Reconfigurable SCA System Development Using Encapsulated Waveform Applications and Components

Andrew Cormier

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

Dr. W. H. Tranter, Chair

Dr. C. B. Dietrich, Co-Chair

Dr. J. H. Reed

March 14, 2008

Blacksburg, Virginia

Keywords: Wireless, Software Defined Radio, Reconfigurable Radio, SCA, Aggregate Application, OSSIE

Reconfigurable SCA System Development Using Encapsulated Waveform Applications and Components

Andrew Cormier

ABSTRACT

The Software Communications Architecture (SCA) is a standard for software defined radios (SDR) designed in part to promote code reuse for long-term development. With the emergence of adaptive/cognitive radios, new SDRs that are capable of reconfiguration during runtime must be developed. One advantage of SDR development over conventional radio development can be ease of design if the proper rapid development tools are made available.

This thesis explores tools designed to help realize the construction of reconfigurable systems while promoting code-reuse within the bounds of the SCA. Developing these tools requires an understanding of the SCA as well as the Open Source SCA Implementation Embedded (OSSIE) for which they are developed. The use of CORBA to link together modularized components is also discussed. Finally, several simulations are conducted in order to approximate the amount of overhead resulting from the use of the reconfiguration tool developed (the "Connect Tool").

The Hokie Nation embraces our own and reaches out with open heart and hands to those who offer their hearts and minds. We are strong, and brave, and innocent, and unafraid. We are better than we think and not quite what we want to be. We are alive to the imaginations and the possibilities. We will continue to invent the future through our blood and tears and through all our sadness.

We are the Hokies.

We will prevail.

We will prevail.

We will prevail.

We are Virginia Tech.

-Nikki Giovanni

Acknowledgements

First of all, I would like to thank my wife, Melissa. She has challenged me to become who I am now and to accomplish what I have accomplished. She has kept me together through one of the hardest times in my life. She has taught me the meaning of happiness.

I would like to thank the members of my advisory board (Dr. Tranter, Dr. Dietrich, and Dr. Reed) for having faith in me and for helping me make the most of my educational experience. I offer extra thanks to Dr. Dietrich for all of the time he has dedicated towards me and towards my research.

I would like to thank several members of Wireless at Virginia Tech: Carlos Aguayo, Philip Balister, Brian Crosby, Joseph Gaeddert, Ben Hilburn, Dr. Brent Ledvina, Christopher Phelps, Shereef Sayed, Tom Tsou, and Haris Volos. They have all been tremendously helpful to me throughout my graduate studies.

I would also like to thank two older members of the OSSIE team: Max Robert and Jacob DePriest. Perhaps unknowingly, they set the foundation for my work, making all of this possible.

Last but not least I would like to thank my parents, Bob and Kay, my brother, Chris, and my parents in-law, Benny and Linda Penley for their endless confidence and support.

Grant Information

This work was supported in part by the National Science Foundation under Grant No. 0520418. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Also, this work was supported in part by SAIC.

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 3

1.3 Thesis Organization 5

2 Background 6

2.1 SCA 6

2.1.1 Waveforms, Waveform Applications, and (SCA) Applications . . 7

2.1.2 Components 8

2.1.3 Uses and Provides Ports 9

2.1.4 CORBA 10

2.2 OSSIE 10

2.2.1 The OSSIE Waveform Developer 11

2.2.2 ALF 12

2.3 Related Work 15

2.3.1 The Aggregate Application Concept 15

3. (SCA) Application Management 18

3.1 Methods for Managing (SCA) Applications 18

3.2 The Use of **omniORB** 20

3.2.1 Obtaining an ORB and a Root Context 21

3.3 Installing Waveform Applications as (SCA) Applications 22

3.3.1 Uninstalling Waveform Applications 24

3.4 Connecting Ports 24

3.4.1 Retrieving a Port Handle from the Naming Service 25

3.4.2 Finalizing the Connection 25

4. Tool Development 26

4.1 The Arbitrary Waveform Generator (AWG) Tool 26

4.2 Compform 35

4.2.1 How the Compform Tool Works 36

4.2.2 The Use of a Default Node 38

4.2.3 Stand-Alone Operation 39

4.2.4 Use in ALF 39

4.3 The Connect Tool 40

4.3.1 Manual Operation 41

4.3.2 Automating Connections Through the Use of XML 41

4.3.3 Headers versus Metadata 44

5 Example Implementations 45

5.1 Methods for Testing 46
5.1.1 Defining Overhead 46
5.1.2 Measuring Latency (**cProfile**) 47
5.1.3 Measuring Memory Usage (**top**) 48
5.1.4 Variations in System Performance 48
5.2 Simulation for Measuring Install Latency 49
5.2.1 Simulation 1: Minimal Overhead Latency 50
5.2.2 Simulation 2: Maximum Overhead Latency 54
5.2.3 Overhead Latency in Intermediate Cases 58
5.2.4 Relating Latency to FFTs 60
5.3 Simulation for Measuring Memory Usage 63
5.3.1 Memory Usage of Simulation 1 and Simulation 2 64
5.3.2 Scalability of the Memory Usage 65

6. Conclusions and Future Work 66

A. Pass Data Component 70

Bibliography 72

List of Figures

2.1 The ALF Main Display 14

4.1 The AWG Tool's GUI 27

4.2 The Initialize Method of the **sources.py** Module 28

4.3 The **gen_sine** Method in the AWG Tool's **sources.py** File 30

4.4 The **read_file** Method from the AWG Tool's **sources.py** File 33

4.5 The Connect Tool GUI 40

5.1 The Data Flow of Simulation 1 50

5.2 Processing Times Resulting from Simulation 1 52

5.3 Breakdown of Execution Time for Simulation 1 54

5.4 The XML File Defining Simulation 2 55

5.5 Breakdown of Execution Time for Simulation 57

5.6 Percent of Time in Overhead for a Varying Number of Packets Sent per (SCA) Application Installation 59

5.7 FFT Processing Times Compared to **complexShort_idl.py** 62

5.8 FFT Processing Times Compared to Reconfiguration Overhead 62

List of Tables

5.1 **pstats** Results from Simulation 1 52

List of Abbreviations

ALF	An open source debugging environment for OSSIE
AWG	Arbitrary Waveform Generator
CORBA	Common Object Request Broker Architecture
DAS	Device Assignment Sequence
GPP	General Purpose Processor
GUI	Graphical User Interface
IDL	Interface Description Language
ORB	Object Request Broker
OSSIE	Open Source SCA Implementation::Embedded
OWD	OSSIE Waveform Developer
POA	Portable Object Adapter
SAD	Software Assembly Descriptor
SDR	Software Defined Radio
SCA	Software Communications Architecture
USRP	Universal Software Radio Peripheral
XML	Extensible Markup Language

List of Commonly Used Modules and Methods

complexShort_idl.py	The OSSIE module that allows Python to send complex short CORBA packets through CORBA.
connectPort	A method belonging to a *uses* port that a third party calls to indicate that the *uses* port should send data to the third party.
cProfile	A Python module used for profiling latency.
create	A method belonging to the Application Factory that deploys components to devices and returns a reference to an (SCA) application.
gen_sine	A method in **sources.py** that generates a digital sine wave.
_get_applicationFactories	A Domain Manager method that returns a list of existing Application Factories.
_get_name	An Application Factory method that returns the *name* attribute of the Application Factory.
getPort	A method belonging to a component that returns a reference to a requested port.

get_sources_list	A method in the module sources.py that returns a Python dictionary of available sources.
installApplication	A Domain Manager method that returns an Application Factory.
math	A Python module with several common mathematical methods.
_narrow	An **omniORBpy** method that changes the data type of a variable.
nodeBooter	A program in OSSIE that boots a node into the Domain.
numpy	A python package that provides several advanced mathematical functions, such as the FFT.
omniORB	An open source implementation of CORBA used by OSSIE.
omniORBpy	The Python interface for **omniORB**.
pStats	A Python module used for processing statistics generated by Python profilers, such as cProfile.
pushPacket	A method belonging to a *provides* port that allows a third party to send data to the *provides* port.
pushPacketMetadata	A method belonging to a *provides* port that allows a third party to send data and metadata to the *provides* port. Not defined in OSSIE 0.6.2.
read_file	A method in **sources.py** that reads data from a file.

releaseObject	An (SCA) application method that uninstalls the (SCA) application.
resolve_initial_references	An **omniORBpy** method that returns a *root context*.
runctx	A method in **cProfile** that runs the profiler on a given function.
sleep	A method from the Python **time** module that delays program execution by a given number of seconds.
sources.py	A module in the AWG package that generates data signals.
time	A Python module that provides several common time related methods.
top	A program that can be used for measuring memory usage.

Chapter 1

Introduction

1.1 Motivation

Rapid development through code reuse is one of software defined radio's (SDR) greatest advantages. By developing SDR software under a set of standards, new systems can be developed using the old code in less time than it would take to write all new code. The software communications architecture (SCA) was in part developed to facilitate code reuse for this purpose. The SCA allows a developer to define encapsulated Waveform Applications that can be later constructed into previously unforeseen applications. With the emergence of adaptive and cognitive radio technologies it becomes necessary to have the ability to construct reconfigurable radio systems out of the modular Waveform Applications in order to exploit the SCA's intention for code reuse.

The encapsulated Waveform Applications available may have internal reconfiguration options, but it is assumed that additional reconfiguration options may not be added (e.g.,

a BPSK system may not be altered to perform as an FSK system). It is assumed that these encapsulated sub-systems may be treated as a "black box" with a known input/output response. Since these encapsulated Waveform Applications cannot be internally reconfigured by the system developer, he/she must reconfigure his/her system by managing connections through (SCA) application management. The developer's components may still be reconfigured using standardized methods (metadata, calling configure methods, use of the event channel, etc.).

According to [1], the first advantage of software defined radio is "ease of design". The use of encapsulated Waveform Applications to develop applications promotes code reuse and can therefore reduce development time by reducing the amount of new code required. As a result of code reuse and reduced development time, ease of design can be realized. The use of existing components and Waveforms Applications to create applications also greatly simplifies the development process, as the need to develop SCA XML profiles and new software can be reduced or even eliminated. This thesis primarily focuses on the use of a single XML file that defines the interconnection between the existing components and Waveform Applications.

It is also stated in [1] that an advantage of SDR is "multimode operation". More traditional radios are designed to function under a single mode of operation. As an example, a traditional car radio may be designed to function only as an AM receiver. The car radio may also have FM receiving capabilities, but additional hardware must be present. For this matter the car radio is actually two radios, commonly utilizing a shared

interface. The radio cannot function as a transmitter and the AM radio device cannot demodulate an FM signal. Switching between AM and FM operation involves a human turning off the AM radio and turning on the FM radio. A software radio varies dramatically in that (ideally) all of the hardware for multiple radio functions is shared, and true multimode operation can be achieved through software. Not only can the radio potentially operate in more modes, it can also be designed to automatically switch between modes based on a given set of policies (adaptive and cognitive radio). In this case a developer has to ability to design radios that are capable to selecting more efficient operating modes based on their environment. This multimode operation can come at the cost of additional processing overhead, which is explored in this thesis. By being able to construct systems from existing components and Waveform Applications, in software, this multimode operation can be achieved with minimal understanding of the SCA.

1.2 Contributions

In order to help facilitate the SCA's intention for code reuse and SDR's intention for ease of design, several rapid prototyping tools are developed. With these tools a developer is able to quickly construct systems out of modularized components and Waveform Applications as well as validate and debug the systems in a timely manner. The following new tools are developed for these purposes:

- The Arbitrary Waveform Generator (AWG) for the ALF graphical debugging environment (named after the 1980's television sitcom "ALF") is created. This

tool allows a developer to connect to a *provides* port in a running system and send data to the port for debugging and profiling purposes. With the ability to send multiple types of waveforms, packet rates, and headers, this tool can be utilized in debugging adaptive radio systems. Also, at the flip of a switch, the tool gives the user the ability to utilize the standard Python profiling module on Python tools without the need to add additional code to the tool being profiled.

- The "Compform" tool allows the developer to effortlessly install an existing component as an (SCA) application [1] in OSSIE. With this ability, the developer can use any tools designed for debugging and validating (SCA) applications to debug and validate components, speeding development time.

- The Connect Tool for the ALF graphical debugging environment is created. This tool allows a developer to connect ports in his/her Domain during runtime. This tool also allows the developer to specify a set of rules (defined in XML) for installing and connecting (SCA) applications to develop reconfigurable systems.

