SOFTWARE DEFINED RADIO ARCHITECTURES EVALUATION

Álvaro Palomo Navarro (NUI Maynooth, Co. Kildare, Ireland, apalomo@eeng.nuim.ie); Rudi Villing (NUI Maynooth, Co. Kildare, Ireland, rvilling@eeng.nuim.ie); and Ronan Farrell (NUI Maynooth, Co. Kildare, Ireland, rfarrell@eeng.nuim.ie)

ABSTRACT

This paper presents an performance evaluation of GNU Radio and OSSIE, two open source Software Defined Radio (SDR) architectures. The two architectures were compared by running implementations of a BPSK waveform utilising a software loopback channel on each. The upper bound full duplex throughput was found to be around 700kbps in both cases, though OSSIE was slightly faster than GNU Radio. CPU and memory loads did not differ significantly.

1. INTRODUCTION

Perhaps the most important feature of software defined radio is its flexibility. Flexibility can take many forms including dynamic run-time reconfigurability in response to network changes (over a period of seconds or less), flexibility to support new waveforms through firmware updates (with interval between updates usually from weeks to years), and flexibility to use common radio hardware in a family of devices, with different features and computational capabilities. Unfortunately, flexibility often comes at either a significant monetary or performance cost. The purpose of this study is to gain some insight into the performance of software defined radios which use a general purpose processor for implementing all baseband signal processing.

Two readily available open source frameworks for software defined radio (SDR) are GNU Radio [1] and OSSIE [2]. GNU Radio is a software application for building and deploying SDR systems under a GNU General Public License. It was initially developed by the Massachusetts Institute of Technology (MIT) under the Spectrum Ware project [3] but it has undergone substantial development since then. It provides a number of signal processing modules written in C++ language, which are interconnected and configured using Python [4]. The GNU Radio includes modules such as basic signal processing elements, timing recovery and synchronization.

The use of Python provides to the design the benefits of object-oriented programming with the ease of an interpreted language, i.e. it can be recompiled during runtime. As a disadvantage, its execution speed might not be as fast as other compiled languages like C++.

OSSIE (Open Source SCA Implementation: Embedded) is an SDR implementation of the Joint Tactical Radio System (JTRS) Software Communications Architecture (SCA) [5] developed by Virginia Tech University for educational use as well as for research applications using software defined radio in 2004. SCA also decomposes a waveform application into components which might be reusable by different waveforms. Unlike GNU Radio, the interconnection, interoperation and properties of the blocks are configured using XML [6] files. More significantly, components are interconnected using a CORBA middleware [7]. This provides the flexibility to transparently host components in different processing nodes at the expense of increased complexity.

The latency between OSSIE inter-component communications has been studied previously (for example [8]) identifying the key factors and proposing which contribute to it. [9] studied the overhead associated with CORBA inter-communication demonstrating that the CPU load produced by the CORBA ORB is significally smaller than the one produced by the signal processing modules of the waveform. OSSIE memory usage is also studied in [10] and [11] measured on a desktop computer implementing full duplex and half duplex applications respectively. The latency of GNU Radio has also been studied over a completely transceiver chain using the USRP (Universal Software Radio Peripheral) board [12] as hardware testbed [13] concluding that the system is not suitable for real time systems in the Mbit range.

It is useful to consider the upper limit on throughput performance that can be achieved by particular SDR framework. The absolute upper limit may ultimately be constrained either by computation or input/output bottlenecks. In this study, we examine the performance of GNU Radio and OSSIE on the same hardware platform when a full-duplex communications system is implemented. The comparison investigates the CPU load, the memory usage and the overall data throughput for each system.

The remaining sections of this paper address the methodology used, the results and finally our conclusions.

2. METHOD

The hardware platform for the performance tests consisted of a desktop computer with a 3GHz Intel Pentium 4 CPU

and 1024MB of RAM. The operating system was Ubuntu 7.1 [14], chosen principally for simplicity of installing GNU Radio from its repositories. The operating system itself was installed and configured for normal desktop use without any additional performance configuration. The framework versions used for the evaluation were GNU Radio 3.1.2 and OSSIE 0.6.2.

In order to focus the performance evaluation on the frameworks and not the specific modulation scheme in use a simple and computationally lightweight waveform application was desirable. A secondary factor, which influenced the ease with which a test application could be developed, was the availability of components that implemented all or part of this modulation scheme. The GNU Radio libraries contain a large number of C++ and Python modules implementing a variety of different components and reusable functions which facilitates rapid development of waveform applications. On the other hand, the current version of OSSIE (0.6.2) is distributed with a rather limited set of components. Many components created for previous versions of OSSIE are not compatible with the current framework and require customization. Furthermore, due to its nature as a research and education tool, many of the modules are designated "experimental" status. Binary phase-shift keying (BPSK) was eventually chosen as the modulation scheme and waveform applications implementing this scheme were implemented in both frameworks. The common test application structure used is depicted in Figure 1.

