SOFTWARE ARCHITECTURE FOR COOPERATIVE APPLICATIONS

Thomas Tsou (Wireless@Virginia Tech, Blacksburg, VA, USA; ttsou@vt.edu)
Jeffrey H. Reed (Wireless@Virginia Tech, Blacksburg, VA, USA; reedjh@vt.edu)

ABSTRACT

With the initial generation of reconfigurable software radios now being deployed, an increasing number of flexible and agile communication devices are being introduced into the field. Existing SDR research has focused primarily on single node applications in terms of reconfigurability and performance. Possibilities exist, however, for distributed applications that leverage the increasing flexibility of SDR in a cooperative manner. This paper examines using general purpose commodity hardware as an enabling factor for initial development of new cooperative waveforms.

Recent research developments in coordinated multi-node communication are introduced as well as key implementation challenges. Furthermore, a proposed development architecture using an open-source SDR platform for supporting cooperative waveforms is described. Specifically, recent real-time extensions of the Linux operating system are explored for providing necessary timing requirements at the user level on general purpose hardware. Initial timing measurements are provided along with discussion of future development directions.

I. INTRODUCTION

Over the past decade the path of software defined radio development has undergone numerous changes. Hardware platforms and software models have evolved significantly with continuing breakthroughs in faster, more efficient processors and reconfigurable software architectures among many other notable advancements. Concurrently as technology progresses and research communities generate new ideas and applications, novel possibilities emerge that can be benefited by and integrated with software defined radio.

Recently a rapid area of growth in the research world that can be assisted by SDR technology is the topic of distributed and cooperative applications. Generally speaking these research areas encompass diverse topics such as sensor networks, signal processing, and network applications. More specifically, we consider a case where multiple relay nodes coordinate transmissions in the time domain in order to improve communications performance and reliability. The architecture requirements are characterized by timing requirements to support coordinated actions at the physical and MAC layers. From a wireless network perspective, our long-term goal is to develop architectures and examine software concepts that can support an initial class of applications.

From a hardware standpoint, we assume a general purpose processor (GPP) software radio model. After the processor, perhaps the most critical component of the operating environment is the operating system kernel. The kernel controls the shared processing resources available to the applications with significant effects on latency, stability, and throughput. Consequently, close examination of a specific operating system, Linux, as a critical component is a primary focus of this paper.

The remaining paper is organized as follows. Section 2 discusses the current application context and associated requirements and constraints that must be supported on the individual node level. Section 3 describes the overall software radio architecture with a specific focus on operating system effects. Section 4 and 5 describe a basic test environment and initial results. Finally, from these initial results, subsequent effects on software architecture decisions are explored in Section 6 with concluding statements in Section 7.

II. APPLICATION AND REQUIREMENTS

Wireless network relaying and directly related topics such as distributed MIMO and virtual arrays has recently emerged as a significant topic of research in a diverse range of communication areas that include information theory, sensor networks, and signal processing. Due largely to an established and still rapidly growing body of published work, researchers are beginning to direct efforts toward development and implementation issues of cooperative networks. Given the high cost of dedicated experimental and test solutions, software radio can play a role in lowering the barriers to entry for exploring these emerging communication concepts.

As an example of a basic test scenario, a relay channel with two relaying nodes is shown in Figure 1. S, R_n, and D refer to the source, n relay node, and destination respectively. We assume a two stage transmission protocol where the source transmits in the first stage, while the relay nodes cooperatively tramsmit in the second stage after receiving the initial transmission. α_n is the complex valued fading channel coefficient for path n at the mth transmission stage while w refers to additive white gaussian noise. Notably, τ_n is the effective delay at the receiver on each path, and the critial aspect to SDR that this paper addresses.

