, we carefully chose our implementation to be transparent to core GNU Radio code.

3. Transparency to users: To leverage the benefit of the large user-base of GNU Radio, the heterogeneous processing framework has the flow graph structure with which users are familiar. This is with the hope that it will increase use and encourage the community to maintain and improve upon this initial effort.

3.1 Development Environment

There are three areas of development for this project:

- Embedded platform
- DSPLink and DSP executable
- GNU Radio development

3.1.1 GPP Operating Environment

The operating system that is run on the embedded GPP is a Linux distribution for embedded processors called Angstrom [21]. It is distributed with all USRP E100s produced

by Ettus Research and as such is the de facto OS for that platform. There are two common methods of developing software for the embedded platform. It is possible to compile code natively on the embedded platform. However, because the resources (memory and processing power) of the embedded platform may be considerably more constrained than a typical desktop computer this can be a time consuming process. An alternative is to cross-compile code for the target architecture (ARM in this case) on a more powerful computer. In this thesis, all necessary dependencies were downloaded directly to the embedded platform and GNU Radio was compiled natively from the source available at [22]. This applies to all applications running on the GPP; as we will see in the next section, applications/libraries for the DSP must be cross-compiled.

3.1.2 DSP Libraries and Operating Environment

The TI C64x+ DSP that is part of the OMAP3530 system on a chip (SoC) runs a separate real-time operating system called DSP/BIOS maintained by Texas Instruments. Additionally, the drivers required by the GPP to interface with the DSP are contained in a library called DSPLink. Texas Instruments provides the tool chain and the libraries necessary to cross-compile code for the DSP and GPP side of applications. In this implementation the code necessary for both the GPP and DSP was written and compiled on a Linux computer before being transfered to the Beagleboard. For the GPP side, a library was created that included all of the DSPLink functions needed by our GNU Radio code.

Listed below are the software packages available from Texas Instruments for the GPP/DSP application development (available at [23]). This assumes that the development platform is running a Linux distribution. There are other software tools for developing on a Windows platform.

- DSP BIOS OS
- C6000 compiler
- DSPLink
- DSPLib (provides signal processing functions for the DSP)

3.1.3 GNU Radio

As previously mentioned, GNU Radio has a library of many signal processing blocks. The blocks, written in C++, are connected into flow graphs using Python. The C++ code offers greater speed for the computationally intensive processing of the blocks while Python allows easy reconfiguration of flow graphs without the need to recompile. The GNU Radio code developed for this thesis is written in C++ and NEON SIMD intrinsics. Additionally, it makes use of the DSPLink library mentioned previously. This code was written and compiled natively on the embedded processor making use of GCC 4.5 and Python 2.7.

3.2 Implementation Architecture Overview

The implementation can be thought of having four specific aspects:

- GNU Radio middleware block
- CMEM shared memory
- Control Array
- DSP program and functions

Figure 3.1: Heterogeneous Processing Architecture

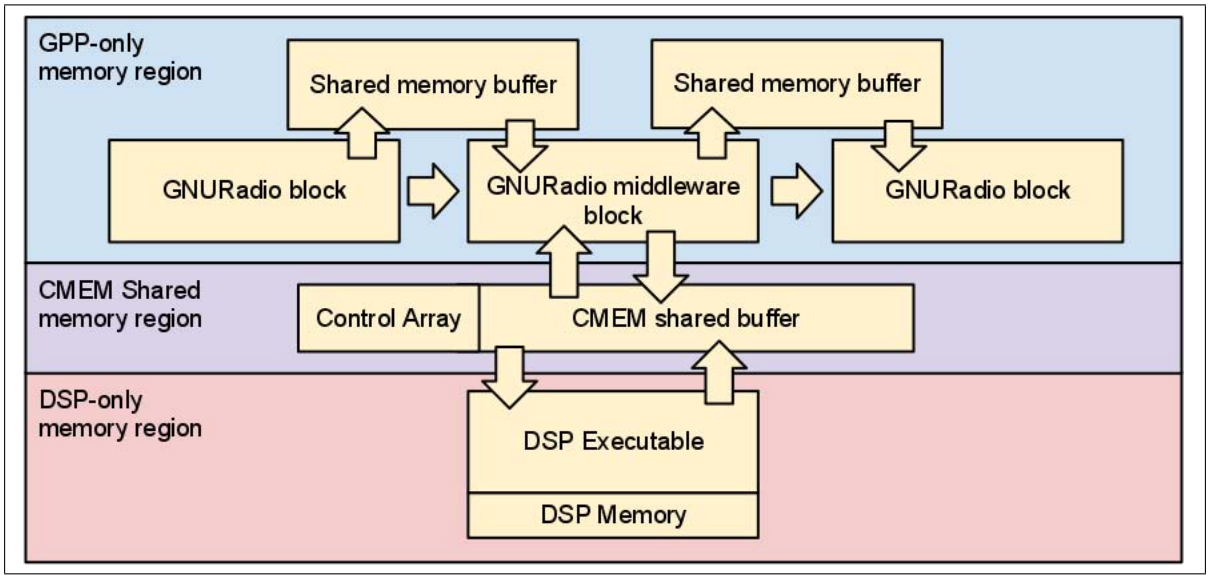


Figure 3.1 shows the lay-out of an SDR program's components within the embedded processor's architecture. From this top-level view, we can see the flow of data as it would be within an example GNU Radio flow graph. The data starts in a GPP memory buffer used by GNU Radio's current implementation. A GNU Radio block accesses the data and

makes a copy of the data into memory that is shared by the GPP and DSP. The GNU Radio block also transfers data from the DSP back to the GNU Radio buffers once processing is complete. The signal processing task required, the amount of data to be processed, and the source and destination buffer addresses are passed to the DSP via a Control Array. This control array exists in the same shared memory block dedicated to the GPP/DSP data transfer and is essential to the operation of the heterogeneous processing mechanism. The final element of this framework is the signal processing function that gets called on the DSP before the data is retrieved by the GNU Radio block. These components are explained next in greater detail.

3.2.1 GNU Radio Block

As mentioned before, GNU Radio employs a data-flow programming paradigm in which many independent signal processing blocks are connected by memory buffers. Two methods for interfacing the DSP into this design were considered.