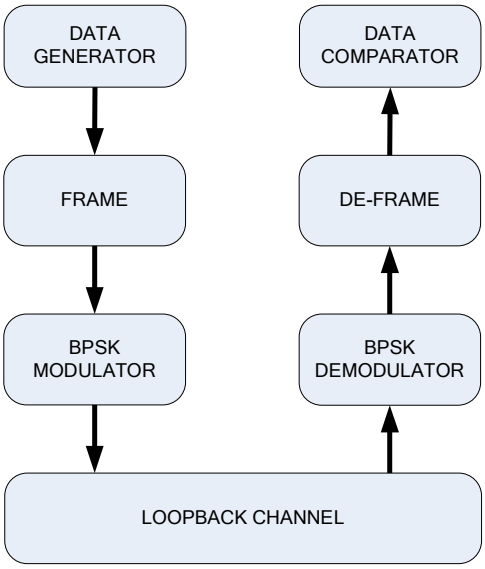

Figure 1 Test waveform application structure used with both the OSSIE and GNU Radio frameworks.

In the OSSIE implementation, the Data Generator generates packets by copying a preallocated data block representing 400 application data bits. After framing, the packet size is 512 bits. (It is important to comment that many of the OSSIE components used were designed with the assumption that packets would only be 512 bits and lifting this restriction would have required more code modification than was worthwhile for this evaluation.) The loopback channel copied the received bits without introducing any kind of channel noise, attenuation, or phase shift. Finally the Demodulator, Depacketizer and Data Comparator implement the receive path.

The GNU Radio implementation is similar. Unlike the OSSIE implementation, data packets are generated dynamically, using the last bits of the packet number as the packet data. During the framing operation a preamble, an access code, a heading, and a cyclic redundancy code (CRC) are attached in every packet. The BPSK modulator and demodulator were implemented using the DBPSK (Differential BPSK) Python module of the GNU Radio blocks library by disabling the differential encoder and differential phase detector features. Since transmission is not done over-the-air (OTA), the use of an interpolator and a root raised cosine (RRC) filter in the transmitter as well as a time-recovery module, a RRC filter and decimator in the receiver was not necessary and these functions and components were not part of the test application.

It was intended that both the GNU Radio and OSSIE test applications would be tested in a number of configurations. Unfortunately, the OSSIE test application was only tested in one configuration due to hard-coded assumptions about packet size distributed throughout the application. OSSIE was tested with just one packet size (64 bytes) while GNU Radio was tested with 64, 256 and 1024 bytes.

Some differences between the test applications, particularly in the area of framing, remained so the evaluation was based on the amount of packet data (consisting of original application data plus any additional framing and control data) transmitted to/received from the modulator/demodulator components. In both test applications the size of the simulated application packet size was adjusted so that the framed version was as specified for the configuration under test.

The total amount of (framed) data to be transmitted during a test was a constant 10MB for all configurations. However differences in framing mean that the number of packets transmitted may have differed slightly between the two test applications.

3. RESULTS AND DISCUSSION

Maximum throughput was estimated for each of the test configurations, defined by a combination of SDR framework

and packet size. Both memory and CPU loads were estimated from several measurements made during just one test configuration for each SDR framework. Finally the GNU radio test application was profiled to gain additional insight into the location and nature of the performance hot spots where most of the computation effort was expended. A similar exercise was not carried out for OSSIE and could be substantially more complex due to the multiple process implementation of a typical OSSIE waveform application.

3.1 Throughput

Total processing time required to transmit and receive 10MB of data (inclusive of framing and control information) was measured for each of the test configurations and used to calculate the throughput. This throughput is a first order estimate of the maximum full duplex throughput achievable in each test configuration, since each application is processing data transmission and reception in parallel.

In the OSSIE test application, total processing time was recorded using the standard `gettimeofday` function in the transmitting data generator and receiving data comparator code. In the GNU Radio application, the same information was obtained using the standard `time` function from the module of the same name introduced into the main() function of the Python's main module.

Table 1. Estimated maximum full duplex throughput per packet size for each test configuration.

Framework	Packet Size (bytes)	Throughput (Mbps)
OSSIE	64	0.72
GNU Radio	64	0.59
GNU Radio	256	0.68
GNU Radio	1024	0.71

The main results for each of the four test configurations are shown in Table 1. It is apparent that GNU Radio throughput depends on packet size. The relationship does not appear to be linear, however. OSSIE at the smallest packet size outperforms GNU Radio at all sizes, though the performance difference is not much when GNU Radio uses larger packets.