A wide range of coding and transmission schemes are possible of varying complexity [1], and theoretical results have been widely explored [2]. However, from a software architecture and implementation perspective our current motivations are somewhat simpler. The critical effect of the software radio architecture and implementation is altering τ due to significant internal processing overhead. τ would otherwise only be subject to propagation delays in a theoretical physical model. For any implementation - software or not - perfect alignment of symbols is not an absolute requirement [3]; however, a certain degree of predictable timing behavior is

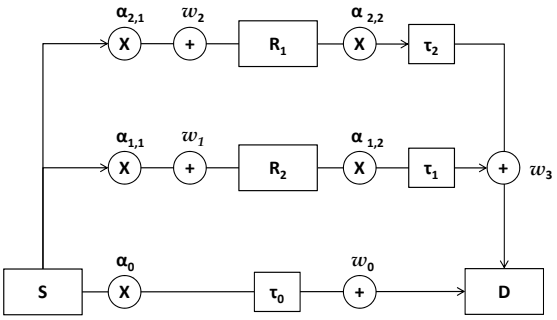

Fig. 1. Basic relay model

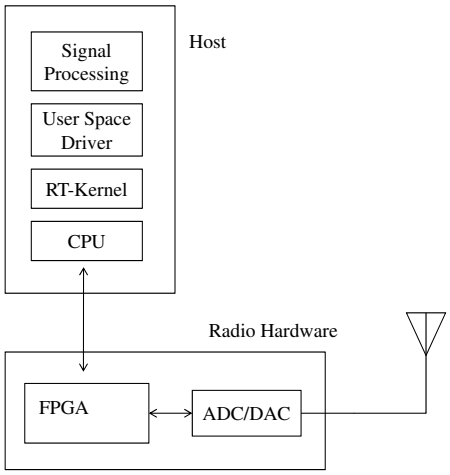

Fig. 2. Generic SDR architecture

necessary between relay nodes, which, at this time, is not readily acheivable on existing general purpose software radio architectures. Specific requirements vary upon application, however, we consider a predictable and bounded additive latency in the low hundreds of μs range suitable for our research and testing purposes.

III. SOFTWARE DEFINED RADIO MODELS

Previous work in software radio use in cooperative communication systems is limited, however, a number of studies have been performed in related areas. Previous studies have examined latency and scheduling of software radio systems in various forms with GNU Radio [4] being a typical starting point. Notably for experimental MAC layer applications, studies have been published in [5] and [6]. Also, the use of software radio for RFID readers was shown in [7]. For cellular applications, the OpenBTS project targets a software radio implementation of a GSM basestation [8].

A generic software radio architecture is shown in Figure 2 and is similar to structures mentioned by various software radio texts [9], [10]. Use of this general model in cooperative

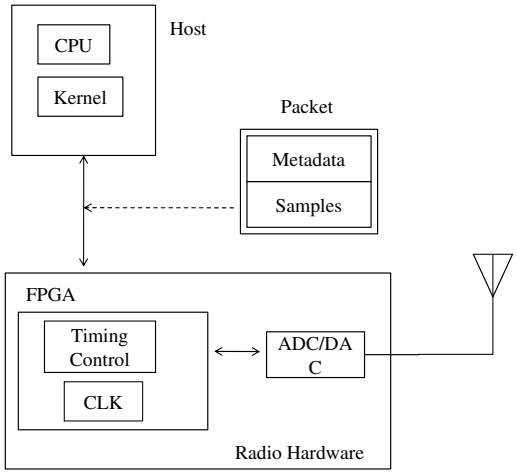

Fig. 3. Split or hybrid SDR architecture

waveforms - and also in MAC, RFID, and cellular applications - has been difficult due to lengthy and unpredictable latencies attributed to complexities and overhead of the software radio processing path. This processing path contrasts with fixed systems where functionalities are tightly integrated and deterministic. As a result, recent software projects with strict timing considerations have introduced hybrid models that take into account transport and processing delays of various sections of the software radio. An example of such an approach is the in-band signalling project [5]; a simplified version of the architecture is shown in Figure 3.

The consequence of these split models is that usability and flexibility are sacrificed to an extent in that programming non-general purpose processors require specialized skills and toolsets. For our case, rather than discounting the feasibility of using the model of Figure 2, we choose to examine specific aspects of the software radio chain and attempt to stabilize latencies where they occur. Delays occur at multiple transport and processing points in a software radio. As an example, we describe the signal path of a popular setup, the Universal Software Radio Peripheral [11] and a Linux based host PC.