[1] The use of "(SCA)": The SCA [2] often redefines commonly used terms, which can often lead to confusion. This paper adopts the convention used in the SCA specification to add clarity: words used in the context defined by the SCA are preceded by the adjective "(SCA)". For example, an "application" used in the traditional communications context simply reads "application", while an "application" used in the SCA context reads "(SCA) application". SCA terms that are defined using capital letters are considered explicit and do not require this naming convention.

1.3 Thesis Organization

This thesis begins by discussing background concepts that are important for understanding the reconfiguration methods considered and implemented, such as the SCA and the Open Source SCA Implementation:: Embedded (OSSIE). In addition, existing OSSIE tools that are used for implementing these concepts are introduced. The thesis then explains the reconfiguration methods along with the logistics of implementing these methods, such as the use of the CORBA middleware and the OSSIE Core Framework. With these concepts explained, the implementations of the reconfiguration methods developed, in the form of OSSIE tools, are discussed in detail. These tools are then utilized to measure the relative latencies and memory requirements associated with the reconfiguration methods.

Chapter 2

Background

This chapter discusses several fundamentals necessary for understanding the concepts discussed in this thesis such as the SCA and OSSIE. Along with the SCA itself, several concepts introduced by the SCA are described. The tools developed in this thesis are based on existing OSSIE tools that are detailed. Finally, related work is discussed.

2.1 The SCA

One of the indisputable advantages of SDR is re-configurability. When developing an SDR, its ability to be updated and modified in the future must be considered. These updates may be made by different groups of people over long periods of time, which highlights the need for a set of standards to govern the format in which code for SDRs is written. Various standards have been developed, and one of the most predominant standards to date is the SCA, originally developed by the United States Department of Defense [2,3]. The SCA provides an extensive set of definitions that govern the development of both an operating system as well as the SDRs that will run in that

operating system. Of particular interest for this thesis are the SCA definitions dictating the installation of (SCA) applications (discussed in Section 3.3) as well as the interconnection between ports (discussed in Section 3.4).

2.1.1 Waveforms, Waveform Applications, and (SCA) Applications

In order to appropriately classify the systems being developed through the concepts discussed in this thesis, it is important to define and understand (SCA) Waveforms, Waveform Applications, and (SCA) applications. The SCA specification defines a (SCA) Waveform as "the set of transformations applied to information that is transmitted over the air and the corresponding set of transformations to convert received signals back to their information content [3]". The SCA specification defines a Waveform Application as "the collection of software elements (modules or components) which perform any or all of the transformations defined for a specific waveform [3]". For this thesis Waveform Applications will be constructed from components and devices. This thesis will be exploring systems constructed of multiple pre-existing Waveform Applications as well as pre-existing components that can be effortlessly constructed into Waveform Applications with the Compform Tool. It is important to note that the systems developed in this thesis are not necessarily (SCA) Waveforms as they may only consist of the transformations applied to transmit the information or the transformations required to receive the information, where an (SCA) Waveform requires both.

It is also important to define and understand the (SCA) application. An (SCA) application is "an executable software program which may contain one or more modules [3]", where said program is constructed by an Application Factory based on a Software Assembly Descriptor (SAD) file. Each Waveform Application used in this thesis is defined by a SAD file, and can therefore be implemented as an (SCA) application. However, as of SCA version 2.2.2, there exists no SCA descriptor file that defines a system constructed of Waveform Applications defined by SAD files that can be used to generate an (SCA) application [3,4]. The systems being developed in this thesis, consisting of multiple (SCA) applications, are therefore applications not (SCA) applications.

The Domain, according to the SCA, is simply a "set of hardware devices and available applications under the control of a single Domain Manager component [3]". Where the Domain Manager component "manages the complete set of available hardware devices and [(SCA)] applications [3]". For the scope of this thesis, all (SCA) applications will reside within a single Domain.

2.1.2 Components

The SCA defines a component as "a software module or element that conforms to and implements a set of interfaces [3]". The SCA also further defines a component in the definition of a *Resource* stating that a SCA-conformant component must "implement the

Resource interface [3]". For the purposes of this thesis, a component is realized as a signal-processing block of any kind that contains *uses* and/or *provides* ports. The implementation of the *Resource* interface is accomplished through inheritance from the OSSIE Core Framework. The components used in the simulations for this thesis are designed to isolate overhead generated by conforming to the SCA specification as much as possible, therefore making the finer details of SCA-conformance of the component arbitrary.

2.1.3 Uses and Provides Ports

The SCA defines two types of ports; a *uses* port is a "source/consumer", while a *provides* port is a "sink/producer [3]". To a developer who is new to the SCA, these definitions are often confusing. In the case of as *uses* port, the "source" definition is referring to the port's ability to be a source of data for other ports, while the "consumer" definition is referring to the port's use of data produced by the component it belongs to. In the case of a provides port, the "sink" definition is referring to the port's ability to accept data, while the "producer" definition is referring the port's ability to provide data to the component it belongs to. In the scope of most OSSIE applications, *uses* ports are used to output data from a component, and *provides* ports are used to input data to a component. For the scope of this thesis, when a connection is established, a *provides* port is always connecting to a *uses* port.

2.1.4 CORBA

The Common Object Request Broker Architecture (CORBA) standard is widely used in the SCA [3]. CORBA is a middleware that allows software objects to communicate to each other through a standardized interface description language (IDL). CORBA is designed to be both language and platform independent. These features allow the developer to create various software applications in various languages on various platforms based on which language and platform is appropriate for each application. The tools detailed in this thesis (written in Python) use CORBA to interact with both the OSSIE Core Framework (written in C++) and existing OSSIE components (often, but not necessarily, written in C++). Another important feature of the CORBA standard is the separation of *interface from implementation* [5]. Even though OSSIE chooses to utilize a specific CORBA implementation (**OmniORB**, detailed in Section 3.2), it remains compatible with any other software package utilizing any other CORBA implementation [5].

2.2 OSSIE

An open-source implementation of the SCA has been developed by Dr. Max Robert and a team of students from Dr. Jeff Reed's Software Radio class and research group at Virginia Polytechnic Institute and State University (Virginia Tech). This effort, OSSIE, provides an environment, available for free to the public, for development and

implementation of (SCA) applications [6]. Since its original conception, OSSIE has been supported and further developed by various Virginia Tech students.

As an open-source effort, OSSIE is still in its development phase. Fortunately, many efforts are being made by the OSSIE group at Virginia Tech to make OSSIE converge towards SCA compliance and to make OSSIE easier to use. One of the most notable achievements contributing towards ease of use is the development of the OSSIE Rapid Prototyping Tools, specifically the OSSIE Waveform Developer (OWD) [2].

2.2.1 The OSSIE Waveform Developer

One tool available to OSSIE developers is the OSSIE Waveform Developer (OWD) [2,7]. OWD has the ability to generate code shells for SCA components as well as generate the SAD files that define a Waveform Application. ALF, discussed in Section 2.2.2, is able to utilize OWD in order to generate a single available component into an (SCA) application. With access to this functionality, ALF is able to directly run components as (SCA) applications in OSSIE. ALF refers to these components being run as (SCA) applications as *Compforms* (named for the Waveform Application defined by the generated XML files: COMPonent-waveFORMS). It is important to note that the term *Compform* is not defined in the SCA. The Compform Tool is discussed in greater detail in Section 4.2.

2.2.2 ALF

ALF is an open-source graphical debugging environment for (SCA) applications running in OSSIE [6]. This software was originally donated from SAIC to the OSSIE project in January 2007. Because ALF is open-source and it utilizes an open-source framework (OSSIE), there is unlimited potential for tool add-ons as well as other modifications. This expandability allows the developer to utilize the tool for unique and proprietary systems. Simply put, if ALF or the ALF tools do not fit the needs of the developer, they can be changed by the developer through modification of the source code.

Because CORBA allows communication across a variety of platforms, ALF is able to use CORBA to communicate to any port in the OSSIE file system on any platform. This capability gives ALF the freedom to utilize its debugging tools on platforms that cannot normally run ALF natively, such as embedded platforms. The simulations in this thesis will focus on the use of CORBA on a general purpose processor (GPP) but potentially could be reproduced on other platforms.

Once a developer has created his/her component code and XML, as well as the appropriate XML files defining the interconnection between components, he/she can use the OSSIE Core Framework along with ALF to run and debug an (SCA) application. Debugging can be performed using various tools to monitor data, to send data to the system, as well as to monitor latency in the (SCA) application. The existing ALF tools are written in, but are not limited to, Python. In ALF's current version, available tools

include Plot, Arbitrary Waveform Generator (AWG), Write to File, Compform and Connect [6].

Through the use of CORBA, ALF is able to connect to any available *uses* or *provides* port running in OSSIE. ALF connects to the OSSIE naming service in order to get a list of all installed (SCA) applications, through which it can obtain a list of all components in each (SCA) application. When a user wishes to establish a connection between an ALF tool and a port, a pointer to the resource is obtained, which allows ALF to get a reference to the port using the **getPort** method through CORBA. In the case of a *provides* port, the handle to the port allows ALF to call port methods such as **pushPacket**. In the case of a *uses* port, the **connectPort** method is called. Once this method is called, all data sent to the component's *uses* port will in turn be sent to the appropriate ALF tool. The connection process is discussed in greater detail in Section 3.4.

Prior to the development of ALF, in order to debug a single component, a Waveform Application containing the component must be generated in order to run the component in OSSIE as an (SCA) application. ALF utilizes OWD to automatically create a temporary (SCA) application for any existing component, allowing the ALF tools to be utilized on stand-alone components (not just Waveform Applications).

Figure 2.1 displays the ALF main window. The upper-left portion of the display contains a list of available Waveform Applications on the operating system's file system. For OSSIE 0.6.2, this includes all Waveform Applications in the "/sdr/waveforms" directory.

To install and start a Waveform Application as an (SCA) application, the user simply double-clicks on the name of the desired Waveform Application. Once installed, the running (SCA) application is added to the list of installed (SCA) applications in the lower-left portion of the display, labeled "Manage Waveforms". A block diagram display of the (SCA) application of interest inhabits the right portion of the display. The middle-left of the display (labeled "Launch Components as Waveforms") provides a list of the components existing in the "/sdr/xml" directory with corresponding executable files in the "/sdr/bin" directory. The process of installing and starting (SCA) applications from these components is described in Section 4.2.4 [8].

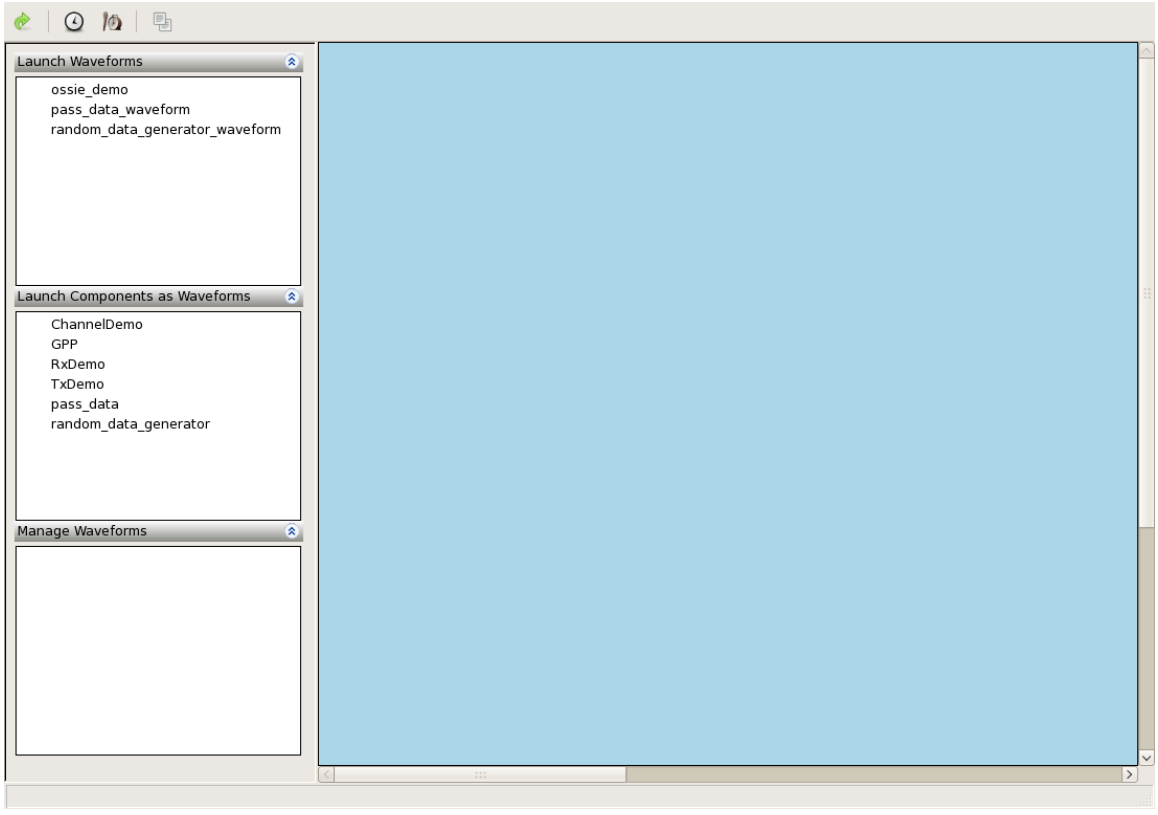

Figure 2.1: The ALF Main Display.

