SOFTWARE DEFINED RADIO EXECUTION LATENCY

Feng Ge, Alex Young, Terry Brisebois, Qinqin Chen, and Charles W. Bostian
Virginia Polytechnic Institute and State University, Wireless @ Virginia Tech, Center for Wireless Communications, Virginia Tech, Blacksburg, VA 24061, USA;
{gef, alex.young, tbrisebo, chenq, bostian }@vt.edu

ABSTRACT

Software Defined Radio (SDR) achieves multi-band multi-mode reconfigurability by moving digital signal processing functions progressively closer to the radio antenna and utilizing software's flexibility. This requires an easy development environment, which is very challenging for digital signal processors (DSP) or field-programmable gate arrays (FPGA). General purpose processors (GPP) are therefore widely used in SDR architectures like GNU Radio and OSSIE.

Nevertheless, GPPs create several problems for SDR development because they are designed for running several general purpose tasks simultaneously – they are not specifically designed for radio or DSP functions. In this paper, we analyze the execution latency in SDR using GNU Radio as an example. We explore most latency sources by following the entire receiver chain from the RF front end to MAC functions in software domain and also examine the GPP architecture characteristics. We give numerical latency measurements taken by using timestamps.

In addition, we compare the execution time difference between GNU Radio and a WiFi card, and show the impact of SDR execution latency for wireless network applications, especially at the MAC layer. We further analyze where this difference comes from and explore possible methods of overcoming it by using parallel architectures.

1. INTRODUCTION

Aided by advances in silicon technology, radio frequency (RF) technology, and software methods, SDR [1] engineers have moved digital signal processing functions progressively closer to the radio antenna. SDRs can now replace inefficient analog circuitry with reliable, low-priced digital circuits. Furthermore, the flexibility of programmable signal processors augments an SDR's ability to accommodate various waveform formats and protocols on one radio platform, and allows the SDR to configure itself "on the fly"; today, SDRs can achieve true multi-band multimode reconfigurability [2]. SDR technology has been widely embraced as the way to develop waveform agile radio platforms and also offer functionalities for cognitive radio [3] and dynamic spectrum access (DSA) developments [4].

Continuously increasing in computing ability following Moore's Law, current general purpose processors (GPPs) are fast enough to do a lot of real time digital signal processing tasks and functions. With many library functions and a very easy development environment, several widely used SDR architectures like GNU Radio [5] and OSSIE [6] are developed on GPPs. However, this makes SDR not just a radio or DSP problem anymore. It is also a computer problem; GPP's architecture [7] and operating system (OS) mechanism [8] should be considered.

In this paper, we use GNU Radio, a typical example of an SDR, to analyze a fundamental limitation in SDR architecture: the execution latency [9]. This latency significantly limits SDR's applications, like supporting networking functions in the PHY/MAC layer [9], realizing DSA and cognitive applications, and sustaining network throughput. In this paper, we follow the entire receiver chain from the RF front end to MAC functions in software domain and analyze all sources of latency and their overall impact on higher layers. By isolating latency introduced from different sources, we study the latency arising from both the SDR architecture and the supporting software environment and hardware components, including the RF front end, data bus, GNU Radio's architecture, GPPs' OS [8], and the memory hierarchy [7]. With these methods, we also explore a few possible ways to improve both hardware and software architectures for SDR development. In our studies using GNU Radio, we believe we have discovered some general limitations of SDR, due either to its own architecture or to supporting platforms.

Wireless standards, e.g., IEEE 802.11 a/b/g, have very strict timing requirements. To satisfy them, current commercial products use ASICs together with DSPs mostly for baseband signal processing, as well as for modulation, demodulation, packet processing, etc. By using IEEE 802.11 as an example of wireless standards, we compare the execution latency of GNU Radio to that of a commercial WiFi chip. This allows us to study the limitations of SDR and explore new methods, especially when running on a GPP, for wireless network applications.