In a real SDR, the packet sizes communicated between the software components will be affected by requirements of the MAC layer frame size, RF front end packet size and latency requirements. As such, the packet size may not be a tunable parameter. However, the results above would suggest that where the choice exists, a bigger packet size is to be preferred, at least for GNU Radio.

The results also indicate surprisingly low values for estimated maximum throughput possible on both GNU Radio and OSSIE. It is probably reasonable to assume that the full duplex throughput could be close to doubled in half duplex operation, since the transmit and receive paths have fairly symmetric implementations (at least for the simple test applications in use).

The modulation scheme used for testing was explicitly chosen for its computational simplicity, so that the computational burden would be dominated by architectural and framework features rather than modulation calculations. It is safe to assume that a more complex modulation scheme (for example OFDM or OFDMA) would achieve lower throughput. It is also safe to assume that communication with a separate RF board would further reduce throughput.

It seems likely that each of the frameworks has strengths and weaknesses affecting their throughput performance. It is likely that inter-process communication, in the form of CORBA calls between components, adversely affects the performance of OSSIE, while GNU Radio seems to have the advantage of executing all components in the context of a single process and address space allowing inter-communication by direct function calls. Conversely, within-component code, implemented in C++ in OSSIE can be expected to run faster than the mix of interpreted Python and compiled C++ used in GNU Radio.

Finally it should be noted that optimizing either the component or framework performance was outside the scope of the current study, whose purpose was to estimate the maximum performance available from the frameworks in their default state. For this reason existing components were used and customized where necessary and it should be noted that some of these components are explicitly of demonstration or experimental quality rather than optimized, production quality. Nevertheless, we were curious about where optimizations might be made in the future, so we decided to examine the distribution of computational for GNU Radio.

3.2 Computation profile

GNU Radio was profiled using the cProfile [15] tool in each of the test configurations for which throughput was reported. The results were somewhat inconclusive and so a modified test configuration where the number of packets, rather than the total amount of data, was fixed. The main results are shown in Table 2.

Table 2. Total computation time and computation time of two most expensive functions for GNU Radio test application: `gr_py_msg_queue_insert_tail` (insert queue) and `conv_0_1_string_to_packed_binary_string` (pack binary).

Condition	Packet Size (bytes)	Total (secs)	Insert Queue (secs)	Pack binary (secs)
10 MB data	64	155.9	48.6	47.4
	256	127.5	100.6	11.7
	1024	118.8	111.3	3.0
10000 packets	64	10.1	3.2	3.0
	256	32.7	25.4	3.0
	1024	125.4	117.6	3.2

The computation hot spots in this test application were the functions `gr_py_msg_queue_insert_tail`, which is used to transmit packets from one component to the next, and `conv_0_1_string_to_packed_binary_string`, which is used to convert a string consisting only of the characters '1' and '0' (that is, the information content is 1 bit per byte) to a string with any valid 8 bit character (that is the information content is 8 bits per byte). This latter function is used primarily to attach the access code and preamble bits to a packet.

When the total amount of data transmitted was constant, the time spent communicating packets between components increased slowly and non-linearly with packet size. The time spent creating packed binary strings decreased rapidly and approximately linearly with packet size.

When the total number of packets transmitted was constant, the time spent creating packed binary strings remained essentially constant. This is to be expected because the number of times this function is invoked is proportional to the number of packets and work to be done does not depend on packet size. In contrast the time spent communicating packets increased in a linear fashion with packet size.

In summary, profiling indicated that most of the computational effort was expended simply moving data between components and not performing the actual transformations within each component. While the balance of effort would change with a more computationally intensive modulation scheme, it nevertheless seems worthwhile to investigate optimization of the component intercommunication in the future.

3.3 CPU load

The CPU load was monitored, using the Linux `top` command, during the execution of GNU Radio and OSSIE test applications in the 64 byte packet condition. In both cases the CPU load was essentially 100%. In the case of OSSIE, individual components, implemented as separate processes, typically consumed 10 to 20% of available CPU load each. The sum of the CPU loads for OSSIE components exceeded 100%, but this is an artifact of the sampling mechanism used by the top command to measure CPU load.

The CPU load indicates, as expected, that there is no source of non-computation delays inherent in either framework. A real SDR application with an external RF front end, would exhibit non-computation delays (or operating system device driver delays) due to the external interface.

With all other factors being equal, it seems reasonable to assume that the throughput achievable in either framework will scale linearly with computation speed of the processor in use. Given that a 3GHz Pentium 4 can achieve a maximum throughput of about 700kbps, an embedded processor (perhaps an earlier generation processor or one that has a lower clock speed) with 14% of the processing power could achieve just 100kbps in the absence of any further optimization. Significant optimization would clearly be required for all but the simplest systems.

3.4 Memory load

The `exmap` tool [16] was used to monitor the memory consumed during the execution of GNU Radio and OSSIE test applications in the 64 byte packet condition.