2.3 Related Work

Development of SCA systems using a collection of Waveform Applications can be approached from two perspectives. The first perspective uses the existing SCA specification (e.g., the approach taken in this thesis), and the second perspective involves modification of the SCA specification by adding a definition of the "aggregate application".

2.3.1 The Aggregate Application Concept

The notion of connecting (SCA) applications to create radio systems is explored by [4] through the concept of an aggregate application. In [4] an addition to the SCA specification, the aggregate application, which defines a means for interconnecting existing (SCA) applications, is proposed. This proposal accommodates the creation of systems similar to the fixed installation method discussed in this thesis, where the (SCA) applications and the connections between them are established during startup. The concepts of installing/uninstalling (SCA) applications during runtime or connecting/disconnecting (SCA) applications during runtime are not discussed in [4]. In [4] it is proposed that the highest level of modularity that the SCA truly provides is the component, and in order to promote the reusability of a component it needs to be designed to be as granular as possible. As an example, it is stated that a component that is designed to execute forward error correction coding (FEC) is more reusable than a full-

blown modem because its functionality is more generalized. Therefore, without a standard for interconnecting collections of components (in this case (SCA) applications), the SCA only provides modularity and reusability at the lower, component level but not at the higher Waveform Application level.

Two approaches to implementing the aggregate application are discussed in [4], one that requires modification to the Core Framework and one that does not. As with the implementations discussed in this thesis, the proposed implementation in [4] that does not require modifications to the Core Framework requires the use of an additional tool. The implementation of the aggregate application through an additional tool proposed in [4] does vary from the concepts discussed in this thesis in one major way: the tool combines the (SCA) applications prior to runtime and installs the resulting aggregate application as a single (SCA) application. As previously mentioned, this approach lacks the ability to interconnect (SCA) applications during runtime or to install (SCA) applications on demand during runtime. It is also pointed out in [4] that by deploying the collection of sub-applications as a single (SCA) application, the developer loses the ability to determine which application might be the cause of a deployment error, in the event that a deployment error exists. It is observed in [4] that there exists no standard for implementing aggregate applications through modeling tools. It is also stated in [4] that even though connection between (SCA) applications is possible through proprietary formats (e.g., the format discussed in this thesis), the use of proprietary solutions is "contrary to the SCA philosophy [4]".

It is implied in that the use of runtime monitoring tools is hindered, but in the case of the OSSIE monitoring tools, this is not true. Since the OSSIE monitoring tools (timing, Plot, etc.) are able to connect to *any* component port in *any* (SCA) application in the Domain, a component's membership to a particular (SCA) application is arbitrary. In the event that the monitoring tools can only monitor on the (SCA) application level, the claim in [4] is correct.

In alternative to implementing the aggregate application concept through modeling tools, it is proposed in [4] that modifying the SAD file to incorporate additional XML nodes required to define the aggregate application. With these modifications come necessary modifications to the SCA Core Framework, specifically the (SCA) *Application* interface and the Domain Manager, both of which are discussed in [4].

Chapter 3

(SCA) Application Management

(SCA) applications can be managed in a Domain in order to reconfigure a radio system. This management is accomplished largely through the use of **omniORB**, the installation/uninstallation of (SCA) applications, as well as the connection/disconnection of ports. This chapter discusses these concepts, which are later implemented in the tools discussed in Chapter 4. It is known that a reconfigurable radio can gain efficiency through proper selection of operating modes, but the efficiency of the reconfiguration process itself must also be considered in order to justify the use of the reconfigurable radio. The extra processing requirements as well as the extra memory requirements must therefore be considered when designing a reconfigurable radio.

3.1 Methods for Managing (SCA) Applications

With any SCA communications system there exists an (SCA) producer. In the case of a receiver, this (SCA) producer can be in the form of an RF, IF, or baseband signal. In the

case of an adaptive radio, decisions are made based on the (SCA) producer's output signal, and the radio is reconfigured appropriately. When analyzing methods for reconfiguration, the adaptive algorithm can be viewed as independent and therefore arbitrary. In the case where the adaptive algorithm is created independent of the reconfiguration method, the adaptive algorithm must send the (SCA) producer signal to something that is capable of reconfiguring the system along with information indicating what should be done with the (SCA) producer signal. For this thesis, a tool called the "Connect Tool" will be used to perform this reconfiguration. The Connect Tool is responsible for forwarding the (SCA) producer signal to the appropriate (SCA) consumer. By adding the Connect Tool as an intermediary, packets can be buffered during the reconfiguration process, eliminating potential packet loss.

Two methods for system reconfiguration via (SCA) application management are considered. In the first method, the installation of (SCA) applications is variable during runtime. In this method, whenever the configuration of the radio changes, any existing, no longer needed (SCA) application can be uninstalled and disconnected, and a new (SCA) application is installed in its place. Using this method, only (SCA) applications that are actively processing data are installed in OSSIE, minimizing memory requirements. This approach exhibits minimal memory usage when only one (SCA) application is installed at a time. The worst-case latency performance occurs when the required (SCA) application must be installed and uninstalled every time a packet is sent. This approach can be extended to ensure that only what is determined to be the most immediately useful subset of available (SCA) applications is installed at a given time.

In the second method considered, the installation of (SCA) applications is fixed, and therefore all potentially required (SCA) applications must be installed during startup. The (SCA) applications installed and connections made (as defined by an XML file described in Section 4.3.2) do not change during the lifecycle of the application. The Connect Tool then reconfigures the system by sending the data to the appropriate (SCA) application during runtime. The Connect Tool keeps a list of available connections. This strategy of installing all possible (SCA) applications during startup dramatically reduces runtime delays that are experienced when the (SCA) applications are installed and connected. It therefore exhibits best-case latency performance. In the case where a large number of potential (SCA) applications must be available, this approach can become unwieldy, especially if only a small number of the (SCA) applications are used at a time, and memory use is not optimized.

3.2 The Use of omniORB

Developing a complete understanding of CORBA as well as implementing the entire standard can be a time-intensive and daunting task. However, having an in depth understanding of CORBA is not required for its use in the SCA. Open-source implementations of CORBA are freely available that provide the functionality necessary to build systems in OSSIE, eliminating the need for a developer to spend time developing an implementation of the standard. Furthermore, many of these implementations only require an elementary understanding of the CORBA architecture.

As previously mentioned, OSSIE chooses to utilize **omniORB** [9] for CORBA implementations [6]. The use of **omniORB** provides both a CORBA implementation for C++ as well as Python (with the additional **omniORBpy** [9] package, installed with OSSIE 0.6.2 [6]). In addition, **omniORB** claims to be "largely CORBA 2.6 compliant [9]". Through the use of **omniORB**, a developer can easily make connections between ORBs. The process of obtaining an ORB is discussed in Section 3.2.1. With this ORB the object, acting as a client, is able to connect to any other ORB, acting as a servant, in the OSSIE naming service, allowing access to the server's methods defined in IDL. When the server is inheriting from the Port class (e.g., in the case that the server is a *uses* port), this access to methods is particularly useful for connecting and disconnecting ports (the connection process is discussed in greater detail in Section 3.4). It is also especially important to note that this use of CORBA for establishing data connections between ports does not require that the ports being connected reside within the same (SCA) application, allowing for the interconnection of (SCA) applications via component ports. Required marshalling involved in transferring data between components is also handled [9].

3.2.1 Obtaining an ORB and a Root Context

The Python package CORBA is used to obtain an ORB for the component as follows:

```
self.orb = CORBA.ORB_init()
```

With a reference to the servant ORB, a reference to the servant's Portable Object Adapter (POA) can be obtained and activated. The POA is a feature of the ORB that manages the server-side resources by deactivating servants of an object when they are not in use [5]. If activate is not called on the POA, the server-side resources will not be available.

Contact with the naming service, which will allow access to ORBs running in OSSIE, can be obtained with a reference to the *root context*. The method **resolve_initial_references** is called on the ORB. The resulting variable is a reference to the *root context*. In order for Python to correctly utilize the root context, the method **_narrow** is used on the object returned by **resolve_initial_references**, which will return the correct Python type[2] [9]. With the *root context* available in the proper type, a reference to the Domain Manager is obtained through a simple function call.

3.3 Installing Waveform Applications as (SCA) Applications

The process of installing an (SCA) application in OSSIE is primarily a matter of making function calls to the OSSIE Core Framework. In the case of doing this process in Python (as done in ALF and the ALF tools), this is accomplished through the use of the Python bindings created during the OSSIE installation. The process also involves a small amount of XML parsing and error checking that are not detailed in this writing.

[2] Python usually acts as a fully dynamically typed language, which would imply that this step is not necessary. The "gory details" of why this process is necessary is detailed in [9].

Additional processing times added by Python during the installation process are generalized as "Python overhead" in the simulations in this thesis, as it is assumed that it is likely possible to speed up the process through the use of a compiled language.

The device assignment sequence (DAS) file is parsed and processed in order to construct a device sequence. The device sequence (of type DeviceAssignmentType defined in the OSSIE Core Framework) is essentially a list of components paired with corresponding devices. This device sequence is later passed to the Application Factory's **create** method (discussed below), which will in turn deploy the appropriate components to the appropriate devices [3].

In order to create an Application Factory for a given Waveform Application, the location of the SAD file defining the Waveform Application is sent to the Domain Manager's **installApplication** method. The **installApplication** method will create an Application Factory for this application [3].

The Domain Manager method **_get_applicationFactories** is used to get a list of all the available application factories in the Domain. Since each of these Application Factories has a **_get_name** attribute that returns the name of the Application Factory, it is trivial to acquire the necessary Application Factory from the list. With the reference to the desired Application Factory, the **create** method can be called to create the final (SCA) application. When the (SCA) application is created, a reference to the (SCA) application

is returned, and start can then be called on the (SCA) application. With start called on the (SCA) application, the installation process is considered complete.

3.3.1 Uninstalling Waveform Applications

It is assumed that the (SCA) application reference obtained during installation is retained, and that the user is not attempting to uninstall (SCA) applications that he/she did not install. If this (SCA) application reference is not present, then some effort must be expended to obtain it. With this (SCA) application reference, the user simply needs to call **releaseObject** on the reference to uninstall the (SCA) application. The **releaseObject** method "terminates execution of the application, returns all allocated resources, and de-allocates the resources' capacities in use by the devices associated with the application [3]".

3.4 Connecting Ports

Once the required (SCA) applications are installed, it is necessary to connect them so that data can be passed between necessary components. The process of connecting a *uses* and *provides* port is accomplished through the use of the OSSIE naming service and CORBA.

3.4.1 Retrieving A Port Handle from the Naming Service

With a reference to the *root context* available as well as a known (SCA) application name, component instance name, and port name, a useable reference to the desired component can be obtained through the **_resolve** method. With this component reference, the **getPort** method from the desired component can be called with the known port name in order to get the reference to the port. With the reference ("port handle") available the user now has access to all of the port's methods that are defined in IDL, such as pushPacket (in the event that the port is of type *provides*).

3.4.2 Finalizing the Connection

For a *uses* port to be connected to a *provides* port, the *uses* port must have a reference to the *provides* port. Since the entity connecting to the *uses* port (often a component) has a reference to the *uses* port (obtained form the **getPort** method described above), the connection is created by the *provides* port calling **connectPort** on the *uses* port with a reference to the *provides* port as an argument. In the **connectPort** method, the *uses* port records the reference to the *provides* port, which is used later whenever data is sent via the **pushPacket** method.

Chapter 4

Tool Development

Three OSSIE tools are developed both for the promotion of rapid development as well as the development and analysis of reconfigurable systems. Among these tools are the Arbitrary Waveform Generator Tool, which is capable of sending signals to any *provides* port in the Domain, the Compform Tool, which allows for rapid deployment of a component as an (SCA) application, and the Connect Tool, which is capable of implementing application reconfiguration.