2. GNU RADIO ARCHITECTURE

GNU Radio is an open architecture for building software defined radios [5]. It was started in the early 2000s by Eric

Blossom and others, and has evolved into a mature software infrastructure used and supported by a large community of developers. The Universal Software Radio Peripheral (USRP) is an openly designed low-price SDR hardware platform which implements radio front-end functionality and A/D and D/A conversion currently using the Universal Serial Bus (USB2.0) to connect to the host PC.

2.1. USRP and GNU Radio

The current, first-generation USRP consists of a motherboard with four high speed 12-bit 64 Msps analog to digital converters (ADC), four high speed 14-bit 128 Msps digital to analog converters (DAC), an Altera FPGA and a programmable Cypress FX2 USB 2.0 controller. The ADCs, DACs and the FPGA together provide support for IF processing. The FPGA on the board provides four digital up converters (DUC) and four digital down converters (DDC) to shift frequencies from baseband to the required operating frequency. The FPGA can be reprogrammed to provide additional functionality such as pulse shaping. RF front ends are attached in the form of daughter cards which can currently cover all the radio bands from 0 Hz to 2.4 GHz.

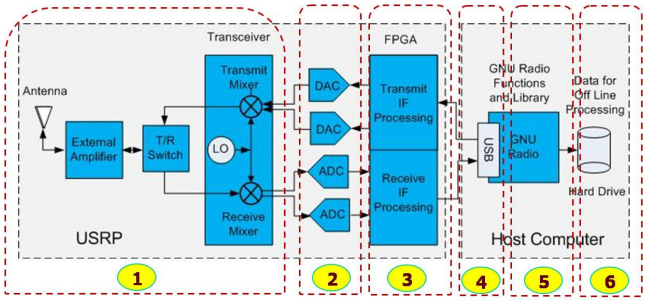

Figure 1 - A basic SDR system based on GNU Radio and USRP [10]

GNU Radio was originally designed to run on GPPs. Combined with minimal analog radio hardware it allows software radio development of waveforms, modulations, protocols, signal processing, and other communications functions in the digital domain. The GNU Radio signal processing library includes existing and developing blocks for most signal-processing functions, such as waveform modulation and filtering. The USRP is fully supported by the GNU Radio library and a combined system of both is shown in Figure 1.

2.2. Latency Sources in GNU Radio

If we follow the receiver chain in GNU Radio, there are six factors that may introduce latency, as shown in Figure 1: (1) analog signal processing and wire delay in USRP's analog circuits; (2) sampling delay in converters and programmable gain amplifier (PGA); (3) filter processing time in FPGA; (4) USB's data queue transmission mechanism, and its limited buffer size and transmission speed; (5) GNU Radio's streaming architecture in both radio control and PHY information, signal processing, and GNU Radio scheduler, which will be detailed in later sections; (6) GPP operating system (OS) latency and uncertainty, GPP memory hierarchy, and others.

Let's use Δ to represent latency in different resources; we can decompose the latency into four parts:

$$\Delta = \Delta_{USRP} + \Delta_{USB} + \Delta_{GNURadio} + \Delta_{GPP} \quad (1)$$

where Δ_{USRF}, Δ_{USB}, $\Delta_{GNURadio}$, and Δ_{GPP} are the total delay time introduced by USRP, USB, GNU Radio software architecture, and the host GPP. Individually, Δ_{USRP}, $\Delta_{GNURadio}$, and Δ_{GPP} can be further decomposed as shown below:

$$\Delta_{USRP} = \Delta_{Ana\log} + \Delta_{Wire\_delay} + \Delta_{AD/DA} + \Delta_{FPGA} + others \quad (2)$$

$$\Delta_{GNURadio} = \Delta_{signal\_processing} + \Delta_{Scheduler} \quad (3)$$

$$\Delta_{GPP} = \Delta_{OS} + \Delta_{memory\_hierarchy} + others \quad (4)$$

In the following sections, we will briefly analyze each of the four parts separately.

