LATENCY PROFILING FOR SCA SOFTWARE RADIO

Thomas Tsou (Wireless@Virginia Tech, Blacksburg, VA, USA; ttsou@vt.edu); Philip Balister (Wireless@Virginia Tech, Blacksburg, VA, USA; balister@vt.edu); and Jeffrey Reed (Wireless@Virginia Tech, Blacksburg, VA, USA; jhreed@vt.edu)

ABSTRACT

Timing and latency are critical design parameters in the development of a communication system. With software radio and packet based methods emerging as viable replacements for traditional static solutions, different approaches are necessary to analyze the timing characteristics of these communication environments. This paper presents important concepts for performing timing and latency profiling within the OSSIE SCA software radio framework operating on the Linux operating system. Furthermore, relevant factors and parameters pertaining to latency performance are measured and compared. Specifically, the operations of inter-component communication within a collocated environment are explored.

1. INTRODUCTION

Time delays are an inevitable aspect of any real world communication system. As radio applications and underlying hardware becomes increasingly complex, however, these latencies become more and more difficult to predict and understand. Predictable latencies and deterministic behavior are necessary in order to meet the requirements of a wide range of today's communication needs. The advent of software defined radio and the use of the general purpose processor (GPP) as a suitable device for radio communications further complicates these issues. Traditionally, timing information and latency characteristics could be determined by examining hardware designs and specifications. With current software radio designs, however, operating system (OS) behavior, middleware, and multi-threaded environments are some of the issues that factor into latency behavior.

The Software Communication Architecture (SCA) [1] is a component based software specification developed for the Joint Tactical Radio System (JTRS) that seeks to address many design issues in developing interoperable software radios. In order to achieve interoperability and portability of applications, the specification defines a number of operating environment requirements for compliant implementations such as POSIX OS standards and the use of Common Object Request Broker Architecture (CORBA) as middleware. CORBA is a standard released by the industry consortium Object Management Group (OMG) and defines the communication between the components of a SCA radio waveform. The SCA and its underlying standards are specifications only and require appropriate implementations for actual use.

OSSIE (Open-Source Implementation::Embedded) [2] is an implementation of the SCA created at Virginia Tech for educational use as well as for research applications with software defined radio. OSSIE relies on a number of other open-source projects in order to address the standard. The implementation runs on Linux and utilizes omniORB [3] as the CORBA middleware implementation. Additionally, TinyXML [4] is used for parsing the XML used in SCA profiles. Also, OSSIE applications rely on a reusable interface library known as Standard Interfaces that simplifies the interaction of signal processing code portions with the implementation details of CORBA IDL.

This paper studies the factors that contribute to inter-component latency in an OSSIE waveform. It is assumed that components reside on the same processor. Distributed radio applications that span multiple nodes present additional factors and are not examined in this study. While a typical Intel GPP based PC is used as a test case in this study, an effort is made such that the general concepts presented are applicable to other platforms. Section 2 of this paper explores previous performed in the general area of software radio performance evaluation and related studies pertaining to some of the CORBA aspects of latency evaluation. Section 3 covers issues relating to performing time measurements in the Linux environment while section 4 introduces significant factors that contribute to the timing characteristics of component-to-component interactions. Timing measurements performed on a test system are shown in section 5.

2. RELATED WORK

Available literature directly concerning latency profiling within a component based SDR systems is currently limited. A number of studies exist pertaining to a more general case of software radio performance on general purpose

computing hardware. [5] and [6] measured overall end-to-end transmit and receive latencies including RF hardware with another software radio framework, GNU Radio [7] and the Universal Software Radio Peripheral (USRP). These studies have limited application to the approach presented in this paper where a finer level of timing granularity is used for exploring the time delays of data transfers occurring within a single processor space.

The SCA defines the use of CORBA for middleware and services as part of the operating environment. Consequently, CORBA performance plays a key role in characterizing the interactions between components. Performance analysis of CORBA implementations in the form of benchmarking has been studied more commonly in networked settings [8-10]. Operating system issues and communications within collocated environments were examined in [10] and [11]. The intended application for some of these benchmarking studies, however, is quite different than the use of CORBA within software radio. Specifically, many applications for SDR are comprised of components in collocated environments where clients invoke numerous requests transferring a large amount of data samples on single long-lived connections. This contrasts with many other CORBA scenarios such as a distributed stock market application where a high number of short requests may be coming from a large number of network connection.

3. MEASURING TIME

Data transfer latencies between collocated software radio components can occur on the order of microseconds. This differs from network latencies often examined in distributed CORBA environments that may have duration times of milliseconds. In order to measure and examine these latencies, accurate clocks and precise timing procedures are required. The methodology used for the measurement of timing information should be able to provide mechanisms for accurate means of measurement while maintaining applicability to multiple platforms when possible.