4.1 The Arbitrary Waveform Generator (AWG) Tool

Within the available ALF tool library there exists an AWG Tool. This tool allows the developer to connect to an existing *provides* (input) port in a running (SCA) application. Once the connection is established, the tool is able to send any arbitrary signal (as defined by the developer) to the port. As long as the port and the component it belongs to are capable of processing enough data in real time (either through the use of efficient

algorithms or buffering), the AWG can send data to the component simultaneously with data being sent through a previously existing connection. The AWG GUI is shown in Figure 4.1.

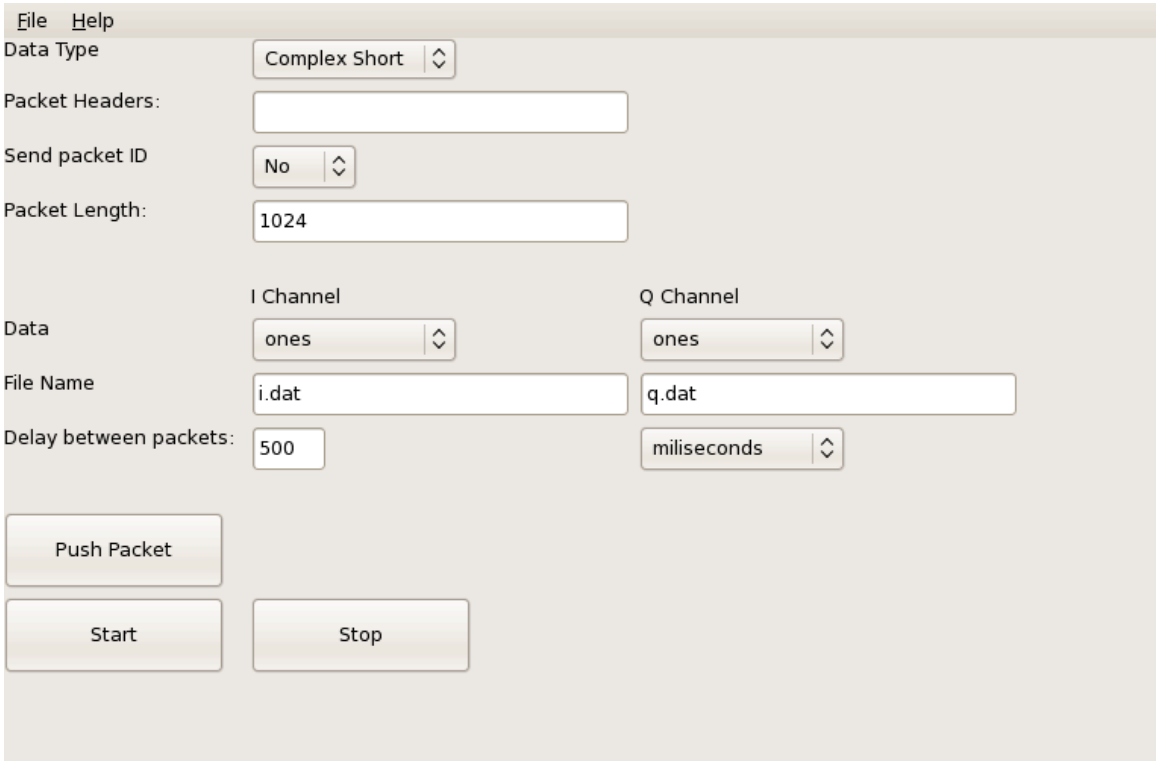

Figure 4.1: The AWG Tool's GUI.

Multiple options exist for the type of signal the developer wishes to generate (or read from file) and "push", where "push" refers to sending the data to the *provides* port via the **pushPacket** method. In the module **sources.py** there exists a signal class "sources". This class contains the method **get_sources_list** as well as a method for each signal type the user wishes to define. The default tool currently has 6 available signal types [6]. The available signal types are defined in the sources' "__init__" attribute, seen in Figure 4.2.

```
def __init__( self, parent ):
        self.parent = parent
        self.available_sources={
                                        'file': 'read_file()',
                                        'sine': 'gen_sine()',
                                        'cosine': 'gen_cosine()',
                                        'random': 'gen_random_data()',
                                        'zeros': 'gen_zeros()',
                                        'ones': 'gen_ones()' }
```

Figure 4.2: The Initialize Method of the **sources.py** Module. This method declares the available sources (left of each ":") and their corresponding methods (right of each ":").

The **get_sources_list** method (seen below) simply returns a list[3] of the available sources:

```
def get_sources_list( self ):
        return self.available_sources
```

This list of sources serves two purposes in the AWG.py module. First, the list is used in the initialization of the AWG GUI so that each source in **sources.py** can be selected from in the GUI menu. When the AWG Tool is running, the developer is able to select the

[3] The list of sources is not a *Python* list. It is in the form of a Python "dictionary [10]".

desired source, and the index of the selected source then dictates which signal method is called.

Second, by having a Pythonic list of signals available, a developer is able to easily add his/her own signal generation code without having to edit any of the code in AWG.py. Therefore, knowledge of GUI programming or even the SCA is not necessary for adding signal sources to the tool's GUI. However, due to the open-source nature of AWG.py a more experienced GUI programmer is still able to add his/her own graphical switches/buttons/inputs/etcetera to the GUI as he/she feels necessary. The SCA nature of the tool, as well as the connections made, also remains transparent to the developer, but is still editable if necessary for more in-depth debugging.

The **gen_sine** method is analyzed as an example of the source methods available. The code for this method can be seen in Figure 4.3.

```
def gen_sine(self):


    #initializations
    count = 0
    sine = []


    #recursively generate the sinusoid
    while count < self.parent.len:
        sine.append(math.sin(self.parent.freq * 2 * math.pi * count / self.parent.len))
        count = count + 1
    return sine
```

Figure 4.3. The **gen_sine** Method in the AWG Tool's sources.py File.

A variable "count" is initialized for the while loop. The variable "sine" is declared as a Python list so that the "append" method can be utilized. A Python list has been selected as the standard return type for the source methods. The list returned (in this case "sine") will be sent directly to the component *provides* port. The type of data within the list (e.g., short, float, or char) can either be set in the signal source method or type-casted in the AWG.py file (in this case, the latter option is used). The data type within the list must be consistent with the data type of the *provides* port being connected (otherwise CORBA will, appropriately, throw an error).

In this example, the signal is recursively generated using the Python **math** module. This approach for generating the list can be replaced with more efficient or appropriate approaches as desired by the developer. The variables “parent.freq” (sinusoidal frequency) and “parent.len” (desired length of the list) are data fields from the GUI class. This object-oriented approach to accessing public data fields allows the developer to easily access the variables that are set by the GUI’s switches/buttons/input/etc.. For example, the developer can add an option for phase noise variance in the GUI, which can then be set as a public class data field that can be accessed by the "sources" class. In the case that the developer wishes to generate signals that are not easily or efficiently constructed by calling methods from the Python math module, custom modules can be written in other programming languages (commonly C).

In addition to the ability to add signal generation methods to the sources class, the **read_file** method exists for developers who wish to generate their signals using other methods, such as third party software, for whatever reason. For example, a developer may desire to generate a signal in MATLAB or use data that has been taken from “over the air.” Currently, the developer is able to specify two file names in the GUI, one for the in-phase channel and one for the quadrature-phase channel. The data delimiter can also be specified (in the current version the delimiter is hard coded to be a comma). The **read_file** method (shown in Figure 4.4) attempts to open the file and, if successful, it reads data in the file as a string. The data can then be reformatted as desired (in this case the square-brackets generated by MATLAB are removed). The tool's ability to reformat data facilitates interoperability with other software packages that may parse data in

different formats. Once the data is reformatted, the Python method "split" (available in the Python string library [10]) breaks the data string into a Python list that can then be returned to the parent (AWG.py).

```
def read_file(self, parent):     # function definition
    try:
        my_file = open(parent.file_name, 'r')        # attempt to open the file:
    except IOError:    # In case of an error
        print "error opening, or no file named " + parent.file_name
        return []          #return an empty packet

    data_from_file = my_file.read()       # read the file as a single string
    my_file.close()

    # reformat the string by removing any unwanted characters
    data_from_file.strip('\n')
    data_from_file.strip('[')
    data_from_file.strip(']')

    data_from_file =
        data_from_file.split(parent.delimiter)       #break string into a list
    return data_from_file
```

Figure 4.4: The **read_file** Method from the AWG Tool's **sources.py** File.

The AWG frame contains a button labeled "Push Packet". Once the desired options are set in the AWG GUI and the Push Packet button is pressed, the in-phase and qudrature-

phase signals are generated by calling the appropriate method in the sources class. The method **pushPacket** is called, with the generated signals as arguments, on the port handle created when the connection was established:

```
self.PortHandle.pushPacket( self.I , self.Q )
```

The data is now in the hands of the component to which the tool is connected. The **pushPacket** method can be called as many times as the user desires [8].

The AWG also has the ability to send a continuous stream of packets, eliminating the need for the user to press the Push Packet button every time a packet is to be sent. When the Start button is pushed, packets will be recursively sent, with a delay between each packet that is also specified in the GUI (discussed below). Also, the user can specify a number of packets to be sent. The packets will continue to be sent until the Stop button is pushed, the specified number of packets is sent, or the tool is closed.

For the purposes of validating the Connect Tool (see Section 4.3), the user is given the ability to dictate a sequence of packet headers (comma delimited). For example:

```
1, 2, 3
```

Will send the first packet with a header '1', the second packet with a header '2', and a third packet with a header '3'. The fourth packet will start again at the beginning of the sequence.

Another essential configuration option is the ability to insert a delay between packets. Since communication using CORBA is entirely packet based, this delay, the time spent generating the data, and the packet length will determine the data rate of the system. The method **sleep** from the standard Python module **time** [10] provides the ability to insert delay with minimal processing overhead.

The AWG Tool contains a switch to turn on Python profiling. When this switch is turned on, the Python profiler will write profile files to the /sdr/tools/alf/profiles directory. These files can be post processed by the Python module **pstats** [10]. The Python profiler and the **pstats** module are discussed in greater detail in Sections 5.1 and 5.2.

4.2 Compform

The OSSIE core framework gives a developer the ability to run Waveform Applications (or complete (SCA) Waveforms) defined by a software assembly descriptor XML file ("SAD" file) and a device assignment sequence XML file ("DAS file"). Because the use of these two files is the only way to define (SCA) applications that can be run in OSSIE, the component must first be part of an (SCA) application defined by these two XML files. The "Compform" Tool (COMPonent-waveFORM) allows an ALF user to run a

component in OSSIE by automatically generating the necessary XML files that define a Waveform Application constructed from a single instance of that component[4]. With this ability available, the developer can easily construct systems in OSSIE not only out of existing encapsulated Waveform Applications, but also out of existing encapsulated components.

4.2.1 How the Compform Tool Works

The Compform Tool is designed to draw from existing OWD modules so that the XML file generation remains consistent with OWD whenever changes to OSSIE are made. The modularized nature of OWD facilitates the reuse of the methods necessary for generating these XML files. Specifically, the following modules are utilized:

getResource: This method, given the name of a component and the directory where the component's XML descriptor files can be found, returns a ComponentClass object. The ComponentClass, as defined by OWD, defines all of the attributes of a given component needed for generating the waveform XML files (i.e., port names, properties, etc.). All of these necessary attributes are obtained through parsing the component's XML descriptor files.

[4] Since it is rare that a component actually performs the entire set of operations to both send and receive data, the term "Compform" is actually misleading. The term originates from a previous misunderstanding of the SCA term "Waveform". Since the goal of this tool is to run a component as an (SCA) application, it could be more appropriately titled "Complication" (COMPonent-appLICATION).

genxml: This method, given a component list (described below), a path to save the generated XML file, and a Waveform Application name, generates the SAD file for the resulting Waveform Application. This file details the components present in the Waveform Application, the connections between the components, and the property overrides for the component instantiations. The connections between components and the property overrides are not used by the Compform Tool.

genDAS: This method, given a component list (described below), a path to save the generated XML file, and a Waveform Application name, generates the DAS file for the Waveform Application. This file dictates what hardware the component(s) in the waveform will run on.

A list of Component Class instances is created that completely describes the Waveform Application. The component class details all of the necessary attributes of the component as it will be deployed in the Waveform Application. Thee attributes include property overrides, connections to other components, and device deployment. In order for the component to be deployed to a device, an object with the default GPP UUID is created. The motivations for using a default GPP UUID are detailed below. In order for the

Waveform Application to run in OSSIE, an assembly controller must be present. In the case of using the Compform Tool, the data field "AssemblyController" is set to *True* for the single Component Class instance existing in the component list.