- A middleware block which would accept data from the GNU Radio block(s) preceding it and then transfer the data to the DSP and retrieve it when processing was completed. Data transfer to be accomplished using the RingIO method as described in [17].
- A new shared memory buffer structure which would replace the current GNU Radio buffers.The buffer is designed to be accessible by both processors and the DSP

would have direct access to all data in the GNU Radio buffers. Shared buffers are created using CMEM, which will be described later.

Initial tests demonstrated that the shared memory buffer had significantly better performance compared to the middleware block design. This was due, in part, to avoiding the overhead associated with the additional copy of data from the GNU Radio block's input buffer to the DSPs input buffer, but the real speed increase was a result of avoiding the processing overhead that is part of DSPLink's data transfer functions.

However, one difficulty of shared memory buffers without a middleware block is managing control of the buffers that are shared between the GPP and DSP blocks. GNU Radio uses POSIX threads in order to safely share data between blocks that run in separate threads. Since POSIX threads are not currently available for the DSP a new mutex architecture would be required in the GNU Radio core scheduling code.

To avoid major changes to the GNU Radio core and to ensure a greater chance of acceptance into the GNU Radio code base, I decided to implement a combination of the two designs. This results in a GNU Radio block that acts as a middleware service while using a shared memory buffer for improved transfer speeds. This architecture offers a performance compromise slower than the direct shared buffers but significantly faster than the DSPLink methods (particularly for small bursts of data that are required for real-time radio applications). Also, it integrates seamlessly into the existing GNU Radio architecture.

The GNU Radio block performs the following tasks: establish shared memory buffers, receive task information, update the Control Array, execute data-type conversions (if necessary) and transfer data to DSP. Brief explanations of each task is provided.

Establish shared memory buffer

A GNU Radio middleware block is required for each GPP-to-DSP-to-GPP transition of data. This can occur multiple times within a flow graph. When a middleware block is instantiated it attempts to locate the shared memory block or creates it if not yet initialized. Additionally, it will create the Control Array and buffers, from the shared memory, that will be needed for the signal processing task that has been selected.

Receive task information

The data provided to the block are in the form of an array consisting of all tasks and arguments that will be transferred to the DSP. These entries are described in greater detail in Section 3.2.3.

Update control array

The Control Array is an array of 1024 integers and is used to pass information between the GPP and DSP. In particular, the Control Array provides the mechanism needed for managing the shared resources of this implementation. The operation of the Control Array

will be described in Section 3.2.3.

Data conversion and transfer

Based on the input data-type and the data-type required for the first DSP signal processing task, the middleware block may perform a data-type conversion. This data conversion and the transfer of data from the GPPs memory to the shared memory are performed by NEON instructions to reduce the processing overhead. The NEON technology allows 128-bit SIMD instructions. The data-type conversion that I implemented processes four 32-bit values per instruction. When the data are retrieved from the DSP a similar type conversion may be performed before placing the data into the outgoing GNU Radio buffer.

3.2.2 Shared memory

As mentioned earlier, in order to avoid the throughput bottleneck that is encountered when using DSPLink, a method to use shared memory between the DSP and GPP was designed. Making use of the TI Contiguous Memory Module (CMEM), a shared memory space was created that can be addressed by both the DSP and GPP.

The CMEM module is part of the Linux Utilities package developed and maintained by Texas Instruments. It requires setup prior to running the SDR application to ensure that the shared memory space is properly reserved. This module also provides various functions to manage the shared memory (e.g. allocations/de-allocation, determine the

virtual or physical address of memory blocks).

A user has flexibility to allocate different amounts, pool vs. heap memory, and memory locations as may be required by a specific SDR application. The shared memory space used throughout this implementation contains one memory pool of 4096 bytes and 10 megabytes of heap memory which may be allocated to buffers needed for signal processing.

3.2.3 Control Array

The Control Array is an array of integers that exists in shared memory. Accessible by both GPP and DSP, the primary purpose of the array is to pass control information between the two processors. This array is instantiated by the first middleware block created by GNU Radio and then shared with the DSP and other GPP middleware blocks that may occur later in the flow graph.

In the current implementation, when a task is assigned to the DSP, a corresponding entry is made into the control array using ten integers per task. Table 3.1 outlines the various fields that are assigned.

Task ID/Next Task ID

The DSP executable has a library of available tasks. When adding an entry for a specific task to the Control Array, the GNU Radio block will use the Task ID number related to the

Table 3.1: Control Array Entry

#	Field Description
1	Task ID
2	Next Task ID
3	Number of available samples
4	Input buffer address
5	Output buffer address
6	Size of one iteration
7	Argument 1
8	Argument 2
9	Address to Argument 3
10	Address to Argument 4

desired function in the library. If there are several signal processing tasks that are to be performed on the DSP in sequence, the 'Next Task ID' field is set to the ID of the following task. If the next task will be performed on the GPP, then this field is set to zero.

Number of available samples

The 'Number of available samples' entry is typically the only data field that is updated while the SDR application is running. This entry is designed to give a processor exclusive control over the data buffer. When the GPP transfers data to the input buffer it uses a positive value for the number of available samples. As long as the value stays positive, the GPP will not access the buffer. The positive value indicates to the DSP that data are

available for processing. When the DSP has finished the work and moved the data to the output buffer, it will change the 'Number of available samples' field to a negative number. In this state, the DSP will not access the data. This allows a single field to describe the ownership of the buffer and the quantity of samples available. This method does not allow the GPP to repeatedly add data to the DSP buffer. However, this typically does not limit the throughput of the flow graph because GNU Radio operates on all available data in a buffer, resulting in very few cases where the middleware block would repeatedly access the buffer.

Input/Output buffer addresses

When a middleware block is initialized it establishes data buffers of appropriate sizes according to the input and output tasks to be performed. The addresses of these buffers are then entered into the Control Array.

As mentioned before, we can construct flow graphs where a single middleware block will be managing data to/from a series of signal processing blocks that are performed sequentially by the DSP. In this case the middleware block will designate this by setting the input/output buffer addresses of intermediate tasks to zero. This instructs the DSP to create the needed intermediate buffers. Then the DSP will set the output buffer from 'upstream' tasks to be the input buffer for 'downstream' tasks until it reaches a task with a non-zero, GPP-given, output buffer address.