3.1 Processor Clocks

Many CPU's maintain internal timers that operate at the processor clock rate. With today's processors running at frequencies in excess of 1 GHz, these internal timers are capable of highly accurate measurements. The Time Stamp Counter (TSC) is one such timer that exists on Intel x86 processors beginning with the original Pentium. The TSC is a monotonically increasing time source that can be accessed through the assembly operation RDTSC (Read Time Stamp Counter). Timing information can be obtained in the form of a processor clock cycle count and converted to human units of time also known *wall time* by using the processor clock speed.

Changes in processor designs, however, have made the TSC problematic in a number of cases. Power management strategies that alter the processor clock frequency can skew times calculated from cycle counts [12]. Furthermore, the advent of dual-core and multi-core computing has made TSC based timing measurements particularly difficult [13]. Multi-core machines maintain separate TSC for each particular core and while each counter is guaranteed to be monotonically increasing, clock drift among cores is a significant issue. Consequently, successive RDTSC calls are not guaranteed to produce monotonic values creating the possibility of backward moving time. TSC drift and synchronization issues are non-trivial to solve and are currently active areas of development with the Linux kernel. As a result, kernel based time functions rather than direct assembly calls provide a better approach to obtaining timing information in many of these cases. In addition, a second detriment to hardware specific timer calls is that it leads to fractured code across different architectures.

3.2 Kernel Timing

Traditionally, the timekeeping structure provided by the Linux kernel relied on a tick-based approach with limited granularity [14]. A *tick* is a periodic interrupt driven by a hardware timer that forms the basis for a large portion of kernel functionality. The tick frequency is dependent on the specific kernel and platform, however, 1000Hz is a typical value. To account for the limited granularity, the kernel used interpolation methods for timekeeping which allowed for increased clock resolution, however, inconsistencies and susceptibility to irregular behavior limited usability for profiling purposes. Recent changes kernel, however, address some of these timing shortcomings.

Hrtimers (high resolution timers) [15] tackle a number of issues concerning the Linux time subsystems. Specifically, timekeeping is addressed in a new `timeofday` [16] approach that eliminates the tick and interpolation based dependency. Furthermore, large portions of architecture specific code are replaced with a modular design that provides an interface allowing the best available clocksource to be used. Issues related to the TSC, or other clocksource specific handling, are addressed within the kernel As a result, higher resolution timekeeping for latency determination can be obtained through the Linux kernel without dealing with potentially problematic architecture specific code. Furthermore, by using kernel timing interfaces, latency testing code can to be used across different architectures. Responsibility still remains with the developer, however, to make sure that clocksources and drivers of sufficient reliability are available to the kernel.

hrtimers were merged into the mainline Linux kernel in version 2.6.16.

4. CONTRIBUTING FACTORS

While it is not feasible to characterize all possible causes of inter-component latency in an OSSIE waveform, a number of significant factors can be brought forth.

4.1 Scheduling

The scheduler is the element of the operating system responsible for determining which process is to execute next on the CPU. The Linux scheduler [17] supports three thread policies, a normal policy, First In-First Out, and Round Robin. SCHED_OTHER is the standard Linux time-sharing mode that applies to processes with no real-time requirements. SCHED_FIFO and SCHED_RR are real-time policies provided for time-critical applications where a runnable high priority processes can immediately preempt currently running lower priority processes. SCHED_FIFO processes run without time slicing while SCHED_RR adds time slicing to the SCHED_FIFO scheme.

The scheduler is important in that OSSIE waveforms share resources with other running processes on the operating system that have large effects on component-to-component delays. Implementing OSSIE components with RT priorities is done through Linux system calls. The required system calls, however, are platform specific and not supported natively by omniORB. As a result, omniORB servant threads are spawned with normal non-RT priorities even if the component is scheduled with higher RT priority. This is addressed using omniORB Interceptors [3] where platform specific code can be inserted and executed at run time after servant thread creation.

4.2 OSSIE Standard Interfaces

OSSIE Standard Interfaces is a class library created for the purpose of simplifying the interaction of signal processing code with SCA required CORBA interfaces. The library facilitates code reuse for many common data types and eases the learning curve of presenting standardized CORBA and IDL to students with backgrounds predominantly in communications and signal processing. Furthermore, Standard Interfaces provides support for multiple outputs in fan-out connection configuration. Currently supported IDL interfaces include basic real and complex data representations in 8, 16, and 32 bit sizes passed in the form of CORBA sequences.