2.2.1. USRP

As shown in Figure 2, the USRP introduces latency because of analog signal processing and wire delay, TX/RX switch, tuning in the voltage-controlled oscillator (VCO), signal sampling process, programming delay in the programmable gain amplifier (PGA), and filter processing in FPGA. TX/RX switch and VCO do not impact steady state execution latency, and most other parts' latency is negligible with the typical latency of 1 μs in the FPGA.

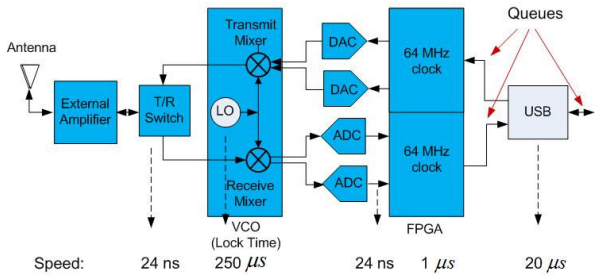

Figure 2 – Radio component time scale.

2.2.2. USB Characteristics

The current USRP connects to the host computer through USB 2.0 ports with buffers on both sides: 8 KB on the USRP side and 32 KB on the host computer side. All the data is put into queues before it goes through the USB

connection, as shown in Figure 2. The USB connection transmits data in blocks with a minimum size of 512 B. This means that smaller data chunk will have to wait for incoming data to fill up to 512B before it can pass the USB. Therefore, the USB latency, which is the time duration for moving data from FPGA to the USB driver on the PC, is decided by three factors: packet size, USB data rate, and buffer size on both sides of the USB connection. The USB data transmitting latency is

$$\Delta_{USB} = \frac{f(512, fusb\_nblocks * fusb\_block\_size)}{sample\_size * f_s} \quad (5)$$

where $f(x, y)$ depends on the amount of data in the buffer and is at least x and at most y, $fusb\_nblocks$ is the number of data blocks in USB, $fusb\_block\_size$ is the size of each block, f_s is the sampling frequency, and $sample\_size$ is usually 32 bits for complex signal samples.

Because of the queue structure in USB, there is latency variation among different data rates in the signal flow. For burst signals flowing from the RF front end when both buffers are empty, the latency is decided by equation (5), which also determines latency for low rate continuous signal flow. For high rate continuous signal flow, new arriving signal data have to wait for previous data flowing out of the buffer. Therefore, the latency accounts for the time that buffers empty previous data. For the TX chain, it is quite similar except that the PC side memory buffer size is larger.

2.2.3. GNU Radio Running Mechanism

Currently, programming in the GNU Radio platform uses a combination of C++ and Python, a simple, high-level programming language. The computationally intensive processing blocks are implemented in C++ while the control and coordination of these blocks for applications that sit on top are developed in Python.

When executing SDR functions, the GNU Radio scheduler processes data as a stream of homogeneous items in any active flow graph. It breaks each data stream into chunks and feeds these chunks one at a time into each block in the flow graph using the mechanism described in Algorithm 1 [11].

Algorithm 1

```
while flow_graph.start do
    for i = 0, 1, ..., # blocks do
        if block i has enough input data and
sufficient room in output port
            process block i;
        end if
    end for
end while
```

The scheduler scans through all included signal processing blocks in the flow graph from the head to the end, and then loops back. The scheduler is essentially a cyclic poller, calling each block in turn to perform its processing function, always cycling in the same order. This mechanism creates extra latency in looping. Even if one block has the highest priority for a particular task, it must still wait for following and preceding signal processing blocks.

2.2.4. GPP Memory Hierarchy and OS Environment

In a GPP operating system environment, GNU Radio executes all the PHY layer functions including IF band and baseband functions. GPPs' memory hierarchy and OS's characteristics will bring overhead that is not a concern in FPGAs and ASICs.