The USRP is a open-source, low-cost digital conversion board designed to allow general purpose computers to capture a wide spectrum band through an USB interface. It was developed for the GNU Radio Project and can be coupled with several RF daughter boards that allow access to different spectrum bands. Delays may occur within the USRP itself, the USB bus, and within the PC host after samples are transferred to main memory. We primarily focus on the third point within the GPP main memory as it relates to operating system use and is the most variable timing factor. A forth delay point is from the signal processing and used algorithms, however, this area is not addressed by operating system or software architecture changes.

Critical to the software interface design is that SDR signal processing takes place in user space. Thus, direct access

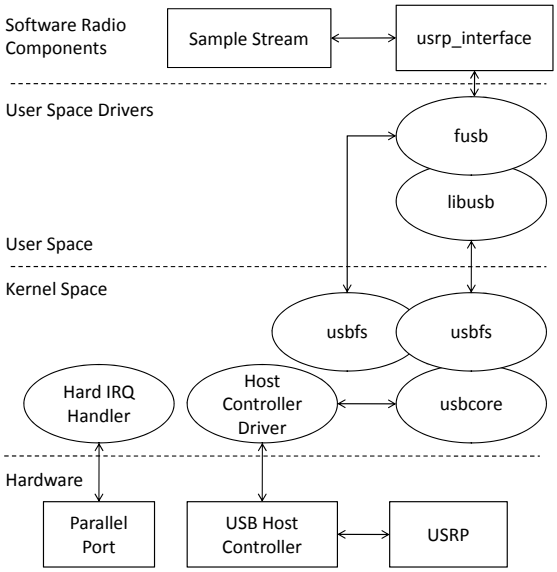

Fig. 4. USRP driver stack

```
latency: 70 us, #268/268, CPU#0 | (M:desktop VP:0,
-----------------
|task: swapper-0 (uid:0 nice:0 policy:0 rt_prio:0)
-----------------

  _------=> CPU#
 / _-----=> irqs-off
| / _----=> need-resched
|| / _---=> hardirq/softirq
||| / _--=> preempt-depth
|||| /
|||||    delay
|||||  time  |   caller
|||||   \   |   /
0dNh.   0us : usb_hcd_irq (handle_IRQ_event)
0dNh.   1us : uhci_irq (usb_hcd_irq)
0dNh.   2us : _spin_lock (uhci_irq)
0dNh.   2us+: uhci_scan_schedule (uhci_irq)
0dNh.   4us : uhci_free_td (uhci_scan_schedule)
0dNh.   4us : dma_pool_free (uhci_free_td)
0dNh.   4us : _spin_lock_irqsave (dma_pool_free)
0dNh.   5us : _spin_unlock_irqrestore
  ...       ...          ...
0dNh.  69us : uhci_urbp_wants_fsbr
0dNh.  69us : _spin_unlock_irqrestore
0dNh.  70us : usb_free_urb (usb_hcd_giveback_urb)
0dNh.  70us : _spin_lock (uhci_giveback_urb)
0dNh.  70us : uhci_urbp_wants_fsbr
0dNh.  71us : usb_hcd_irq (handle_IRQ_event)
0dNh.  71us : trace_hardirqs_on (handle_IRQ_event)
```

Fig. 5. Hard interrupt latency trace

to hardware is restricted to interfaces provided through the operating system API calls. In our example, received samples enter main memory through the USB bus and subsystem. Subsequently, this data needs to be made available to the the user space driver and applications by the operating system. The code paths of these internal operations are shown in Figure 4.

Given the complexity of operations to move data from the RF interface to user space where primary signal processing occurs, it is evident that the operating system plays an integral role in timing sensitive software radio waveforms. The kernel is the central component of the operating system and interacts directly with system hardware as well as providing interfaces for user space applications. Linux is based on a monolithic kernel - similar to most general purpose operating systems - in that low-level access to hardware is limited to the operating system and user access is provided through a set of primitives or system calls. The scope of the monolithic kernel contrasts with microkernels - sometimes used in hard real-time operating systems - where services such as device drivers and protocol stacks are contained in user space and are more readily accessable. Consequently, in our case, the Linux kernel directly impacts upper layer applications that require timing sensitive interactions with the hardware.