The implementation of Standard Interfaces separates the bulk of CORBA handling away from the signal processing code. This is performed by buffering data in the servant code and subsequently returning the CORBA call

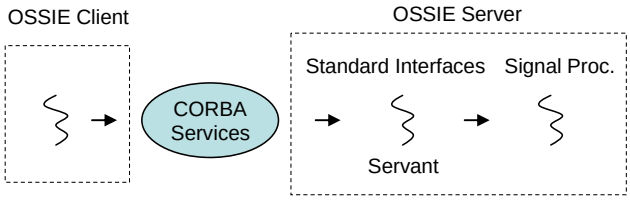

Figure 1 OSSIE Threading Model

without entering the signal processing code space. In a separate signal processing thread, the data is passed in by reference using non-CORBA C++ method calls built into Standard Interfaces. This effectively decouples the CORBA call sequence from signal processing code. The thread structure is shown in Figure 1. Thread safety is maintained through the use of semaphores. With regards to latency, Standard Interfaces minimally adds a data copy and thread switch to the overall latency of a CORBA invocation. Additional port connections will also incur an additional delay.

4.3 CORBA Transport

omniORB is a compliant Object Request Broker (ORB) implementation of the 2.6 version of the Common Object Request Broker Architecture (CORBA) specification. With CORBA, communications between ORBS is specified by an abstract protocol called the General Inter-ORB Protocol (GIOP). The GIOP protocol, unusable in its abstract from, is made concrete by mapping to a specific transport. For example, the Internet Inter-Orb Protocol (IIOP) is a concrete mapping of TCP/IP to GIOP. omniORB supports IIOP as well as other transports through Unix domain sockets and SSL [18]. With current use of the OSSIE framework, there is limited applicable use of the SSL transport, however, UNIX domain sockets can provide

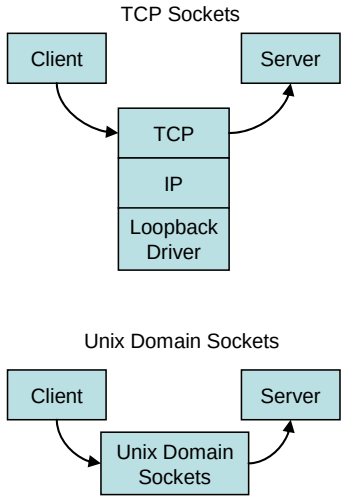

Figure 2 TCP and UNIX Domain Sockets

significant benefits in terms of data transport efficiency.

A primary detriment to the use of IIOP in a collocated environment is the inefficiency stemming from the fact that CORBA messages are passed through the TCP/IP protocol stack to and from the loopback driver. As an alternative when multiple components reside on the same Unix system, the Unix domain protocols provide a form of inter-process communication (IPC) that is similar in use to internet sockets. The primary benefit over TCP sockets, however, is simplicity in the underlying implementation. A comparison of the data flow is shown in Figure 2. As a result, Unix domain sockets require less processing which results in increased transfer speeds and an effective reduction in latency and jitter.

4.4 CORBA Threading

omniORB provides two modes of operation for server side request handling [3]. In one thread per connection mode [20], a new dedicated thread is created for each connection. It should be noted that the initial call invocation takes a significantly longer than subsequent calls due to the cost of thread creation. In thread pool mode, all incoming connections are watched by a single thread which assigns calls to a thread chosen from a pool. This mode is more efficient for a large number of connections when the number of threads becomes high; however, thread pooling adds an additional latency component due to thread switching for each call. Thread per connection is the most efficient form of operation when the number of connections is low. Thus, it is the preferable configuration for the majority of software radio applications

4.5 Packet Size

The invocation path of a remote call between two components in an OSSIE waveform (and CORBA based systems in general) can be broken down into a number of factors. In this paper, call latency is the minimum cost of sending any message at all regardless of packet size. Marshaling latency consists of the time spent by the ORB turning structured data into a buffered byte stream and vice-versa. Transport latency refers to the time taken to move a number of raw bytes through some transport mechanism between two process spaces. Sometimes transport latency is included in marshalling latency [20]. Also note that call latency and transport latency are partially related.

How each of these values factor into the overall latency is heavily dependent on the amount of data sent. When sending zero length sequences, call latency dominates which is dependent on the transit latency and internal ORB processing. Call latency becomes less of a factor as packet lengths become long. Marshalling latency and transmit latency are directly related to the amount of data sent.