Specifically, modern GPP OS is designed to maximize resource utilization – to assure that all available CPU time, memory, and I/O are used efficiently, and that no individual user takes more than her fair share [8]. To maximize resource utilization, the OS uses processes and threads to run multiple jobs simultaneously at all times: applications, system programs, drivers, etc. The CPU scheduler manages all threads and processes according to CPU-scheduling algorithms like First-Come, First-Served (FCFS) scheduling, or Round-Robin (RR) scheduling. All scheduling algorithms balance several criteria like CPU utilization, throughput, waiting time, response time. The implication for GNU Radio is that the OS will create latency uncertainty for GNU Radio implementation because GNU Radio signal flow graph processing is just one process running in the operating system even though GNU Radio can set a high priority for this single process.

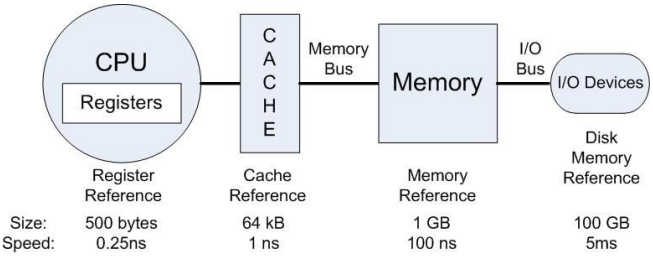

Figure 3 – Memory Hierarchy in GPPs and the Speed Difference [7].

Furthermore, GPPs have a memory hierarchy as shown in Figure 3. GPPs run any program instructions (with data) in the CPU with registers while instructions and data are usually stored in the disk or memory. There is a vast speed difference between CPU and memory [7]. To solve this problem, several levels of caches are inserted between the CPU and memory, and speculative methods are used to pre-fetch instructions or data into the cache. However, any failure in speculative pre-fetching will cause the CPU to

wait for data read from memory with a long delay, as shown in Figure 3.

3. LATENCY MEASUREMENTS

In Section 2, we analyzed the possible latency, while, in this section, we will give numerical measurements. To measure the latency, we use two USRPs connected to one computer so that we can utilize timestamps within one computer to measure the time between transmitting a packet and receiving it. As shown in Figure 4, we implement a digital transmitter and receiver – each in its own USRP - and we insert four timestamps: the first (Timer Head) is right before the data packetizing, the second (Timer 1) is right before the transmitter IF band, the third (Timer 2) is right after the receiver's IF band, and the fourth (Timer Tail) is right after data de-packetizing. Our experiment uses an Intel Core 2 Duo processor (@2.40 GHz) with 2G of memory and the GNU Radio code is based on version 3.1.

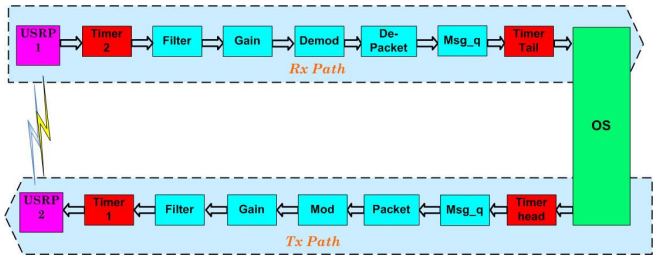

Figure 4 – Experiment Setup for Measuring SDR Execution Latency

Next, we transmit different sizes of packets and measure the total latency, i.e., the time difference between Timer Head and Timer Tail. For BPSK at different bit rates, the latency time vs. packet size is shown in Figure 5.

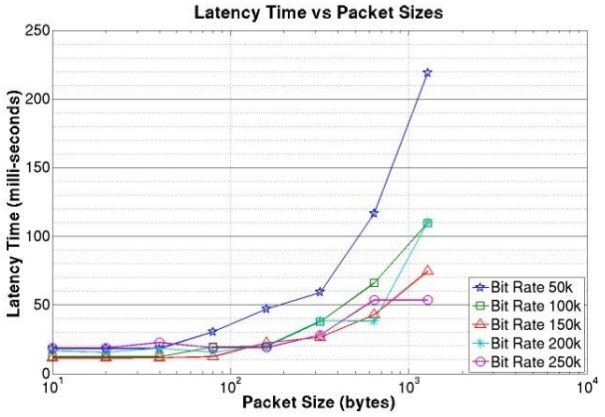

Figure 5 – The Latency Time between Transmitting a Packet and Receiving it Using BPSK with Different Packet Sizes.