Arguments

There are four fields reserved for passing arguments to the DSP. Two of the fields are for simple arguments that would only require a single integer (e.g. a scaling factor). Two fields are designed to hold pointers to arrays of arguments (e.g. filter tap coefficients).

It is important to also observe that the Control Array has the ability to be updated during run-time. This gives the heterogeneous processing system an advantage over the current implementation of GNU Radio. It is possible to quickly and seamlessly change arguments for DSP tasks (e.g. filter coefficients, scaling parameters) or even add or remove DSP tasks from the current GNU Radio flow graph during operation.

3.2.4 DSP Program and Functions

Control of the DSP is provided via DSPLink process (PROC) commands which allow the user to load programs onto the DSP and start/stop execution. These commands are incorporated in a DSPLink library and are called by the GNU Radio middleware block. When the first middleware block is instantiated, it will transfer a DSP executable binary from the GPP memory to the DSP memory and execute it.

The structure of the DSP executable is outlined in the following pseudo-code.

```
Initialization
Retrieve Control Array address from GPP
Iterate through Control Array to initialize DSP-side buffers
While(1){
    For all tasks in Control Array{
        If task has available work{
            do work;
            move data to output buffer
        }
    }
}
```

This code causes the DSP's processor to go to 100% as it is constantly searching the Control Array for available work. This method of searching for available tasks was used because it is able to quickly identify available tasks thus introducing the least delay into the data transfer. Additionally, since the tasks listed in the Control Array are in order of how they are performed in the flow graph, any sequential tasks that are to be performed on the DSP will be performed in order and returned to the GPP expeditiously. This limits the time that the GPP might be waiting on data from the DSP.

One of the parameters in the Control Array is the signal processing task identification number. This number is used by the DSP to look up what particular signal processing function it will perform on the input data. Since DSP code must be cross-compiled on a different computer it makes sense to include a substantial library of common functions versus recompiling the DSP executable for each SDR application. With such a library distributed alongside GNU Radio code, most users would never need to install the DSP tool chains or recompile any of the DSP code and would still receive the benefits of the

heterogeneous processing capability. Throughout this research the DSP functions that were implemented were part of the DSPLib package provided by Texas Instruments. The following functions were implemented during the course of this research: Complex FFT, Complex FIR filter, and scale by an integer.

3.3 Performance Results

To measure the performance of this heterogeneous implementation, I designed a simple flow graph that would take a large array of input values, perform a Fast Fourier Transform (FFT), then discard the values into a null sink. Additionally, a GNU Radio block was created to record the number of samples that are operated on by the flow graph for metric gathering purposes.

Three variations of this flow graph were compared.

- The first uses only GPP-based GNU Radio blocks to give us baseline performance data. In this flow graph, the FFT algorithm is provided by the Fastest Fourier Transform in the West (FFTW) library (which is integrated into GNU Radio) [24].
- The second variation is a heterogeneous processing approach similar to that seen in [20] and [17] and represents the previous heterogeneous implementations on this embedded platform. It uses the DSPLink RingIO method for data transfer between the GPP and DSP.

- The final variation is the implementation described in this thesis. It uses a middleware GNU Radio block and CMEM shared memory for the data transfer.

Figure 3.2: Time per FFT Iteration vs. FFT Size

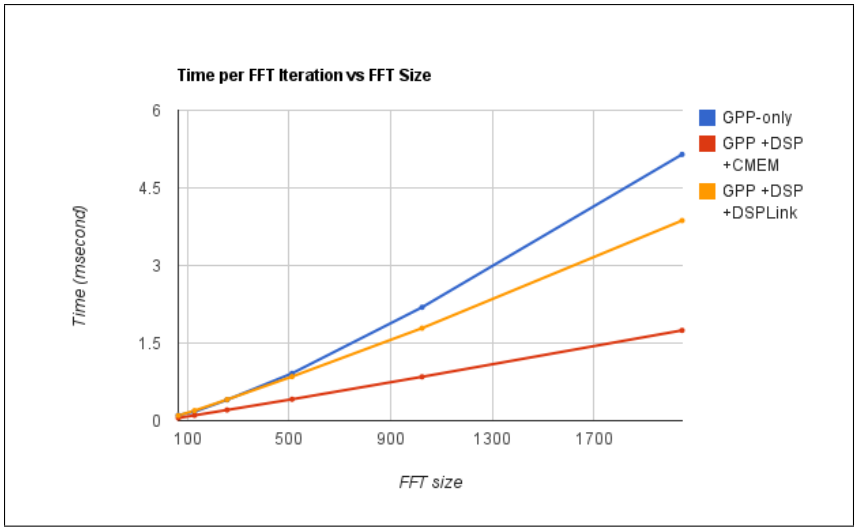
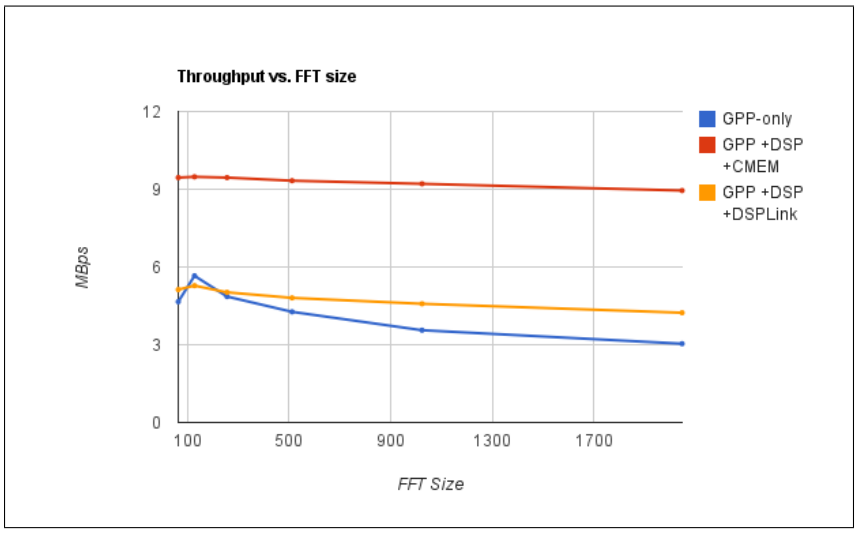


Figure 3.3: Throughput vs. FFT Size



In Figure 3.2 we see a comparison of the time required to complete one FFT on a group of sample data versus the size of the FFTs. We can see that both heterogeneous processing implementations outperform the GPP-only solution. The DSPLink variation pro-

vides a slight improvement, while the implementation described in this thesis (denoted as GPP+DSP+CMEM) achieves a 2-3x speed up. It should be noted that the DSPLink variation is only able to show a performance increase when it operates on large blocks of data, performing many FFTs before returning the data to the GPP. This is a hindrance in SDR applications where using large data blocks results in increased latency from the receiver to the processor.