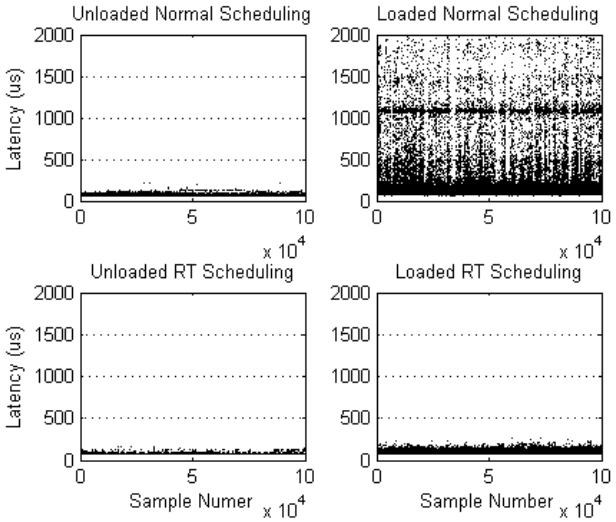

Figure 3 RT Scheduling Performance

5. SELECTED RESULTS

The following results examine latencies that occur in a connection between two components of an OSSIE software radio waveform. The test platform was a basic PC with a Pentium 4 CPU clocked at 3.2 GHz with 2 GBytes of RAM. The test system uses the 2.6.22.1 kernel with rt9 version of the real time patch [21]. The latency from the invocation of the client CORBA call to the time the data is made available in the signal processing thread is measured with using the `gettimeofday` function. From Standard Interfaces, a `complexShort` interface is used with a sequence length of 512, which equates to 2048 bytes of sample data per call. The sequence length was chosen to be representative of a typical value used in a number of sample waveforms. Tests were run for 500,000 time samples. While this example is simple and only partially representative of larger signal processing waveforms, it allows a close examination of the interactions and associated latencies underlying data transfer between OSSIE software radio components.

5.1 RT Scheduling

The RT scheduling test examines the latency performance of a real-time process under processor load. In this case the latency with and without the SCHED_FIFO parameter is examined with a multi-threaded kernel build. Scatter plots are shown in Figure. It can clearly be seen that under load the non-RT case exhibits extremely large maximum latencies. This behavior is expected as CPU resources are shared equally with the background load. With RT scheduling, the measured values increase slightly but remain bounded compared to the non-RT case. RT scheduling is used for the remainder of the results.

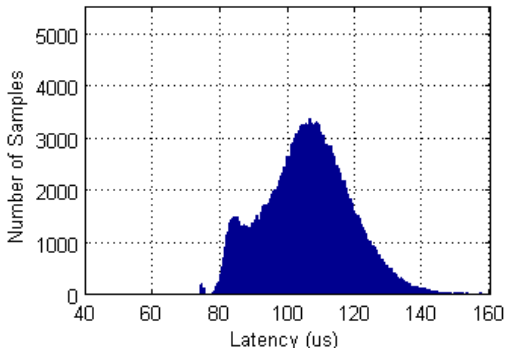

Figure 4 OSSIE component latency using IIOP/TCP CORBA transport

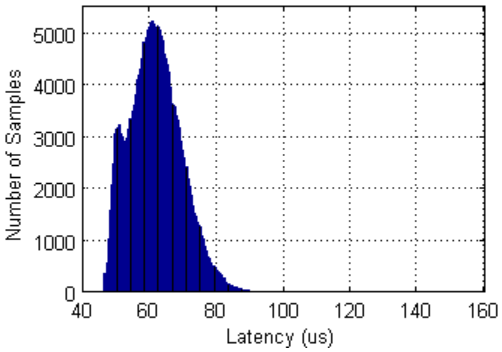

Figure 5 OSSIE component latency using Unix socket transport

5.2 Comparing CORBA Transports

The use of Unix domain sockets as transport layer for omniORB was compared to IIOP using TCP. In this case, average latency and jitter were measured and compared. As expected, the use of Unix domain sockets provided a significant performance improvement with a mean latency of 62us compared 107us for TCP. Furthermore, the standard deviation decreased from 14us to 8us. The histograms for measured values are shown in Figures 4 and 5.

These results indicate that a significant portion of CORBA latency and jitter can be attributed to the underlying transport layer. This is expected in network situations where highly variable packet transmit times exist. In this OSSIE case, however, the use of TCP still adds a noticeable amount of unpredictability even though packets never leave the PC.

5.3 Packet Size

Figure 6 shows measured mean latencies for sequence lengths from 0 to 4096. The equivalent transmit sizes not including CORBA sequence metadata and GIOP header and encoding overhead are 0 to 16384 Bytes. Thus, on this system, the minimal possible latencies between two components using the `complexShort` interface are roughly 50us and 100us using Unix sockets and TCP transport respectively. With Unix sockets, latency increases in a linear fashion as sequence length grows. With TCP, the results are more irregular and most likely related to underlying TCP and IP packet size handling.