Determining memory consumption is complicated by the sophisticated memory management schemes employed by modern operating systems that enable a process or set of processes to run successfully despite requiring more memory than is physically available.

Virtual memory size is the total address space that is allocated to a process. It includes not only the amount of memory required by the process in its current mode of operation, but also any memory requested by the process to date, whether or not that memory has been subsequently used. Buffer space that has been allocated, but not yet required falls into this category.

Mapped memory size is amount of virtual address space that has ever been used. (This is a slight simplification as read-only memory such as code sections can be unmapped under memory pressure situations.) Once used, mapped memory is either resident in RAM or temporarily stored on disk in a paging or swap file. In an embedded system without a disk, all mapped memory would have to be resident in RAM.

Substantial portions of memory can be shared between processes. For example all processes using a shared library will share the code sections, though they will generally have private data sections. Inter-process communication frequently makes use of shared data memory. For this reason, both the virtual memory size the mapped memory

size for a set of processes will count the same memory multiple time giving a summed total that overestimates the memory really required by the set of processes. The effective mapped memory is an adjusted figure which assumes that N processes sharing a piece of memory really only use a fraction 1/N each and using this correction, the sum is a more accurate reflection of the true memory needs.

The GNU Radio virtual memory allocation was 53MB of which 20MB was actually mapped and just 17MB effectively mapped to GNU Radio. The OSSIE virtual memory allocation (summed across all processes used) was 690MB, of which 67MB was mapped and just 17MB was effectively mapped to OSSIE processes.

It would appear that both frameworks require approximately the same amount of memory for the test application evaluated and that either application could run within a 32 to 64MB memory.

4. CONCLUSIONS

Both OSSIE and GNU Radio require some customization in order to run even a simple loopback performance benchmark. Such a benchmark (assuming it is implemented in the manner of typical waveform applications on the framework in question) is a reasonable estimator of an upper performance limit on the throughput performance of that framework. Real SDR applications will always achieve lower performance than this limit.

The evaluation indicated that, without any specific effort spent optimizing the application, the upper limit of throughput performance for both OSSIE and GNU Radio was surprisingly low (around 700kbps), although OSSIE was the faster of the two by a narrow margin. Both frameworks load the CPU and memory in an essentially equal manner.

Future work may include optimization of the test application itself followed by further study of the computation profile to determine if and how the frameworks themselves could be further optimized.

5. ACKNOWLEDGEMENTS

The authors wish to thank Jean-Christophe Schiel and François Montaigne for their assistance and support. Also the authors extend thanks to the sponsors EADS and IRCSET for the PhD program.

6. REFERENCES

[1] "GNU Radio – The GNU Software Radio", http://www.gnu.org/software/gnuradio/.
[2] Wireless@VirginiaTech, "OSSIE", http://ossie.wireless.vt.edu/trac/.
[3] V. Bose, "Design and Implementation of Software Radios Using a General Purpose Processor", Massachusetts Institute of Technology, PhD, 1999.
[4] "Python Programming Language Official Website", http://www.python.org/.
[5] "Software Communications Architecture Specification", Joint Tactical Radio System (JTRS) Joint Program Office, Version 2.2.2, May 2006.
[6] "Extensible Markup Language (XML)", http://www.w3.org/xml.
[7] "The OMG´s CORBA Website", http://www.corba.org/.
[8] T. Tsou, P. Ballister, and J.H. Reed, "Latency Profiling for SCA Software Radio", SDR Forum Technical Conference, Denver, CO, November 2-4, 2007.
[9] P.J. Balister, M. Robert, J.H. Reed, "Impact of the Use of CORBA for Inter-Component Communications in SCA Based Radio", SDR Forum Technical Conference, 2006.
[10] P.J. Balister, C. Dietrich, J.H. Reed, "Memory Usage of a Software Communications Architecture Waveform", SDR Forum Technical Conference, Denver, CO, November 2-4, 2007.
[11] P.J. Balister, "A Software Defined Radio Implemented Using the OSSIE Core Framework Deployed on a TI OMAP Processor", M.Eng.Sc, December 4th , 2007, Blacksburg, Virginia.
[12] "GNU Radio – Universal Software Radio Peripheral" http://gnuradio.org/trac/wiki?UniversalSoftwareRadioPeripheral/.
[13] S. Valentin, H. von Malm, and H. Karl, "Evaluating the GNU Software Radio for Wireless Testbeds", Technical Report TR-RI-06-273, University of Paderborn, February 2006.
[14] "Ubuntu Official Website", http://www.ubuntu.com/.
[15] "cProfile Reference Manual", http://docs.python.org/lib/module-profile.html/.
[16] "Exmap Official Website", http://www.berthels.co.uk/exmap/