OSSIE/GNU RADIO GENERIC COMPONENT PERFORMANCE ANALYSIS OVER DISTRIBUTED SYSTEMS

ABSTRACT

The OSSIE/GNU Radio Generic Component (GC), demonstrated at SDR '10, integrates the OSSIE and GNU Radio (GR) open source Software-Defined Radio (SDR) platforms. The GC can encapsulate one or more GR blocks in a flowgraph, and can reconfigure the encapsulated flowgraph and GR block properties in near real-time. It can run in a standalone mode for GR testing and can also run as a component in an OSSIE waveform, interfacing the encapsulated GR flowgraph with other OSSIE components. Recent enhancements to the GC include use of Cython in the processing core to accelerate execution over the initial Python-only implementation and encapsulation of GR GUI sinks (fftsink2, scopesink2, and waterfallsink2) in the optional GC GUI. GC performance was benchmarked and component resource use profiled in multi-node distributed SDR waveform applications. The results show that using GR components within the GC in a distributed environment is comparable in latency to using analogous OSSIE components. The Universal Software Radio Peripheral 2 (USRP2) was used in close conjunction with the GC for analysis and evaluation.

1. INTRODUCTION

The OSSIE/GNU Radio Generic Component (GC) demonstrated at SDR '10 combines both the granularity provided by GNU Radio and the capability of distributed applications provided by OSSIE. The GC also allows for the reconfiguration of internal GR flowgraphs during runtime, making the GC a powerful and flexible addition to the OSSIE waveform development. To combine the two platforms, the GC was originally written in Python alone because GR flowgraphs can be created in Python and OSSIE components can be implemented in Python. However, Python is not the ideal language for use in the critical path of a signal processing application due to the inherent slowness of a scripting language. For that reason, the processing core of the GC was re-written in Cython in hopes that the overhead that comes with increased functionality would be mitigated. Cython allows for the declaration of C types as well as using C-extensions in Python that allow Python programs to run at speeds much closer to that of C++ programs. Smaller tests have shown this to be very successful in terms of increasing the throughput of the GC to speeds closer to its limit, which is an OSSIE component implemented in Python that simply passes data without processing [1]. To test the utility of the GC's capabilities against its overheard, two distributed waveforms were created to compare standard OSSIE solutions to GC solutions. Since OSSIE and GR are working in series, the GC can only be as fast as the slowest of these two platforms in any given environment. GR has been shown to be faster than OSSIE in some environments and slower in others [2][3][4]. The GC waveform capitalizes on the ability to encapsulate many DSP blocks in a single GR flowgraph within the GC as opposed to the OSSIE waveform, which contains numerous OSSIE components to accomplish the same task.

2. METHODS

The GC was tested on Virgnia Tech's Cognitive Radio Network Testbed (CORNET) using two nodes, each equipped with an USRP2. Each node has a Quad Core Xenon E5506 that runs at 2.13 GHz and holds 12 GB of RAM. The nodes communicate with a gigabit switch and the USRP2 with gigabit ethernet. Timing components were inserted into the waveforms to record timestamp and latency. A waveform was created in OSSIE's Eclipse-based waveform developer that was composed of a USRP_ Commander, timing component, two GC's in series, and a final timing component. This waveform was distributed across two nodes on CORNET with the first node consisting of the USRP_Commander, timing component, a GC, and the final timing component. The second node consisted of a GC only (Figure 1). Note that the USRP_ Commander is not connected to the first component because the USRP2 logical device (not shown) is connected to that component and also

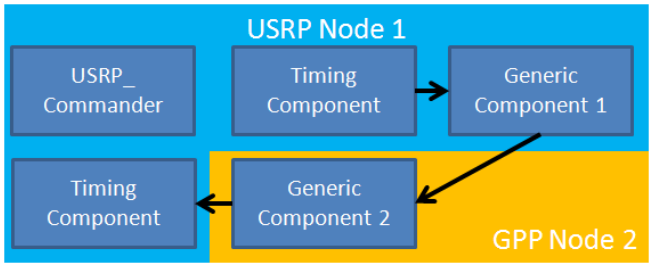

Figure 1, GC waveform

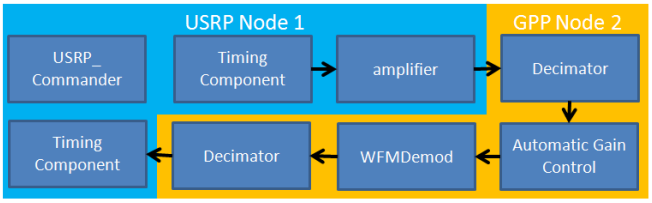

Figure 2, OSSIE waveform

connected to the physical USRP2. The USRP_Commander is connected to this logical device. This is the case for all of the following waveforms.

Another waveform was created in the same way for the purpose of comparison which consisted of a USRP_Commander, timing component, amplifier, decimator, automatic gain control, WFM demodulator, another decimator, and another timing component. The node was also distributed across two nodes with the first node containing the USRP_Commander, timing component, and amplifier. The second node contained the decimator, automatic gain control, WFMDemodulator, another decimator, and timing component (Figure 2).