Figure 3.3 compares the throughput of the GPP-only and the heterogeneous method described in this thesis. Both methods have decreasing throughput as the FFT size increases. The GPP implementation throughput decreases quickest because the GPP workload grows at approximately $N \times log(N)$ where N is the FFT size. The heterogeneous implementation however, does not decrease as quickly because the increased workload on the DSP happens concurrently with other GPP signal processing. For the FFT sizes tested here, the DSP is able to perform the FFT before the GPP is ready to retrieve the data.

Chapter 4

Performance Prediction and Resource Mapping

When using heterogeneous processing, developers must carefully divide the computational tasks to take advantage of each processing element. The matching of processing tasks to heterogeneous resources is a computationally difficult problem that has been explored for many years. There are numerous approaches to automatic resource mapping for heterogeneous systems, covering a variety of system architectures.

The resource mapping described in this chapter is designed to allow a GNU Radio developer to include heterogeneous elements in a flow graph without needing to hand-tune the placement of tasks to achieve optimal throughput. Particularly, we implement an off-line (performed prior to execution of the flow graph) search method that seeks to

identify an optimal or near-optimal flow graph that balances the processing load between the GPP and DSP.

4.1 Performance Measurement Data

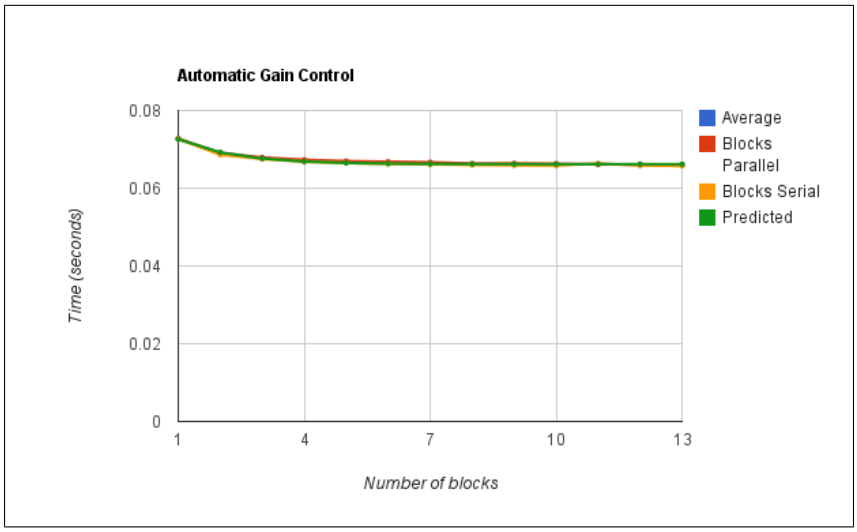
Any off-line search mechanism for resource mapping requires data that accurately describes the performance of the target applications. Typically these algorithms use execution speed information to find a mapping that will minimize the execution time of the application. However, as most SDR applications operate on continuous streams of data, our goal is better described as maximizing throughput of the flow graph.

To compare the performance of a specific task on heterogeneous components a common measurement for performance is defined. As stated before, SDR applications will commonly not have a set termination point. Therefore instead of execution time, time to process one block of data is measured and used as the performance metric. While the selection is arbitrary, one million samples (Msample) is used as the block size for all measurements in this thesis.

The performance prediction and resource mapping methods are intended to be architecture independent. There are some functions within the GNU Radio library that have architecture-dependent implementations (e.g. the FFTW library) [25] [26]. As such, performance data is gathered empirically. This relieves the user of the profiling task.

Scripts were developed that construct and execute GNU Radio flow graphs designed to collect throughput (samples/second) data. To gather performance data for a particular block, the script creates and tests ten flow graphs with various configurations of the block in series and parallel. With the measurements, a decaying exponential is fit to describe the contribution of this particular block and GNU Radio overhead to an arbitrary flow graph. Profiling data was collected for seven GPP-based GNU Radio blocks.

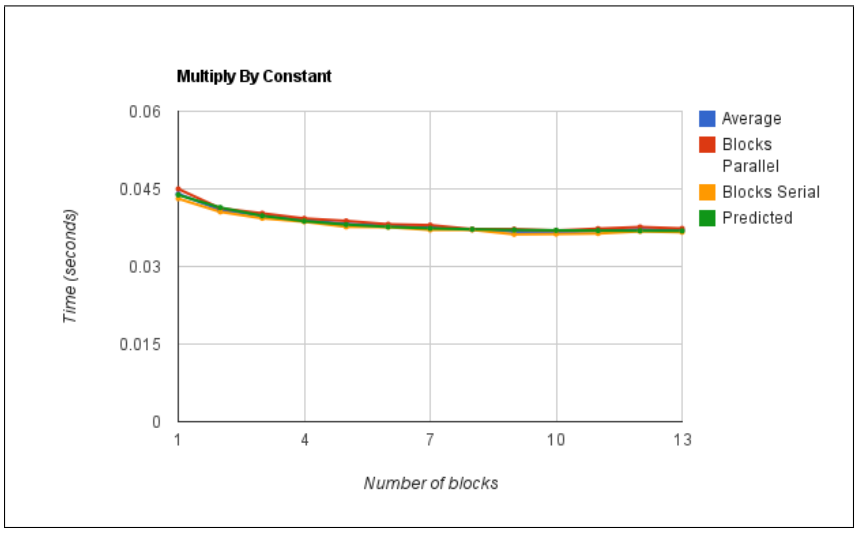
Figure 4.1: Time per Msample for Automatic Gain Control Block



Figures 4.1 and 4.2 show the performance data for two GNU Radio blocks. These graphs show how the time to process data changes based on the number of blocks in the flow graph. Of the blocks that were profiled, most fit an exponential decay model. However, for some, the performance is less predictable. The variations are likely due to interactions between buffer sizes, CPU caches, and the size of data per iterations.