4.2.2 The Use of a Default Node

Since OSSIE version 0.6.2 does not support the automatic deployment of components to appropriate devices [6,7], the user must explicitly define the deployment of components to devices through a DAS XML file. Two devices are released with OSSIE 0.6.2, GPP and USRP. Two additional devices, supporting sound input and output, are also available for OSSIE 0.6.2 but were released after OSSIE 0.6.2's original inception [6]. Under OSSIE 0.6.2, the GPP is the only device to which resources can be deployed. Because of this, ALF assumes that all waveforms produced by the Compform Tool are deployed to the GPP. OSSIE 0.6.2 also has a set of default nodes that come installed with OSSIE, which have various combinations of the available devices. Since the waveforms developed by the Compform only have a single component deployed to the GPP, the default GPP node, consisting of only one GPP with a standardized universally unique identifier (UUID), is used. By using a default GPP node that is always installed with the standard OSSIE 0.6.2 installation, the Compform Tool does not require the development of a new node.

4.2.3 Stand-Alone Operation

A user has the option to run the Compform Tool from the command line independent of ALF (by typing "python compform.py" into a terminal with the arguments described below). This option is useful if a developer wishes to only develop the XML files without automatically running the Waveform Application as an (SCA) application in the ALF debugging environment (e.g., if he/she wishes to deploy the component to a node other than the default_GPP_node, the XML can be generated and then modified prior to runtime). With this option, the user sends the script a component name, a destination directory (for the generated XML and Python install files), and a Waveform Application name (optional). If the Waveform Application name is omitted, the Waveform Application is automatically named to be the component name followed by "_waveform". The stand-alone method also generates a setup.py file that, when run, will install the generated XML files to the /sdr/waveforms directory. Note that the installation destination directory can vary depending on the version of OSSIE being used.

4.2.4 Use in ALF

The use of the Compform Tool in ALF is relatively trivial. ALF can utilize Compform in two ways: through the main ALF window, and through the Connect Tool. On the left side of the ALF main window (see Figure 2.1) there is a panel labeled "Launch Components as Waveforms". In this panel is a list of components that have been

installed to the /sdr directory. In order to create and run a Compform as an (SCA) application, the user simply needs to double-click on the desired component. When the Compform Tool is finished running, the resulting (SCA) application can be viewed in the "Manage Waveforms" panel of the ALF main window. The use of Compform in the Connect Tool is detailed in Section 4.3.

4.3 The Connect Tool

As the name of the tool implies, the Connect Tool allows an ALF user to connect existing ports in his/her Domain. To start the connect tool, the ALF debugging environment must first be running. At the top of the ALF main display there exists four buttons. The button on the far right (fourth from the left, to the right of the two timing buttons) starts the connect tool. The GUI for the Connect Tool can is shown in Figure 4.5.

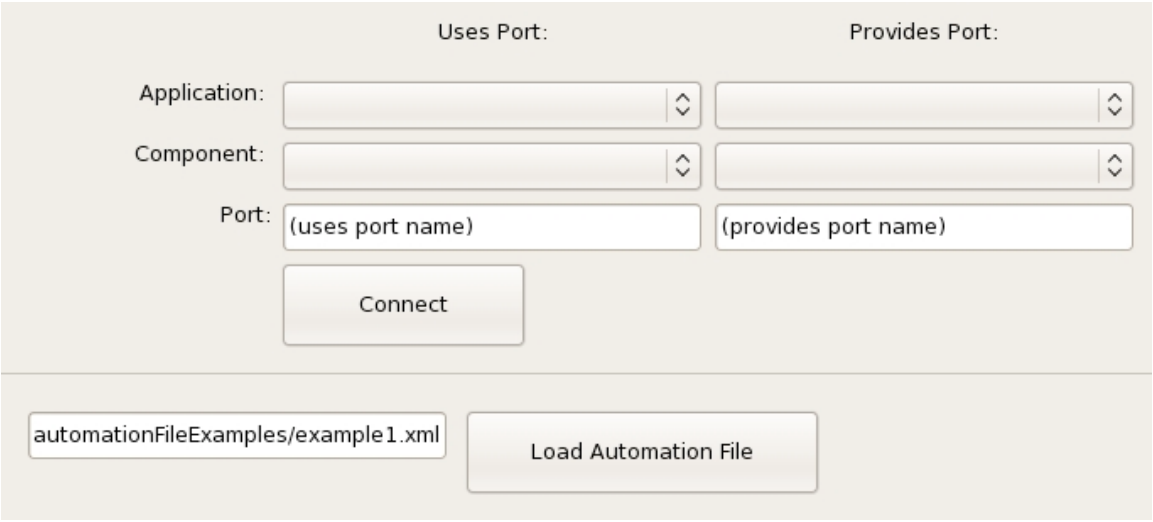

Figure 4.5: The Connect Tool GUI

4.3.1 Manual Operation

When the tool is started, a list of available component instantiations, and their corresponding (SCA) applications, is retrieved from the naming service. The user is able to specify both a producer (left) and consumer (right) from a set of drop down menus. Since the naming service does not provide a list of ports available in the Domain, the user must type the names of the desired *uses* and *provides* ports in the spaces available. When the "Connect" button is pressed, the connection is established.

4.3.2 Automating Connections Through the Use of XML

The Connect Tool provides the developer the ability to automate the connection process. By giving the developer this functionality, the tool can be used to automatically manage connections based on packet headers. With the Connect Tool's ability to manage the (SCA) applications as well, the adaptive/cognitive algorithm is able to interpret policy in order to appropriately manage the system's resources, but it requires no knowledge of the destination of the data or the rest of the system. In this manner, the logic involved in the Connect Tool does not need to be reproduced for every adaptive/cognitive data source that is developed. The connection tool implements the methods that should remain constant for the development of all adaptive systems that are constructed from encapsulated Waveform Applications.

XML [5] is the standard markup language used in the SCA [3]. By using XML for specifying the conditions for automating connections, the user is able to create radio systems through simple text editing. Since XML is designed to be platform and language independent, theses files can be created and parsed on other systems (e.g., a DSP) using other languages (e.g., C++).

Below the manual operation section of the Connect Tool GUI is a text box where a user is able to specify an XML file that dictates the rules for automating connections. To the right of this text editor is a button labeled "Load Automation File", which, as the name implies, will load the XML file that will dictate the Connect Tool's operation. The XML file allows the user to describe both "consumers" and "producers". The SCA defines a consumer as "a software component that can receive user data traffic [3]"; the SCA defines a producer as "a software component that can supply user data traffic [3]". Though the SCA defines these terms as types of components, the terms are loosely interpreted to include OSSIE tools that are not necessarily components. The consumers and producers specified are also classified by the (SCA) applications in which the component instantiations reside. As many producers and consumers as the user wishes to specify can be specified.

A "producer" XML node contains tags that will define the data source. A "name" tag (optional) allows the user to give the producer (or consumer, detailed below) a name for

the sake of clarity. The user can either specify a tool (e.g., the AWG), a Waveform Application, or a component as the data source.

The user has the option of specifying a "waveform" XML tag. This tag, optional, will indicate a Waveform Application to be installed to the Domain as an (SCA) application. A component instance name (defined in the waveform's SAD file) must be specified as the component where the port of interest resides. Finally, the *uses* port to connect to must also be specified.

If neither a tool nor a waveform is specified, the user can specify a component as the producer, which will be automatically installed to the Domain as an (SCA) application using the Compform Tool. A *uses* port to connect to must still be specified.

The specified "consumer(s)" must contain the same tags as the producer(s), with two additional tags. A "header" tag will identify which packets will be received by the given destination. For example, if the header is set to "1", all packets with the header "1" will be sent to this consumer. In the case that the header is negative, the corresponding consumer (SCA) application (corresponding to the absolute value of the packet header) will be uninstalled after the packet is sent. If a negative packet header is sent, and the corresponding consumer (SCA) application is not installed, it will be installed. Once installed, the data will be sent, and the consumer (SCA) application will then be uninstalled. In this case, the consumer (SCA) application is being installed and uninstalled for every packet sent. In practice, this installation and uninstallation for every

packet is not generally used, but it helps provide a "worst-case" observation for the amount of overhead created through these processes.

The "install_at_startup" tag (taking a value of *True* or *False*) indicates, as the name implies, whether or not the consumer (SCA) application will be installed at application startup. By setting the tag to *True*, the (SCA) application will be ready to accept data prior to the sending of the first packet, allowing the developer to implement a system with fixed (SCA) application installations (as described in Section 3.1). By setting the tag to *False*, the (SCA) application is not installed until the first packet is received, thus optimizing memory usage until data is present for the consumer (SCA) application to be processed.

4.3.3 Headers versus Metadata

Both headers and metadata can potentially be used. However, the method used for sending metadata to the Connect Tool is not completely defined since metadata is not defined in OSSIE version 0.6.2 [6]. In the event that metadata is sent with the packet (**pushPacketMetadata** is called), the header can be automatically inserted into the beginning of the packet for further processing. Inserting the header in this manner promotes code reuse. The only parts of the code that change are the Connect Tool's **pushPacketMetadata** method (found in the file loadAutomations.py) and the **pushPacketMetadata** method calls that are called on the port handles within the Connect Tool's **pushPacketMetadata** method.

Chapter 5

Example Implementations

As discussed in Sections 3.1 and 4.3.2, the Connect Tool can be used to create reconfigurable systems from modularized components and Waveform Applications. As discussed in Section 4.1, the AWG Tool can be used to test and profile these systems. In this chapter, multiple reconfigurable systems are simulated in order to provide insight into the minimum and maximum latency overheads achievable through fixed and variable (SCA) application installation, as well as the general memory requirements for each approach. Furthermore the number of packets that must be sent for unavoidable overhead caused by variable (SCA) application installation to become negligible is approximated. To a gain perspective of the magnitudes of the avoidable overhead as well as the overhead associated with the additional CORBA connection, they are related to the execution time of a common engineering algorithm, namely, the Fast Fourier Transform.

5.1 Methods for Testing

Two criteria are considered for determining system performance, execution time and memory usage. In order to measure execution time, the Python modules **cProfile** and **pstats** are utilized, and in order to measure memory usage, the program **top** (available on most Linux distributions, as well as OSX) is utilized.

5.1.1 Defining Overhead

In order to assess the maximum achievable overhead and minimum achievable overhead, the term "overhead" must first be defined. In general, overhead will be discussed as added latency to the system. Software developers commonly find themselves making a tradeoff between efficiency and robustness. The Connect Tool is designed to be robust, and as a result it is often slower than it could be. It is acknowledged that the code for the Connect Tool can be customized for a given application that requires less robustness to increase efficiency at the expense of flexibility. This overhead is classified as "avoidable." The overhead created by this tool is therefore not considered when calculating the minimum and maximum overheads for the reconfiguration approaches, though it is still measured.

Three sets of overhead are generalized as "unavoidable" overhead, overhead from the OSSIE Core Framework, overhead from the Python bindings for the OSSIE Core

Framework, and overhead from **omniORBpy**. Though it may be possible to reduce the overhead created by these software packages (being treated as "third party" software), their source code is considered outside the scope of the thesis. The simulations in this thesis therefore provide insight into the best one can do using these third party software packages.

5.1.2 Measuring Latency (cProfile)

The module **cProfile** is available in the Python standard library [10]. This module allows the user to determine the amount of time the Python interpreter spends in each function that is called by the function being profiled. This module is incorporated into the AWG Tool. With profiling turned on, the user is able to gather statistics on the running system. Of particular interest in these simulations is the amount of time spent sending data and the amount of time spent installing/uninstalling/connecting/disconnecting.

The module **pstats** [10] (also built into Python) allows the user to process the profile files generated by **cProfile**. The statistics processed can be sorted by time spent in each function, cumulative time spent in each function, as well as other criteria that are not utilized in this thesis. A detailed analysis of the output of the **pstats** module is detailed in Section 5.2.

This profiling method is implemented by creating an instance of the Profile class provided by **cProfile**. This class provides the method **runctx**, which inputs a function

name and parameters. In the case of the AWG, when profiling is enabled, the **runctx** method is used to call **pushPacket**. The resulting data collected will provide information pertaining to the time spent in the resulting function calls in Connect Tool.

5.1.3 Measuring Memory Usage (top)

The program **top** [11] is used in order to approximate memory used by the system. The use of this program is rather straightforward. Simply typing "top" into a terminal starts the program. The field MEM displays the amount of physical memory in use by a specific task. Though many other methods for measuring memory usage exist, this program will be sufficient for demonstrating that each identical component being run in OSSIE utilizes the same amount of memory.