As we can see in Figure 5, the latency time is on the order of ten milliseconds between sending and receiving even a very small packet (10 bytes) at all test bit rates. As the packet size increases beyond 100 bytes, this latency time increases continuously and reaches the order of 100 milliseconds for 1000 bytes, though a higher bit rate results in a shorter latency.

Following Algorithm 1, a big packet in GNU Radio is broken into a sequence of data chunks and each one is processed following the flow graph shown in Figure 4. Considering the uncertainty from the three queues shown in Figure 2 and the limited size of USB buffers, we further measure the amount of time spent on transmitting and receiving a packet at different modulations and bit rates. Particularly, we let the transmitting packet go directly to the receiving side at the baseband and USRPs are not used, therefore removing the impact of waiting for emptying USB queues and other hardware latency. Such time latency is purely decided by the GNU Radio architecture and the CPU speed, and theoretically it is proportional to the number of samples (per second) for the same baseband functions setting if there is enough available CPU and memory.

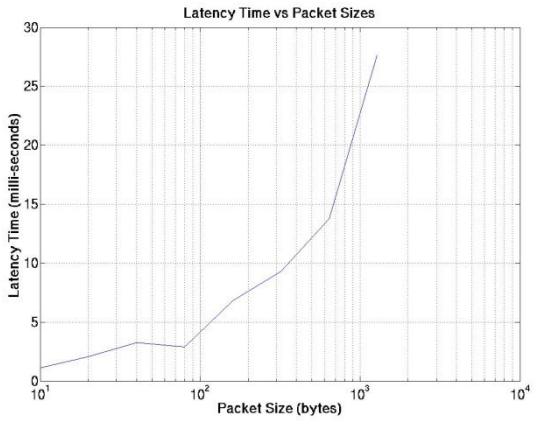

Figure 6 – BSPK Baseband Signal-processing Time

As an example in Figure 6, we use BPSK in transmitting one packet and receiving it at the baseband. The behavior of time latency vs. packet size is similar to Figure 5, though it is one order of magnitude smaller. Such latency is usually at the scale of tens of micro-seconds for small packets.

4. THE GAP BETWEEN SDR AND A WIFI CHIP

IEEE 802.11 consists of a set of very successful protocols and their MAC functions are crucial to this success. There are several functions that require precise and very fast timing performance like TDMA (sync), CSMA (DIFS, SIFS), carrier sense, dependent packets (ACKs, RTS), fine-grained radio control (frequency hopping), etc., as shown in

Table 1. To achieve the above timing requirement, ASICs are usually used in commercial products.

Table 1. Summary of important timing constants in 802.11b, 802.11a, and 802.11g [12].

Parameter	Value			
	802.11b	802.11 a	802.11g only	802.11 g + legacy
SLOT	20 µs	9 µs	9 µs	20 µs
SIFS	10 µs	16 µs	10 µs	10 µs
DIFS	50 µs	34 µs	28 µs	50 µs
PHY Header	192 µs [long] 96 µs [short]	20 µs	20 µs	20 µs

4.1. A WiFi Chip Architecture

A typical Wi-Fi card [13], as shown in Figure 7, uses a markedly different architecture from GNU Radio. The RF front-end and signal processing are on separate chips. The RF front-end chip on this diagram serves the same function as the daughterboard connected to the USRP. Filtering, switching, frequency conversion and amplification are all performed on this chip. An analog, baseband signal is transmitted between this chip and the DSP chip which also does analog-to-digital and digital-to-analog conversion. Furthermore, there are separate components on the DSP chip for filtering, modulation and demodulation, MAC functions, and encryption and decryption. This architecture allows many functions to be performed simultaneously. For example, for a continuous and constant stream of data, the physical layer operations, baseband process, MAC layer operations, and encryption can be performed at the same time. GNU Radio processes them in a sequential way in which each processing block must wait for a discrete chunk of data to be processed by the previous block. The speed difference, as we can see from Figure 5 and Table 1, is that GNU Radio and USRP is approximately three orders of magnitude slower than a WiFi chip for sending and receiving a small packet. Not counting USB impact, even the baseband signal processing within GNU Radio software domain alone is slower than a WiFi chip for small packets of tens of bytes.