The signal processing on the DSP must also be profiled in order to predict performance for heterogeneous flow graphs. The signal processing library, DSPLib, from which

Figure 4.2: Time per Msample for Multiplier Block



the functions used in this thesis are called was profiled by Texas Instruments. This provides the data needed for performance prediction. As development of this prediction method continues, a DSP-evaluating flow graph/GNU Radio block should be created to allow profiling of custom DSP code.

4.2 Prediction Method and Results

The prediction of performance is relatively straight forward: First, the total number of blocks present in the flow graph is calculated (n). The time-cost (total time to process one Msample), a function that depends on the length of the flow graph, is calculated for each individual block using the collected performance data. Finally, the time-costs for all blocks are summed to provide us with the total time to process a million samples through the entire flow graph.

$$Time/Msample = \sum_{i=0}^{n} cost_i(n)$$

To predict the performance of a flow graph in which a signal processing task is performed on the DSP we evaluate the performance of the GPP and the DSP separately. The GNU Radio middleware block that manages the data going to-and-from the DSP has been profiled and contributes to the GPP processing time. However, the time taken by the DSP to perform its task does not contribute to the GPP's processing time and it is concurrent with GPP processing. As such, the total processing time per Msample of a heterogeneous processing flow graph can be found by evaluating the processing time of the GPP and the DSP separately. The processor with the longest processing time determines the processing time for the flow graph.

Table 4.1: Performance Prediction Results

# Flowgraph Blocks	% Error from Measured Results
2	1.67%
3	1.52%
3	-1.76%
3	-3.07%
6	3.82%
6	5.12%
10	-7.78%

Using the seven GPP blocks that were profiled in the previous section, several test flow graphs were designed to verify the prediction method. Table 4.1 shows the predicted performance is within 8% of actual performance for the worst case but is often within 3% for smaller flow graphs. These flow graphs were composed of a single linear chain of blocks with no parallel branches. Additionally, each block produced an equal number of samples as it consumes. Further research will be needed to account for blocks that consume/produce different numbers of samples.

These results give us reasonable confidence in the ability to accurately predict the performance of an arbitrary flow graph. Now that performance prediction (cost function) has been established, we can develop the optimization strategy.

4.3 Optimization Strategy

Using the heterogeneous processing framework that has been defined in this thesis, a user may create a flow graph that contains several blocks that can be run on either the GPP or the DSP. Instead of forcing the user to discover the optimal placement of those blocks, we have developed a resource mapping algorithm to do this automatically.

As mentioned earlier, the metric we are using to track the performance of a flow graph is time per Msample. We will be using this metric again in the optimization algorithm. In particular, we are searching for the flow graph in which the processing element (GPP or DSP) that has the longest processing time, is shorter than all other flow graph variations.

In other words, we are searching for the $Min(Max(COST_{GPP}, COST_{DSP}))$. Because the DSP and GPP can operate simultaneously on data, once the pipeline of signal processing blocks becomes filled with data, only the processor with the heaviest processing load will determine the overall time per Msample.

An iterative search for the least-cost flow graph results in 2^N calls to the cost function (where N is the number of possible heterogeneous processes). This is the worst case search method considered by this thesis in this problem set. For present day SDR applications, often containing only a few signal processing tasks that are available for heterogeneous processing, this search method is simple and complete.

However, we will explore another optimization search method. Two implementation specific aspects will influence our optimization strategy. First, the variation in processing cost for a given task on the heterogeneous processors. There are cases in which the DSP is faster, slower, or equal to the GPP. Additionally, there are some blocks that have not been implemented on the DSP. Second, the communication costs between processors on the hardware platform must be accounted for. With these considerations, a resource mapping algorithm motivated by prior work is developed.

In [27] the effect of communication link costs is described and a mapping algorithm that clusters tasks on processors in order to minimize communication costs is analyzed. Intuitively, as communication costs increase, this type of mapping quickly outperforms one that ignores the inter-processor communications. On the hardware platform chosen for this thesis, there is a communication cost of GPP overhead and latency when data

is transferred from the GPP to the DSP as well as when it is retrieved. To minimize the communication cost, a clustering of GPP-only (non-heterogeneous) and GPP/DSP (heterogeneous) blocks will be made. In particular, all GPP-only blocks are considered one cluster, while sequential chains of one or more GPP/DSP blocks are separate clusters. This will be described in greater detail later.

In [28], the performance of various mapping algorithms for heterogeneous systems is compared. The algorithms were ranked by their performance both in closeness to the optimal solution and in solving time. The algorithm with the best average performance was a Genetic Algorithm (GA). However, the GA in this study was seeded with already near-optimal results from a Min-min algorithm. The Min-min algorithm produced results that were within 10% of the best results of the GA and with an average solving time orders of magnitude faster than the GA. A brief description of the Min-min algorithm is as follows.

The Min-min algorithm starts with a list of unassigned tasks (U) and the current processing load for each processor (M). The processing loads (current + new task) for all tasks in U are identified. Next, the task/processor combination that results in the overall minimum processing load is selected and the task assigned to that processor. The processor load list, M, is updated and the task is removed from list U. The goal of the Min-min algorithm is to assign tasks in the order that changes the processing loads by the least of any available. This should result in the pairing of tasks to processors that are best suited for that task the majority of the time.

Applying the clustering and Min-min algorithms to the SDR application happens in four steps.

1. Partitioning the clusters: In order to minimize the communication overhead, the number of transitions from GPP-to-DSP and the reverse are minimized. This is done by combining any sequential chain of GPP/DSP blocks into a cluster.