5.1.4 Variations in System Performance

Systems developed under the SCA are designed to be as platform independent as possible. The tools and components used in this thesis can potentially be run on a variety of platforms. Variations in system performance across multiple platforms are unavoidable. In the case of running the applications on general purpose processors, the performance will vary based on processing speed, available memory, operating system, etc.. The data collected in the following simulations is taken in order to determine the relative amount of delay and memory usage that a developer might expect on average.

The simulations in this thesis are conducted on a personal computer running Fedora Core 8 Linux with a Pentium 4 processor and 2 GB of RAM. It is widely known that Fedora Core Linux utilizes virtual memory and as a result some variations in delay can be experienced. These variations can be dependent on a variety of factors, such as other applications running in the operating system.

5.2 Simulation for Measuring Install Latency

The ability to reconfigure a radio system comes at a price. With the systems discussed in this thesis, the price comes in the form of increased execution time. This increased execution time must be considered as it can lead to latency in the system. Several simulations are conducted in order to approximate the amount of latency that is added to a radio system under different reconfiguration scenarios. The first simulation (labeled "simulation 1" for reference) represents a best-case scenario in terms of latency added to the system by the reconfiguration method, while the second simulation (labeled "simulation 2" for reference) represents a worst-case scenario of latency added to the system through the reconfiguration method. The latencies of the reconfiguration method between the best-case and worst-case scenarios are analyzed in the set of simulations in Section 5.2.3. Finally, the latencies discovered in the simulations are related to the execution time of a Fast Fourier Transform (FFT). All simulations conducted are performed on the same processor, with the same operating system.

5.2.1 Simulation 1: Minimal Overhead Latency

Simulation 1 is created in order to explore latencies associated with implementing fixed installations of (SCA) applications via the Connect Tool. In this simulation, the AWG Tool is set up to send data with alternating headers to the Connect Tool, which in turn sends the data to the appropriate consumer. Both consumers are (SCA) applications containing a single Pass Data component (more information on the Pass Data component can be found in Appendix A). The Pass Data component is a component that accepts data into a buffer at its *provides* port and, as the name implies, simply passes the data to the *uses* port. A block diagram of the data flow for this simulation can be seen in Figure 5.1.

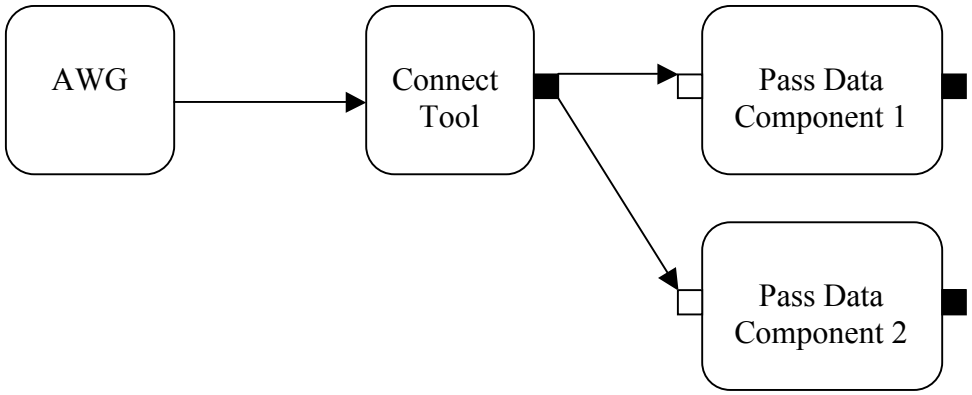

Figure 5.1: The Data Flow of Simulation 1. The connection between the AWG Tool and the Connect Tool is not a CORBA connection. The connections between the Connect Tool and the Pass Data components are CORBA connections.

The profiling feature of the AWG is activated. The resulting data from the Python profiler gives insight into the amount of time spent in each function of the simulation up

until the CORBA connections between the Connect Tool and either of the Pass Data components. The processing beyond the CORBA connections are not of interest for this simulation as the measurement of latencies associated with the Connect tool are being measured. In order to correctly interpret the results of the **pstats** module, the flow of function calls in the simulation must be understood. Figure 5.2 demonstrates this flow of function calls. Table 5.1 lists the time spent in each function as reported by **pstats**. The results of **pstats** as well as the flow of function calls are explained in greater detail following Figure 5.2 and Table 5.1.

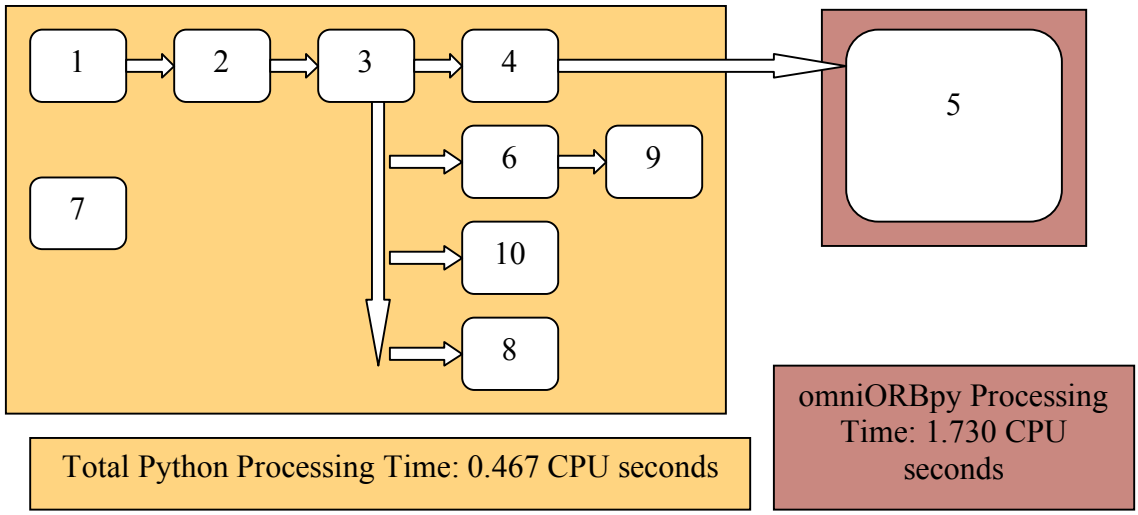

Figure 5.2: Processing Times Resulting from Simulation 1. Code blocks shown in the box on the left side (orange) represent Python processing in the AWG and Connect tools. Code blocks on the right side (maroon) represent **omniORBpy** processing time. Function 11 (not shown) is overhead associated with the Python profiler (0.006 CPU seconds). Functions corresponding to each number are listed in Table 5.1.

#	filename:lineno(function)	Ncalls	tottime	percall	Cumtime	percall
1	connectTool.py:442 (pushPacket)	5356	0.026	0.000	2.154	0.000
2	loadAutomationFile.py:320 (pushPacket)	5356	0.092	0.000	2.128	0.000
3	loadAutomationFile.py:338 (_pushPacket)	5356	0.164	0.000	2.037	0.000
4	complexShort_idl.py:191 (pushPacket)	5356	0.037	0.000	1.767	0.000
5	{_omnipy.invoke}	5356	1.730	0.000	1.730	0.000
6	loadAutomationFile.py:303 (stripHeader)	10712	0.072	0.000	0.083	0.000
7	<string>:1(<module>)	103	0.001	0.000	0.043	0.000
8	{abs}	10712	0.017	0.000	0.017	0.000
9	{len}	10712	0.011	0.000	0.011	0.000
10	{method 'has_key' of 'dict' objects}	5356	0.006	0.000	0.006	0.000
11	{method 'disable' of '_lsprof.Profiler' objects}	5356	0.006	0.000	0.006	0.000

Table 5.1: pstats Results from Simulation 1. The first column (labeled #) is added after **pstats** is run. This table demonstrates the relative amount of time spent in the AWG and Connect Tool code versus the amount of time spend sending the packet through CORBA. In more complex simulations this table is often substantially larger. Units are CPU seconds.

The field ncalls in Table 1 represents the number of times that a particular function or method is called during the experiment. The field tottime represents the amount of time it took the function "f" to run "exclusive to other functions that f called [10]". The first percall field represents the amount of time the function took to run (excluding execution time of sub functions called) once. In the example shown in Table 1, the functions are modularized enough that the amount of time spent in the function (excluding function calls) is less than the resolution provided by the **pstats** module. According to [10], the value of the first percall can be obtained to a higher degree of accuracy by simply dividing tottime by ncalls. This process is not necessary in this example since the cumulative time spent in all of the iterations is of interest. The field cumtime represents the cumulative amount of time spent in the function or method, throughout the entire simulation, including time spent in functions and methods called by the function or method in question. In this example, the time spent in connectTool.py's **pushPacket** function also includes the time spent in loadAutomationFile.py's **pushPacket** method since the first method calls the second. As expected, the cumulative time spent in connetTool.py's **pushPacket** method is greater than the cumulative time spent in loadAutomationFile.py's **pushPacket** method. The second percall method, similar to the first, is equal to the cumtime divided by ncalls. The last field indicates the name of the function as well as the line number where the function call is found. The column on the left (labeled #) is added to Table 1 in order to easily reference functions.

The data from Table 1 is categorized into time spent in **omniORBpy**, time spent in Python processing, and time spent in the Python profiler. The resulting distributions of

execution time can be seen in Figure 5.3. As seen in Figure 5.3, most of the processing time is spent sending data (via **ComplexShort_idl.py**) from the Connect Tool to the consumer component. Since most of the profiler overhead is removed during runtime [10], the observed profiler overhead is negligible.

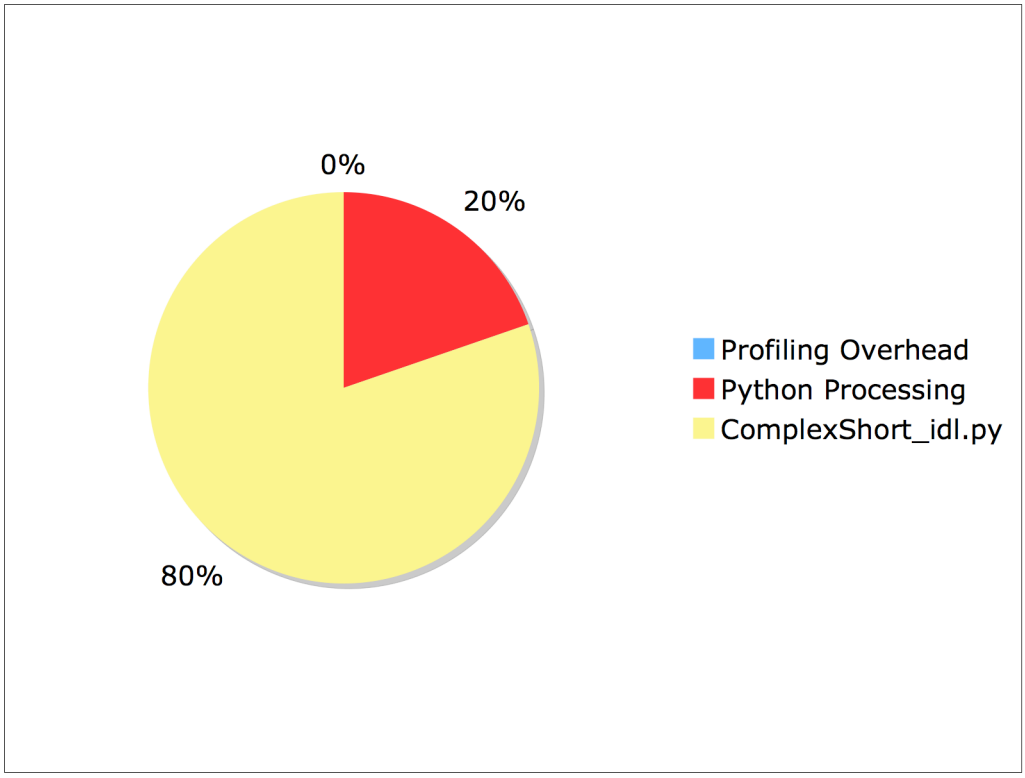

Figure 5.3: Breakdown of Execution Time for Simulation 1. Categorized as Python processing, profiling overhead, and **ComplexShort_idl.py**

5.2.2 Simulation 2: Maximum Overhead Latency

The amount of time spent installing/uninstalling/connecting/disconnecting obviously varies by the frequency in which the user performs the operations. In order to determine

an approximate "worst-case" overhead involved with the method, a simulation (labeled "simulation 2" for reference) is set up to measure the relative amount of overhead experienced when the consumer (SCA) application is installed every time a packet is sent.