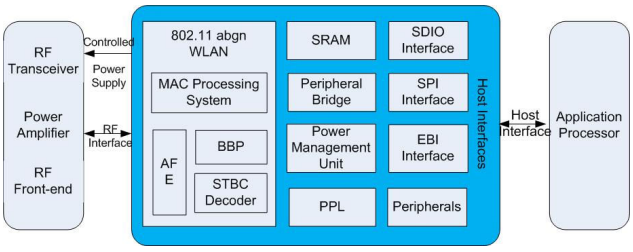

Figure 7 – A WiFi Chip Architecture Example.

4.2. Another Look at SDR Latency

Fundamentally, GPPs can only run one task at a time and each program is executed sequentially, even though some instruction and data level parallelisms like instruction pipeline, super-scalar instruction execution, and SIMD, are widely implemented [7]. The CPU scheduler switches CPU and memory resource among multiple running tasks very quickly [8]. Even worse, GNU Radio scheduler adopts a purely sequential way to execute the signal flow from IF band to the MAC layer functions. Analog radio components, ASICs, FPGA, and commercial wireless devices all execute signals in pipeline (a continuous sequence of signal chunks are executed in parallel on different components while there is some overlap for two subsequent chunks on one functional component), as exemplified by the WiFi chip in Section 4.1. The speed difference in executing same amount of computation between pipeline and sequential can be significant, as illustrated by a hypothetical example in Figure 8.

Although we use GNU Radio as just one example, it does demonstrate most common characteristics of SDR on GPPs. Even though other GPP based SDRs like OSSIE and IRIS [14] use different configuration architectures, they still have the same latency problem resulting from GPP's OS and memory hierarchy as well as sequential execution of SDR code just like GNU Radio.

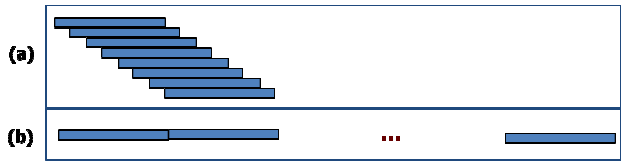

Figure 8 – An hypothetical illustration of speed difference: pipeline (a) is 8 times faster than sequential (b).

Message-block (m-block) is introduced in GNU Radio to use meta-data and control channel on the USRP's FPGA, therefore cutting the latency [15]. However, this method still uses GPPs to sequentially execute the majority of the PHY/MAC layer functions. Moving such functions more into the FPGA requires expertise in hardware and low-level languages and it will complicate the development environment.

Multi-core or many-core architecture can use parallelism in achieving the speed requirement while still maintaining the flexibility provided by a GPP development environment. For example, the Cell Broadband Engine (Cell BE) has nine heterogeneous cores [16], nVidia GPU has 256 cores [17], and Intel has an 80-core CPU [18].

In addition, the I/O interface speed plays a very important role in SDR for timing sensitive applications, where USB is certainly not suitable and better I/O interfaces

such as PCI-X are needed. Ideally, the computing component together with the RF front end and ADC/DAC should be put in one system (system-on-package) if not one chip (system-on-a-chip).

5. CONCLUSION

In this paper, we used GNU Radio (and USRP) as an example to analyze most latency sources following the entire radio signal receiver chain from the RF front end to the PHY/MAC layer. These sources include the USRP, the USB, the GNU Radio execution mechanism, and the GPP OS and memory hierarchy. We also gave numerical measurements on the latencies within the software domain and the overall path from the transmitting packet to the receiving packet.