2. Non-heterogeneous loading: The GPP-only blocks are assigned to the GPP processor and the total processing load for these blocks is computed.

3. Min-min load balancing (clusters): The GPP/DSP clusters are balanced based on the Min-min method described above, i.e. the processing loads for each cluster are computed for both GPP and DSP, then the current loads of the GPP and DSP are added to each and the cluster that results in the smallest total processing load on either GPP or DSP is then assigned to that processor. Because the clusters are chosen to be as large as possible in step 1, this may lead to an unbalanced distribution by the Min-min algorithm.

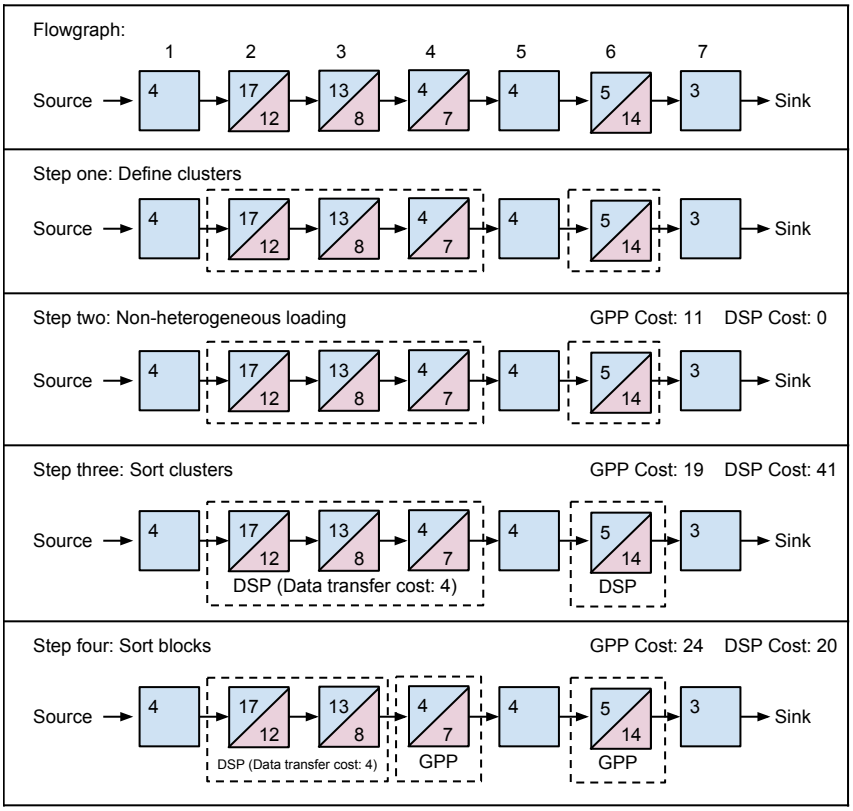
4. Min-min load balancing (blocks): A second round of Min-min balancing is done, this time on the blocks that are the beginning and ends of the heterogeneous clusters. Changing the placement of one of these blocks will not increase the communication overhead whereas a block in the middle of a heterogeneous cluster would incur an additional round-trip cost. The edge block with the smallest difference between processor costs (GPP vs. DSP) and which will result in a lower

$Max(COST_{GPP}, COST_{DSP})$ is swapped. This is repeated until no further improvement can be made.

4.3.1 Optimization Example

A brief example of the optimization process is presented in Figure 4.3 and described below.

Figure 4.3: Optimization Example



In the top row of Figure 4.3 an example flow graph containing seven signal processing blocks is presented. Of the seven blocks, four are able to be implemented on both a DSP and GPP (indicated by the block being divided). Example processing costs have been

assigned to each block (numbers in upper-left and lower-right of each block). Costs in the upper-left represent GPP-costs, lower-right are DSP-costs.

Step One: Define clusters

If several blocks can be performed on the DSP sequentially there is no need for additional data transfers between each block. This significantly reduces overhead within the heterogeneous flow graph and as such any groups of GPP/DSP capable blocks are formed into clusters. In the example, we have two clusters. Blocks 2, 3, and 4 will be Cluster 1. Block 6 is Cluster 2.

Step Two: Non-heterogeneous loading

Since there are some blocks that are only implemented on the GPP, we need to identify them first and calculate their processing cost. In the example, blocks 1, 5, and 7 are non-heterogeneous and have a total GPP-cost of 11.

Step Three: Sort clusters

Using the Min-min approach to sort the clusters, we first calculate the processing costs of each cluster for each processor. Cluster 1 (blocks 2, 3, 4) has the following costs: GPP-34 , DSP-4:27. Cluster 2 (block 6) has the following costs: GPP-5 , DSP-4:14. (Note: For the DSP cost we have two numbers. The first is the cost to the GPP to perform data transfer. The second is the cost of performing the signal processing on the DSP). Next, we add the current GPP and DSP costs (from clusters that have already been mapped). This results in the following cluster costs: Cluster 1: GPP-45 , DSP-15:27, Cluster 2: GPP-39 , DSP-15:14. From these two options we select the minimum cost mapping of these possible cluster

mappings. In the example it is the DSP mapping for Cluster 2. Once this is applied we have a new overall cost of GPP-15, DSP-14.

Repeating this step, we once again add the current GPP and DSP costs to Cluster 1 revealing the following costs for its possible mappings: GPP-49:14, DSP-19:41. Again, the minimum cost is found by mapping Cluster 1 to the DSP.

Step Four: Sort blocks

In this step we will evaluate if there is further benefit by moving blocks that are at the beginning or end of each cluster. In the example, we will consider blocks 2, 4, and 6. The costs for each mapping of the blocks are calculated. The block that results in a overall cost decrease is re-mapped.

In the example flow graph the three end blocks have the following costs:

Block 2 GPP-19:41, DSP-36:29

Block 4 GPP-19:41, DSP-19:34

Block 6 GPP-19:41, DSP-20:27

In this case, moving Block 6 to the GPP results in a minimum overall cost of GPP-20, DSP-27.

Repeating this step again for blocks 2 and 4. We will find that re-mapping Block 4 to the GPP results in a better mapping with costs GPP-24, DSP-20. Reviewing the end blocks one more time reveals that no improvement can be made to this mapping and so the optimization process ends.