In order to set up this simulation, an XML file is created to define the system. This XML file can be seen in Figure 5.4.

```
<automations>

   <producer>
      <name> first (and only) producer </name>
      <toolname> AWG </toolname>
   </producer>

   <consumer>
      <name> first (and only) consumer </name>
      <header> 1 </header>
      <install_at_startup> False </install_at_startup>
      <waveform> pass_data_waveform </waveform>
      <componentInstance> pass_data </componentInstance>
      <port> cshort_in </port>
   </consumer>

</automations>
```

Figure 5.4: The XML File Defining Simulation 2.

This XML file is loaded into the Connect Tool, which automatically installs and starts the necessary consumer (SCA) application (when the first packet is sent), as well as the AWG Tool (at startup). In the "Packet Headers" field of the AWG main window (see

Figure 4.1) "-1" is entered, indicating that the packets should all be sent to the consumer specified in the XML file (named "first (and only) consumer"). Since the header value is negative, the consumer (SCA) application will be installed/uninstalled/connected to/disconnected from every time a packet is sent. Once the "Packet Headers" field is set, the "Start" button in the AWG main window is pressed, and the simulation beings to run.

Three sources of execution time are considered: execution time required to install/uninstall/connect/disconnect (OSSIE Core Framework), execution time required to send the packet (via CORBA) from the Connect Tool to the consumer component, and execution time required for all other Python processing associated with the AWG and Connect Tools (loosely referred to as "Python overhead"). The resulting relative execution times are summarized in Figure 5.5.

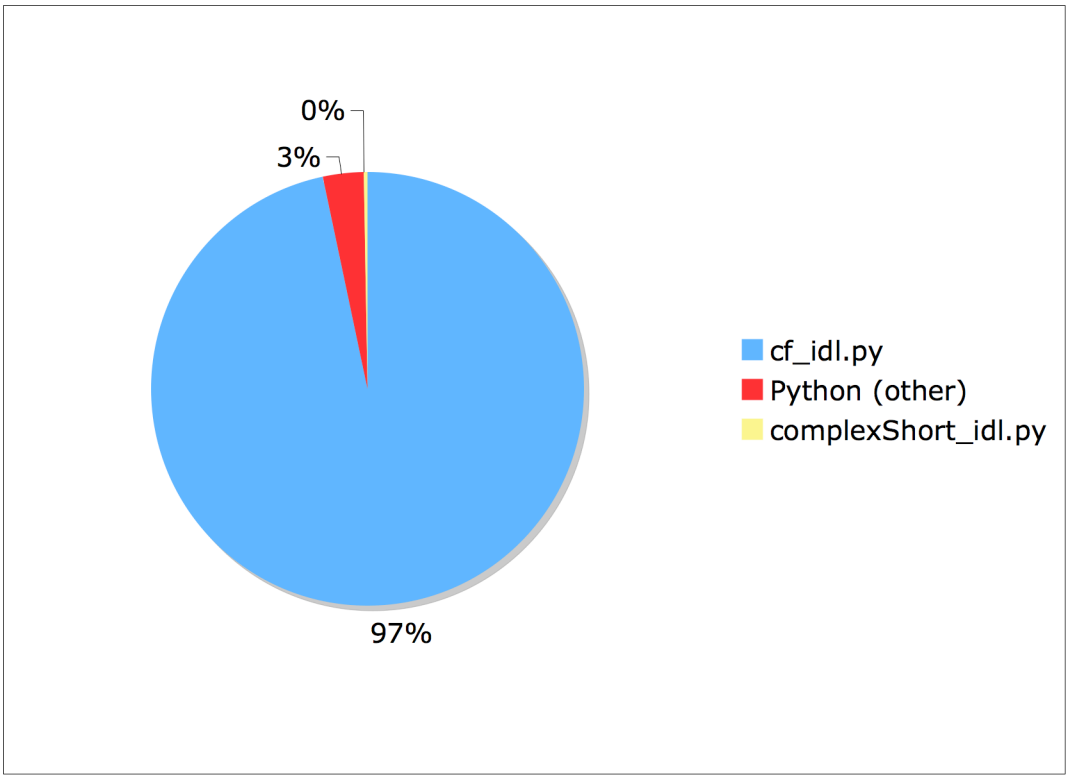

Figure 5.5: Breakdown of Execution Time for Simulation 2. Assumes that consumer (SCA) application is installed/uninstalled/connected to/disconnected from every time a packet is sent. The amount of time spent in **complexShort_idl.py** is minimized since the Pass Data component, as far as **complexShort_idl.py** is concerned, is merely loading the data into a buffer.

As expected, most of the execution time results from processing in the OSSIE Core Framework (as well as the Python bindings for the OSSIE Core Framework). This large amount of processing time is a result of the hefty requirements of installing the (SCA) application (explained in Section 3.3). Further analysis of the **pstats** output indicates that a vast majority of this time is spent in the Application Factory's **create** method.

Both simulations 1 and 2 involve overhead in the form of an additional CORBA connection. In the case of simulation 1, this overhead could be significant, depending on

the relative amount of processing occurring later in the system. I the case of simulation 2, the amount of extra execution time added by the extra CORBA overhead (represented by **complexShort_idl.py** in Figure 5.5) is negligible compared to the extra execution time added by the OSSIE Core Framework. In both cases, the amount of time spent in each for a different system can be found relative to the amount of time it takes to send a complex short packet of length 1024 through **complexShort_idl.py**. In the case of the second method analyzed, the amount of expected overhead, implied by simulation 2, should be more than 100 times the amount of time it takes the system to send a complex short CORBA packet of length 1024 through **complexShort_idl.py**. Similarly, since most of the overhead involved in the first simulation is a result of the extra CORBA connection, when implementing the first method discussed, the amount of expected overhead, implied by simulation 1, should be proportional to sending a single complex short CORBA packet of length 1024 through **complexShort_idl.py**.

5.2.3 Overhead Latency in Intermediate Cases

With the minimum and maximum performance capabilities of the reconfiguration methods established, it is natural to start considering the performance in between the two extremes. A set of simulations is conducted in a similar manner to simulation 1 and simulation 2. Similar to simulation 2, the consumer (SCA) application is not installed during system startup. In each simulation, the consumer (SCA) application is first installed, then sent a varying number of packets, and then uninstalled. As the number of packets sent between installations increases, the relative time spent in unavoidable

overhead decreases. The number of packets sent between installations is increased by a factor of two for each simulation until the relative time spent in unavoidable overhead becomes negligible. The resulting percentage of time spent in overhead for these simulations can be seen in Figure 5.6.

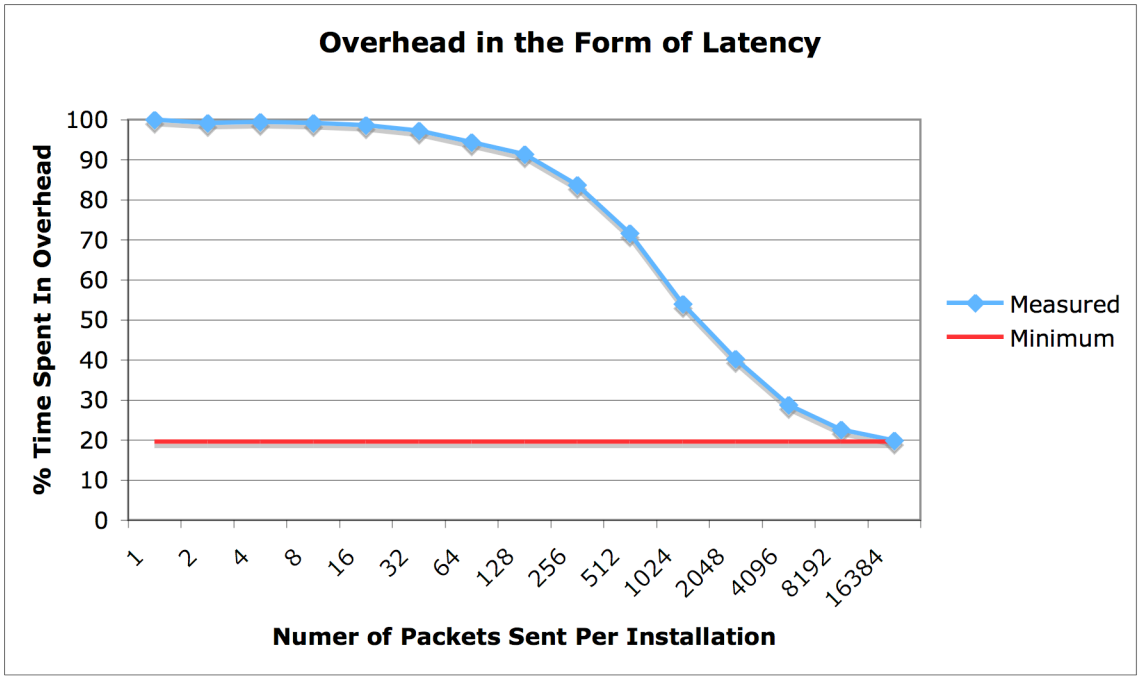

Figure 5.6: Percent of Time in Overhead for a Varying Number of Packets Sent per (SCA) Application Installation

The resulting percentages of time the simulation is in overhead (seen in Figure 5.6) is consistent with expectation. As with simulation 2, when a single packet is sent for each consumer (SCA) application installation, the overhead of the installation is nearly 100 %. As the number of packets sent between installations increases, the percentage of time spent in overhead asymptotically approaches the results from simulation 1. The relative

amount of time spent in necessary overhead is considered negligible (approximately 1% of overall overhead) when roughly 16384 packets are sent between installations.

The amount of time it takes to send 16384 packets of size 1024 is put into perspective by calculating the amount of time the data would take to be sent at two fairly common data rates. In a system processing data at 56 kbps (e.g., a typical dialup modem), 16384 complex short packets of length 1024 are sent in approximately 9.4 seconds. In a system processing data at 1.545 Mbps (e.g., a typical T1 modem), 16384 complex short packets of length 1024 are sent in approximately 0.34 seconds.

5.2.4 Relating Latency to FFTs

Though the latencies measured in simulations 1, simulation 2, as well as the intermediate cases can be put into perspective by comparing the amount of time it takes to send a CORBA packet using **complexShort_idl.py**, it is helpful to compare the execution times to a common algorithm in order to gain additional perspective. The FFT is chosen as a point of comparison because of its scalability (adjustable through FFT order) and its widespread use and availability in the engineering community. In order to perform a "fair" comparison, the FFT algorithm that is used should use compiled code (not interpreted Python), in this case C is used, with a Python interface (in a manner similar to the functionality of **omniORB**py [9], and the OSSIE Core Framework [6]). The FFT algorithm from the software package **numpy** is chosen to meet these criteria [12].

The system that is used in simulation 2 is run for 512 iterations (512 packets of length 1024) and compared to 512 iterations of the **numpy** FFT algorithm. The same Python profiler used in simulation 2 is used on the FFT algorithm. This process is repeated for FFT orders varying between 2^{10} and 2^{21}. The resulting execution times of the FFTs are compared to the execution times of **complexShort_idl.py** and the other overhead. The resulting comparisons are summarized in Figures 5.7 and 5.8.

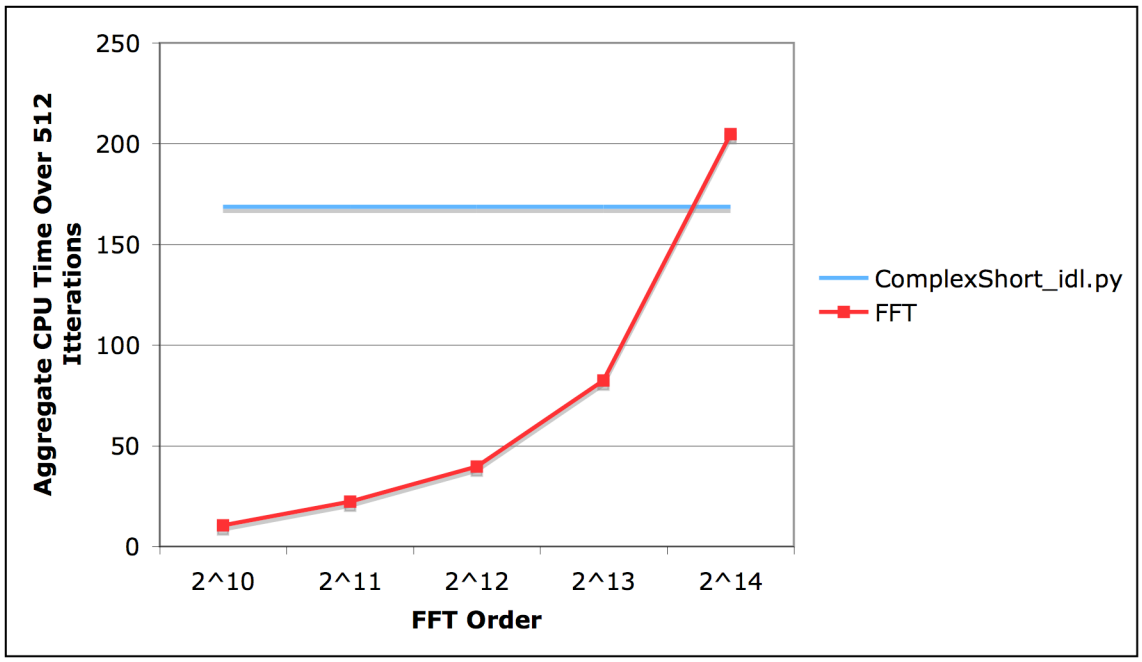

Figure 5.7: FFT Processing Times Compared to **complexShort_idl.py**.