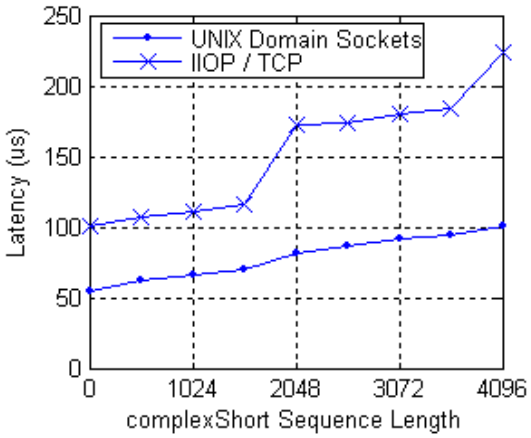

Figure 6 Latency vs complexShort sequence length. A complexShort type consists of two 16-bit values.

6. CONCLUSION

This paper describes relevant concepts necessary for timing and latency analysis of an OSSIE SCA waveform operating on the Linux operating system. Requirements and means for accurate profiling of collocated components were explored. In addition, a number of key factors contributing to inter-component latency were identified. In efforts to improve existing latencies, scheduling and underlying CORBA transports were examined and measured on a test machine with varying packet sizes. While these measurements are specific to the given system, the concepts and approach can also be applied to other platforms.

7. ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under Grant No. 0520418. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The authors also acknowledge support from the National Institute of Justice and Wireless@VT partners.

8. REFERENCES

[1] "Software Communications Architecture Specification," Joint Tactical Radio System (JTRS) Joint Program Office, Version 2.2.2, May 2006.

[2] Wireless@Virginia Tech, "OSSIE," http://ossie.wireless.vt.edu/trac/.
[3] D. Grisby, S. Lo, D. Riddoch, *The omniORB version 4.1 User's Guide*, AT&T Laboratories Cambridge, 2007.
[4] L. Thomason, "TinyXML", http://www.grinninglizard.com/tinyxml/.
[5] T. Schmid, O. Sekkat, M.B. Srivastava, "An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios," in *Proc. of the Second ACM Iinternational Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2007, pp. 59-66.
[6] S. Valentin, H. Malm, H. Karl, "Evaluating the GNU Software Radio Platform for Wireless Testbeds," Technical Report TR-RI-06-273, Feb. 2006.
[7] "GNURadio," http://www.gnu.org/software/gnuradio/.
[8] P. Tůma, A. Buble, "Open CORBA Benchmarking," *Proc. of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001)*, 2001.
[9] J. Zou, D. Levy, "Evaluating Overhead and Predictability of a Real-time CORBA System," *Proc. of the 37th Hawaii International Conf. on System Sciences*, 2004.
[10] H. Callison, D. Butler, "Real-time CORBA Trade Study," Boeing Phantom Works Advanced Information Systems, Doc. Num. D204-31159-1, Jan. 2000.
[11] D. Schmidt, M. Deshpande, C. O'Ryan, "Operating System Performance in Support of Real-time Middleware," *Proc. of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, 2002, pp. 199-206.
[12] P. Work, K. Nguyen, "Measure Code Sections Using the Enhanced Timer," Intel Corporation, Tech. Rep.
[13] R. Brunner, "TSC and Power Management Events on AMD Processors," AMD, Tech. Rep., posted to comp.unix.solaris, Nov. 2, 2005.
[14] R. Love, *Linux Kernel Development Second Edition*, Novell Press, 2005.
[15] T. Gleixner, D. Niehaus, "Hrtimers and Beyond:Transforming the Linux Time Subsystems," in *Proc. of the Linux Symposium*, 2006, pp. 333-346.
[16] J. Stultz, N. Aravamudan, D. Hard, "We Are Not Getting Any Younger: A New Approach to Time and Timers," *Proc. of the Linux Symposium*, 2007, pp. 219-232.
[17] "SETSCHEDULER(2)" Linux Man Pages.
[18] T. Nakajima, "Supporting Multiple Transport Protocols in a CORBA System," *Proc. of the International Conf. on Network Protocols*, 2000, pp. 220-229.
[19] W.R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*," Addison-Wesley, 1996.
[20] M. Henning, S. Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.
[21] S. Rostedt, D.V. Hart, "Internals of the RT Patch," in *Proc. of the Linux Symposium*, 2007, pp. 162-172.