We compared GNU Radio's execution latency against IEEE 802.11 MAC protocol's timing requirement. We also explored and analyzed the fundamental difference in signal processing, i.e., sequential vs. pipeline, between GPP based SDRs and conventional radios based on analog circuits, ASICs, or FPGA. To maintain a simple development environment while achieving fast speed, we propose to use multi-core or many-core architectures in SDR development.

6. ACKNOWLEGEMENT

This work was supported by DARPA under grant W31P4Q-07-C-0210, by the National Institute of Justice, Office of Justice Programs, U.S. Department of Justice under Award No. 2005-IJ-CX-K017, and by the National Science Foundation under Grant No. CNS-0519959. The opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of these sponsors or the official policy or position of the DARPA, Department of Defense, or the U.S. Government.

7. REFERENCES

[1] J. Mitola, *Software Radio Architecture: Object Oriented Approaches to Wireless Systems Engineering*: John Wiley and Sons, 2000.

[2] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering,* . Englewood Cliffs, NJ: Prentice-Hall, 2004.

[3] J. Mitola, "Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio," Royal Institute of Technology (KTH), 2000.

[4] M. McHenry, E. Livsics, T. Nguyen, and N. Majumdar, "XG dynamic spectrum access field test results," *IEEE Communications Magazine,* vol. 45, pp. 51-57, 2007.

[5] E. Blossom, "Exploring GNU Radio," *http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html,* November 2004.

[6] http://ossie.wireless.vt.edu/.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture : a Quantitative Approach*, 3rd ed.: San Francisco, CA : Morgan Kaufmann Publishers, 2003.

[8] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*: Wiley; 7 edition, 2004.

[9] T. Schmid, O. Sekkat, and M. B. Srivastava, "An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios," in *WiNTECH: The Second ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and CHaracterization,* Montreal, QC, Canada, 2007.

[10] F. Ge, Q. Chen, Y. Wang, T. W. Rondeau, B. Le, and C. W. Bostian, "Cognitive Radio: From Spectrum Sharing to Adaptive Learning and Reconfiguration," in *2008 IEEE Aerospace Conference,* Big Sky Montana, MT, 2008.

[11] "ADROIT: GNU Radio Architectural Changes," http://acert.ir.bbn.com/downloads/adroit/gnuradioarchitectural-enhancements-3.pdf, Ed., May 2007.

[12] K. Medepalli, P. Gopalakrishnan, D. Famolari, and T. Kodama, "Voice capacity of IEEE 802.11b, 802.11a and 802.11g wireless LANs," *IEEE Global Telecommunications Conference,* vol. 3, pp. 1549-1553, 2004.

[13] http://www.redpinesignals.com/.

[14] P. Mackenzie, "Software and reconfigurability for software radio systems." vol. Ph.D dissertation: Trinity College Dublin, Ireland, 2004.

[15] G. Nychis, T. Hottelier, Z. Yang, P. Steenkiste, and S. Seshan, "Enabling MAC Protocol Implementations on Software Defined Radios," in *US-Ireland International Workshop on Next Generation Open Architectures for Software Defined Radio (CISDR),* Maynooth, Ireland, 2008.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development,* vol. 49, pp. 589-604, 2005.

[17] http://www.nvidia.com.

[18] http://www.intel.com.

Copyright Transfer Agreement:

The authors represent that the work is original and they are the author or authors of the work, except for material quoted and referenced as text passages. Authors acknowledge that they are willing to transfer the copyright of the abstract and the completed paper to the SDR Forum for purposes of publication in the SDR Forum Conference Proceedings, on associated CD ROMS, on SDR Forum Web pages, and compilations and derivative works related to this conference, should the paper be accepted for the conference. Authors are permitted to reproduce their work, and to reuse material in whole or in part from their work; for derivative works, however, such authors may not grant third party requests for reprints or republishing.