These two waveforms both act as wide-band FM receivers, but how the signal is demodulated in both waveforms is different. This is to be expected since the first waveform uses the GR Digital Signal Processing (DSP) libraries and the second uses OSSIE components. The timing component mentioned in these waveforms is not part of the OSSIE 0.8.2 release and was created specifically for this experiment to measure latency. These components pass all data and output the time in UTC according to the system clock. For this reason, the two timing components had to be deployed on the same node. Although the overheads added by these timing components were not computed, they are consistent between the two waveforms and do not obfuscate the comparison. Two additional waveforms were created for comparison which were deployed in the same way as the GC waveform. The first of these two waveforms did not use the Cython implementation of the GC (Figure 3).

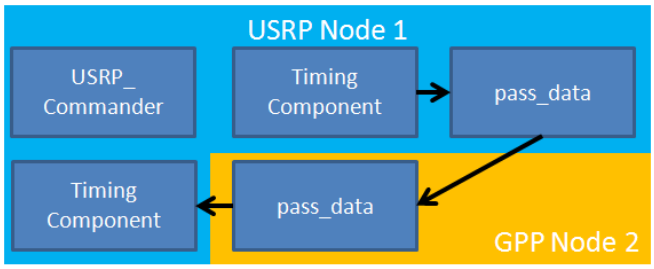

Figure 3, Pass_data waveform

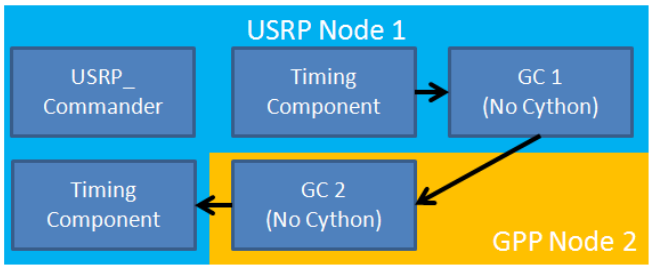

Figure 4. GC waveform with no Cython

The second of these two waveforms replaced the two GC's with python pass\textunderscore data components (Figure 4).

Each of these waveforms were installed and started on two nodes and latency measurements were taken from between the two timing components. The processing blocks between timing components encase all of the DSP necessary to demodulate an FM signal. Each of these waveforms received 100 packets of size 2048 and OSSIE data type complexShort.

3. RESULTS

The results are summarized in the following table:

Waveform	OSSIE Waveform	Pass Data Waveform	GC Waveform	GC (No Cython) Waveform
Average latency [ms]	1.495	.933	4.296	160.06

4. DISCUSSION

Comparing the C++ based OSSIE waveform and the Python based GC waveform reveals that indeed, there is a significant amount of overhead even though the GC can encapsulate many DSP blocks. It is also likely that this more pronounced than what can be seen here since the timing components have added some additional overhead. The difference between the GC waveform and the Python based pass data waveform shows that the GC is still about three times slower than the theoretical lower bound. That limit does not account for the processing done by the GR components, so the limit on the GC's throughput is likely to be higher. Since the entire GR infrastructure was invoked by the GC within an OSSIE component, and it was shown that GR can be slower than OSSIE under certain cases, this discrepancy in latency is not surprising. Comparing the Cython GC waveform with the with its non-Cython implementation shows the significant increase in throughput as a result of implementing the critical DSP path of the GC in Cython.

These results indicate possible performance improvements if the GC were re-implemented in C++ instead of Python. Taking this approach would likely increase the throughput significantly due to the removal of slow scripting code. If this were done, the GC would likely have latencies similar to C++ components and would likely offer a slightly more flexible alternative to many OSSIE waveforms. The GC would still suffer a slight overhead because the Python interpreter would still be necessary to run the core of the GC.

Fundamentally, the GC serves the purpose of allowing the user to pick from any ordered set of pre-compiled DSP blocks that is offered by GR. To make a truly generic component, it might be much flexible to have a library of DSP modules that are invoked in the program in the order the user wishes. In this way, a library could be created from any platform by extracting the signal processing code. If done correctly, this could be implemented in any C++ based platform, so the component would not just be processing generic, but platform generic too. This would require something completely different from the current GC implementation, but the overarching idea is fundamentally the same.

5. CONCLUSION

Encapsulation of GR flowgraphs within the GC affords increased flexibility and granularity of DSP. This increased flexibility comes at the price of increased latency. By implementing the processing core of the GC in Cython, the GC's throughput has increased dramatically. The GC in its current form will allow about a fourth of the same throughput that can be achieved by an OSSIE waveform alone. This may be a hindrance to some applications, but for others the capability for automated run-time compositional reconfiguration of a waveform under control of a cognitive engine will be worth the price of decreased throughput.

10. REFERENCES

[1] D. Chen, G. Vanhoy, M. Beaufait, and C. Dietrich, "OSSIE/GNU Radio Generic Component." New York, NY: Wireless Telecommunications Symposium (WTS 2011), April 2011

[2] E. Paone, "Open-source SCA Implementation-Embedded and Software

[3] Communication Architecture, OSSIE and SCA Waveform Development,"

[4] Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, February 2010

[5] G. Abgrall, F. L. Roy, J.-P. Delahaye, J.-P. Diguet, and G. Gogniat, "A comparative Study of Two Software Defined Radio Platforms." Washington, D.C.: SDR '08 Technical Conference and Product Exposition, October 2008

[6] P. Navarro, . R. Villing, and R. Farrell, "Software-Defined Radio Architectures Evaluation,." Washington, D.C.: SDR '08 Technical Conference and Product Exposition, October 2008