4.4 Optimization Results

The heterogeneous processing implementation described in Chapter 3 contains only three signal processing tasks that can be run in a heterogeneous manner. With so few heterogeneous blocks, optimization of a flow graph is trivial. As such, for this section we will be testing our optimization algorithm using simulated block performance data.

The results of this optimization algorithm are compared against an iterative search algorithm. As we know, the iterative search method requires 2^N (N is the number of heterogeneous blocks) calls to the cost-function and it identifies the optimal solution. The optimization algorithm outlined in this thesis, however, requires a maximum of 2N, and finds a near-optimal solution. To evaluate the average solution time and the error compared to the iterative solution, 100 iterations of flow graphs from size N = 5 to 30 are tested. In these flow graphs 50% of the blocks are heterogeneous-capable (randomly distributed throughout the flow graph). Similar to Section 4.2, the flow graphs used in this test were composed of a single linear chain of signal processing blocks, each with a 1:1 data consumed/produced ratio.

Figures 4.4 and 4.5 show the solution time for the various sizes of flow graphs. As expected, the iterative search has a exponential increasing solution time, whereas the cluster-based Min-min search results in an average solution time that is linear to the number of heterogeneous blocks in the flow graph.

As seen in Table 4.2 the cluster-based Min-min algorithm has a good average solution

Figure 4.4: Time to Solve using Iterative Search

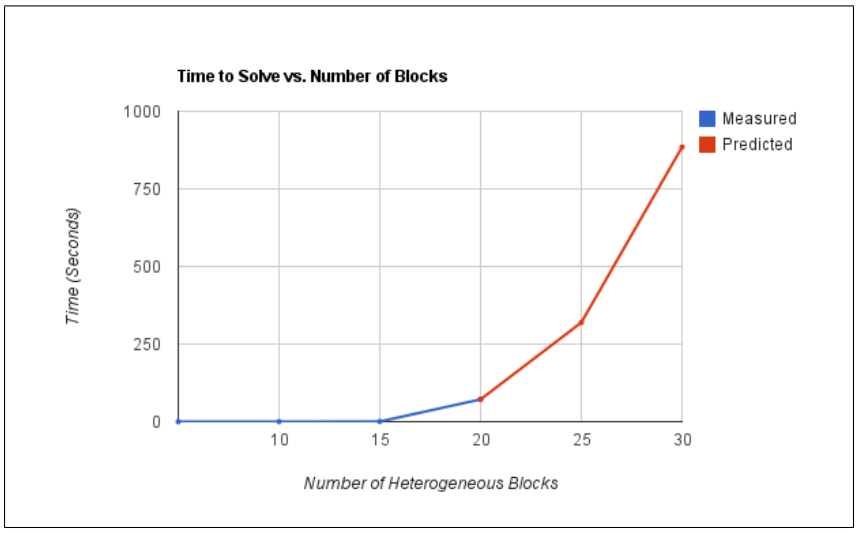
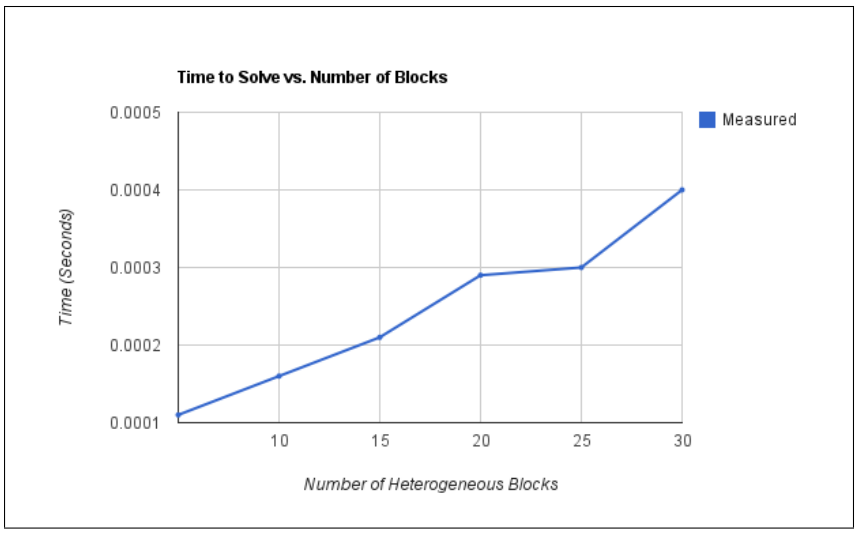


Figure 4.5: Time to Solve using Clustering/Min-min Search



performance, however, there are cases when the maximum error is considerably greater than the optimal solution. This occurs when a cluster of heterogeneous blocks contain a sequence of low processing-cost blocks with higher processing-cost blocks on either side. A result of the partitioning phase of this algorithm, which in this implementation is relatively simplistic, is that all heterogeneous-capable blocks are clustered together while

Table 4.2: Average and Maximum Error

# Heterogeneous Blocks	Average Error	Maximum Error
5	3.47%	28.5%
10	1.95%	10.8%
15	0.75%	3.8%
20	1.9%	2.3%

all non-heterogeneous blocks compose the other clusters. When the second round of Min-min balancing takes place, the only options are changing the placement of a block with large processing-cost when a more optimal choice may be to move one of the middle blocks and accept the additional communication overhead. If the partitioning phase was improved to create clusters not only based on inter-block communication costs but also by processing costs then the Min-min algorithm would more appropriately identify blocks that balance the work load.

4.4.1 Optimization conclusion

Using the Clustering/Min-min method, we have found that we can consistently identify a solution within 5% of optimal (achieving optimal resource mapping approximately 70% of the time) for relatively simple, non-branching flow graphs. Additionally, execution time of the Clustering/Min-min algorithm remains linear as the number of heterogeneous blocks increase.

Because of the short solution time for our resource mapping algorithm, it can be integrated into GNU Radio as a optimization aid for users. This would allow the user to specify a flow graph containing heterogeneous blocks that would be mapped to resources at runtime without user input.