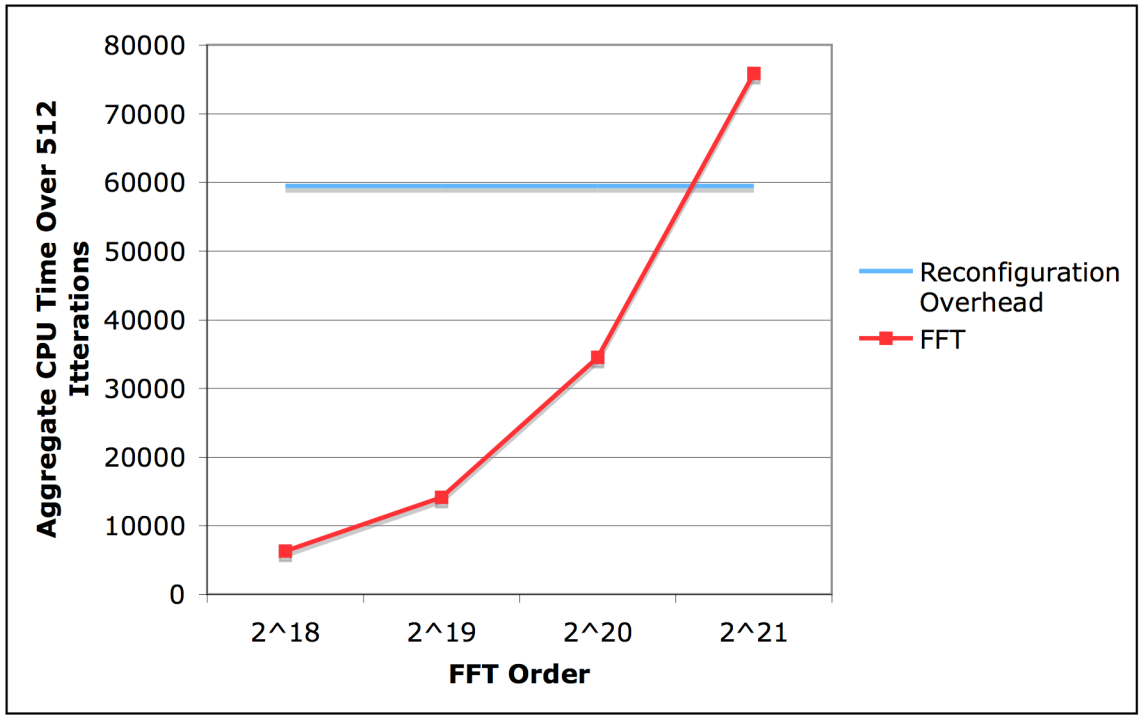

Figure 5.8: FFT Processing Times Compared to Reconfiguration Overhead

As seen in Figure 5.7, the amount of execution time required for sending a single complex short packet of length 1024 using **complexShort_idl.py** is roughly equivalent to the amount of execution time of performing a 2^{13}-2^{14} point FFT on that packet. As seen in Figure 5.8, the amount of execution time for the remainder of the reconfiguration process (Python overhead, Core Framework overhead, etc.) is roughly equivalent to taking a 2^{20}-2^{21} point FFT on the data packet. Though a 2^{13}-2^{14} point FFT could conceivably be used in a realistic system, a 2^{20}-2^{21} point FFT is clearly an excessive amount of processing for a real-time communications system. The excessive amount of processing time associated with installing (SCA) applications with each packet implies that this extreme approach (restarting an (SCA) application for every packet of data) is not likely feasible for a real-time system.

5.3 Simulations For Measuring Memory Usage

When designing a SDR, it is important to consider memory requirements. By minimizing memory requirements, the SDR can be manufactured at lower cost. Three simulations are run in order to determine how much memory is saved through minimizing the number of (SCA) applications running in the Domain. These simulations validate that, as expected, the amount of memory required is directly proportional to the number of Python processes running, which in these cases corresponds to the number of components running in OSSIE.

5.3.1 Memory Usage of Simulation 1 and Simulation 2

A set of simulations is run in order to approximate the memory usage of simulations 1 and 2. The Python profiler for both systems is disabled. The program **top** is run while the simulations are running. In the case of both simulations, top indicates that 1.7% of the system memory is always allocated to python, regardless of the number of (SCA) applications installed. On the system that the simulation is being run on, this 1.7% corresponds to 35 megabytes of RAM. This memory is clearly being utilized by ALF and the ALF tools. Each instance of the Pass Data (SCA) application installed adds an additional Python process that occupies 0.3 % of the system memory. On the system the simulation is run on, this 0.3 % corresponds to 6 megabytes of RAM. These Python processes represent the Pass Data components. Obviously, as the complexity of the components, as well as the number of the components in each (SCA) application, increase the memory usage will also increase.

No additional memory usage is added as a result of the ORBs for the ALF tools or the components. As discussed in Section 3.2.1 the POA for each of the ORBs frees the memory used when it is inactive. With both simulations the memory usages of GPP and **nodeBooter** are constant.

5.3.2 Scalability of the Memory Usage

An additional simulation is conducted in order to determine the scalability of the results. A simulation similar to simulation 1 is created that has 10 consumers instead of 2 consumers. As expected, 11 Python process result: 1 for ALF and the ALF tools, and 1 for each instance of the Pass Data component. Also, as expected, each additional Python process started by the Pass Data component consumes 6 megabytes of RAM.

Chapter 6

Conclusions and Future Work

The SCA provides a set of definitions that allows for the development of standardized SDRs. With implementations of the SCA standard (i.e., the OSSIE Core Framework), as well as the CORBA standard (i.e., **omniORB**), the design process itself becomes modular, saving the developer time as long as the standards are appropriately understood and followed. However, an advanced understanding of how to implement the standards is not necessarily needed in order to develop SDRs.

The OSSIE tool set assists the developer in creating SCA based applications on multiple levels. OWD provides a means for developing necessary code and XML profiles for components, as well as an environment for constructing these components into Waveform Applications. Through the use of ALF, these Waveform Applications, and even the components themselves, can be effortlessly installed and run as (SCA) applications. Through the addition of the Connect Tool, OSSIE provides a means to develop reconfigurable radio systems using encapsulated components and encapsulated Waveform Applications that are developed by the aforementioned rapid prototyping

tools. ALF also provides the ability for the developer to debug and analyze the performance of these systems through the use of ALF tools, such as the AWG Tool.

The development of several OSSIE tools, specifically the Connect Tool, opens the doors for development of many currently unanticipated radio systems. As policies are created for these radio systems, the implementation of these policies can utilize the OSSIE tools or utilities with similar functionality in order to reconfigure the system appropriately. Since all policies are inherently different, the advantages of using the reconfiguration methods discussed in this thesis can be analyzed on a case-by-case basis with the AWG Tool's profiling ability.

In a single mode radio overhead associated with switching between operating modes is obviously non-existent. The simulations in Chapter 5 prove that the act of reconfiguring the radio system involves overhead in the form of latency, which can in some cases be substantial. It has been proven that, with the implementation described in this thesis, the overhead in the form of latency is directly proportional to the frequency in which (SCA) applications are installed/uninstalled/connected/disconnected. A single mode radio will therefore always be more efficient in this regard. However, the benefits of having a multimode, dynamically reconfigurable radio cannot be ignored. By being able to switch between operating modes, the radio has to potential to be more power efficient and spectrally efficient. The need for power efficiency is becoming increasingly important in modern radio design because of the desire to make smaller, more portable radios. Also,

the need for more spectrally efficient radio systems is apparent because of the vast number of wireless radio systems in existence.

This substantial overhead associated with installing an (SCA) application is both a reflection of the hefty requirements of the SCA's definition of the installation process (specifically Application Factory's **create** method) as well as the OSSIE Core Framework implementation itself. Whether or not optimization of the OSSIE Core Framework will significantly improve the (SCA) application installation process (in terms of reduced execution time) requires additional extensive research. Also, the SCA's requirements for this installation procedure should be thoroughly investigated. Though the SCA provides many benefits towards rapidly building systems from existing code, for some applications the benefits might not outweigh the significant overhead the SCA imposes.

As discussed in [4], there currently exists no standard in the SCA for defining the interconnection between (SCA) applications. Further investigation into the ideal standard for this concept needs to be explored. In the event that a solution is selected for the next SCA version, the advantages and disadvantages of updating the OSSIE Core Framework and OSSIE Tools should be explored, and implementations of the new standard should be considered. The existence of the reconfiguration methods available in the Connect Tool can provide a baseline for this research.

The maximum throughput of the entire system is, of course, impacted by the size of the CORBA packets. The effects on the relative overheads when using different packet sizes can be analyzed.

Appendix A

Pass Data Component

The Pass Data component is developed with the aid of OWD. The component contains one *uses* port of type complex short, and one *provides* port of type complex short. This component is designed to be as simple as possible in order to isolate overhead associated with the SCA implementation. Signal processing is not present in this component. The component simply accepts data at its *provides* port, stores the data in a buffer, and sends the data to the *uses* port. The use of Python allows the use of **omniORBpy**, which allows the reader to relate the CORBA connection used by the tools (also **omniORBpy**) on an appropriate scale ("an apples and apples" type comparison).

In OSSIE 0.6.2 ALF lacks the ability to easily connect ALF tools to devices or other ALF tools; however, the Pass Data component's minimal overhead and *uses* and *provides* ports gives the developer an indirect method for using the ALF tools on devices as well as other ALF tools. To use the Pass Data component to aid in connecting an ALF tool to a device, a Waveform Application is created in which the device's *uses* port is connected to the Pass Data component's *provides* port. Likewise, the Pass Data component's *uses*

port can be connected to the device's *provides* port. In order to view data being sent from the device's *uses* port, the developer connects the desired tool (e.g., the Plot tool) to the Pass Data's *uses* port, which is sending the same information.

In order to connect two ALF tools to each other, the Pass Data component can be launched as an (SCA) application using the Compform Tool (discussed in Section 4.2). Once the (SCA) application exists, an ALF tool sending data (e.g., the AWG Tool) can be connected to the Pass Data component's *provides* port, and the ALF tool receiving data (e.g., the Plot Tool) can be connected to the Pass Data component's *uses* port. With these two connections established, all data sent from the ALF Tool sending data will be received by the ALF Tool receiving data.

Bibliography

[1] J.H. Reed, *Software Radio: A Modern Approach to Radio Engineering*, 1st ed. New Jersey: Prentice Hall, 2002.

[2] Drew Cormier, Jacob A. DePriest, Carl Dietrich, Max Robert, and Jeff Reed, "OSSIE Rapid Prototyping Tools Help Easily Reconfigure Software Defined Radios," Mobile Handset DesignLine, February 5, 2007, Available: http://mobilehandsetdesignline.com

[3] Software Communications Architecture Specification, 2nd ed., Joint Tactical Radio System (JTRS) Joint Program Office, April 2004.

[4] F. Lévesque, S. Bernier, "Interconnecting JTRS SCA Applications," SDR Forum Technical Conference 2007, Denver, CO, November, 2007.

[5] [Online]. Available: http://www.omg.org

[6] [Online]. Available: http://www.ossie.wireless.vt.edu

[7] DePriest, Jacob A. 2006. "A Practical Approach to Rapid Prototyping of SCA Waveforms," Masters Thesis, Virginia Polytechnic Institute and State University. 108 p.

[8] Drew Cormier, Carl Dietrich, "Debugging Strategies for SCA Components and Waveforms," SDR Forum Technical Conference 2007, Denver, CO, November, 2007.

[9] [Online]. Available: http://omniORB.sourceforge.net

[10] [Online]. Available: http://www.python.org/doc

[11] [Online]. Available: http://www.linux.org/docs

[12] [Online]. Available: http://numpy.scipy.org/