Chapter 5

Conclusion

5.1 Conclusion

This research consisted of two major focuses: designing and implementing a heterogeneous processing framework for GNU Radio and developing an optimization strategy for resource mapping within heterogeneous SDR platforms. Within these areas, goals were defined based on specific architecture requirements and to promote the intended use within the SDR community.

Through analysis of current SDR hardware and software platforms the motivation and opportunity for heterogeneous SDR platforms has been presented. Further exploration of past heterogeneous computing systems identified specific considerations and goals for our implementation. A novel heterogeneous framework consisting of a GNU

Radio middleware block, shared memory data transfer, and a DSP signal processing library was described and evaluated. This design provides substantial performance improvement without changing any of the core GNU Radio code. The code for this framework will be contributed to the GNU Radio community so it may be refined and expanded upon.

In parallel with the heterogeneous framework, a method for optimizing the heterogeneous resources has been explored and presented. While the results are currently based on simulation, they present a compelling area for further research.

5.2 Future Work

The concepts and goals in this thesis were selected and carefully implemented in a manner that would promote the use of heterogeneous processing in the GNU Radio SDR platform. It is my hope that I have demonstrated the benefits and possibilities that heterogeneous processing can open in the realm of SDR applications.

There are many areas of future expansion and exploration that can build on this research. A few areas are listed below:

- Expanding the DSP function library.
- Integrate the FPGA as a component (not just a source/sink) in the flow graph.
- Refine the partitioning process. The current optimization algorithm groups all con-

secutive GPP/DSP blocks into a single cluster. This sometimes results in sub-optimal resource mapping. An alternative partitioning process that takes into account both communication and processing costs should result in improved resource mapping solutions.

Bibliography

[1] "What is software defined radio," 2008. [Online]. Available: http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf

[2] M. U. et al., "SDR comes of age: Technology." Washington, DC, USA: SDR09 Technical Conference and Product Exposition, 2009.

[3] V. Natoli. Heterogeneous processing: Trite or trend? [Online]. Available: http://www.hpcwire.com/hpcwire/2009-06-24/heterogeneous_processing_trite_or_trend.html

[4] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall PTR, 2002, iSBN 0-13-081158-0.

[5] Microsoft Research, "Microsoft research: Software radio," 2009. [Online]. Available: http://research.microsoft.com/en-us/projects/sora/

[6] "Ettus research: Products," 2012. [Online]. Available: https://www.ettus.com/product

[7] "Funcube dongle products," 2012. [Online]. Available: http://www.funcubedongle.com/?page_id=286

[8] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "SORA: high-performance software radio using general-purpose multi-core processors," *Commun. ACM*, vol. 54, pp. 99–107, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1866739.1866760

[9] G. Minden et al., "KUAR: A Flexible Software-Defined Radio Development Platform," *Proc. IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, 2007.

[10] "WARP, a Unified Wireless Network Testbed for Education and Research," *Microelectronic Systems Education*, pp. 53–54, 2007.

[11] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly, "WARP: a flexible platform for clean-slate wireless medium access protocol design," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 12, pp. 56–58, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1374512.1374532

[12] P. Balister, "Building GNU Radio on the BeagleBoard," 2010. [Online]. Available: http://www.opensdr.com/node/17

[13] ——, "OSSIE-SCA on the Gumstix Overo," 2008. [Online]. Available: http://opensdr.com/node/11

[14] D. Stephens, B. Salisbury, and K. Richardson, "JTRS Infrastructure Architecture and Standards," 2006. [Online]. Available: http://www.public.navy.mil/jpeojtrs/sca/Pages/sca.aspx

[15] G. Abgrall et al., "Predictibility of Inter-component latency in a Software Communications Architecture Operating Environment," *Parallel and Distributed Processing, Workshops and Phd Forum*, pp. 1–8, 2010. [Online]. Available: www-labsticc.univ-ubs.fr/~gogniat/papers/abgrallipdpsw2010.pdf

[16] T. Ulversoy and J. Neset, "On Workload in an SCA-Based System, with Varying Component and Data Packet Sizes," 2008. [Online]. Available: http://ftp.rta.nato.int/public/PubFullText/RTO/MP/RTO-MP-IST-083/MP-IST-083-03.doc

[17] C. Rinaldi, "Performance evaluation and optimization of an OMAP platform for embedded SDR systems," Master's thesis, Royal Institute of Technology, 2011.

[18] E. Blossom, "Gcell an spe scheduler and asynchronous rpc mechanism for the cell broadband engine," SDR 08 Technical Conference and product Exposition. SDR Forum, 2008. [Online]. Available: http://comsec.com/papers/gcell-sdrf-2008.pdf

[19] Texas Instruments, "DSPLink Overview," 2011. [Online]. Available: http://processors.wiki.ti.com/index.php/DSPLink_Overview

[20] A. Fayez, "Designing a Software Defined Radio to Run on a Heterogeneous Processor," Master's thesis, Virginia Tech, 2011.

[21] "Angstrom Linux." [Online]. Available: http://www.angstrom-distribution.org/

[22] "GNU Radio Source." [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/Download

[23] Texas Instruments, "DSPLink Build Instructions," 2011. [Online]. Available: http://processors.wiki.ti.com/index.php/Building_DSPLink

[24] M. Frigo and S. Johnson, "FFTW Website," 2011. [Online]. Available: http://www.fftw.org/

[25] R. Vuduc and J. Demmel, "Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW Position Paper," in *Semantics, Applications, and Implementation of Program Generation*, ser. Lecture Notes in Computer Science, W. Taha, Ed. Springer Berlin / Heidelberg, 2000, vol. 1924, pp. 190–211. [Online]. Available: http://dx.doi.org/10.1007/3-540-45350-4_14

[26] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," *Acoustics, Speech and Signal Processing*, vol. 3, pp. 1381–1384, 1998.

[27] K. Taura and A. Chien, "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources," vol. 2000. Cancun, Mexico: Heterogeneous Computing Workshop, 2002, pp. 102–115. [Online]. Available: www.hipersoft.rice.edu/grads/publications/taura-00-heuristic.pdf

[28] T. Braun et al., "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810–837, 2001. [Online]. Available: www.engr.colostate.edu/~hj/journals/71.pdf