Until recently, however, Linux did not have direct support for real-time operation. With development coming from the PREEMPT_RT community [12], Linux is becoming a more suitable operating system for time sensitive applications. Recently, many significant features have been merged into the mainline kernel, which now supports mechanisms such as a preemptable kernel and priority scheduling. Without full pre-emption capabilities conflicts over shared resources may lead to unpredictable delays. For example, due to device driver designs, certain interrupt handlers operate in hard context with interrupts disabled. Examination into our development system using ftrace [13], reveals that during a testing interval the longest time period with interrupts off is roughly 70 s. A truncated trace is shown Figure 5 and illustrates the complex operations that may occur in a non-preemptable interrupt context. In the next section, we describe a test environment so these effects can be measured in a method more specfic to a software radio waveform applications.

IV. TESTING

In order to determine the suitability of the Linux kernel as the foundational software layer for our distributed software radio tasks, we created tests to measure interrupt latency and jitter from within the kernel as well as from user space applications, where software radio signal processing functionality resides. The test approach is representative of a receive data transfer from another device on the hardware platform, the ADC or front-end FPGA for example, using direct memory access (DMA), which occurs independently of the central processor operation. When incoming data is ready for processing, an interrupt is triggered by the transferring device through a dedicated interrupt on the processor. The processor subsequently performs a mode switch and begins executing an interrupt handler which either directly processes the data, or more likely prepares the data for processing by a userspace application. The goal of our tests is to measure the latency and stability of the interrupt signaling process in

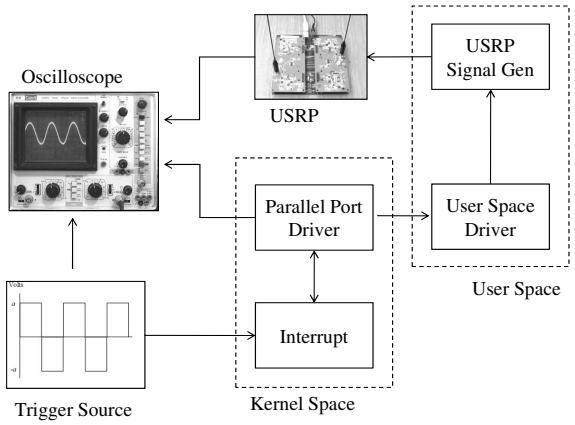

Fig. 6. Software delay measurement setup

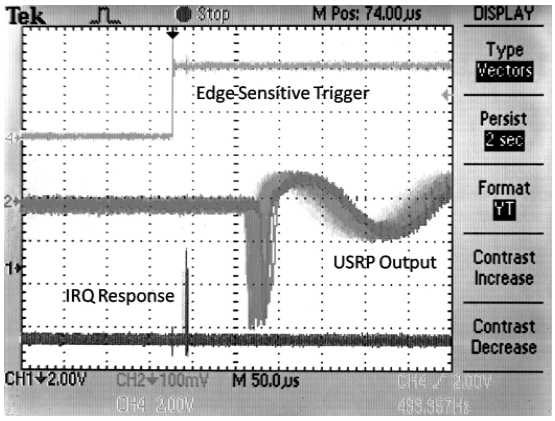

Fig. 7. Oscilloscope output

controlled settings representative of software radio operation. This approach allows us to identify specific causes of system latency that would be otherwise difficult to trace using an end-to-end testing with a complete waveform.

To measure interrupt latency we created a simple device driver for the parallel port on a basic PC (Intel Pentium 4, 3.20GHz). To initiate the interrupt, we fed in a square wave which triggered the parallel port and Intel Advanced Programmable Interrupt Controller (APIC) [14] with the leading edge on an unshared IRQ at an interrupt frequency of 10 kHz. Upon interrupt, the handler immediate returned a pulse on an output pin of the parallel port from interrupt context. Both the input wave and output pulse were output on an oscilloscope where the timing values were examined and logged. Operating from a hard interrupt context is the shortest and most direct method for simple request-and-acknowledgement signaling, but it is not representative of the more general of software radio waveform operation since no transfer of samples is involved.

As another test, we wrote a similar user space driver triggered from the parallel port. In this test setup, the interrupt handler finished immediately upon triggering and signaled that a return signal be generated from the user space application. The user space application was run at a real-time priority using the preemptive scheduler, SCHED_FIFO. In addition, a number of other procedures were taken to minimize possibilities of non-determinacy. These procedures included locking memory to avoid page faults due to virtual memory use [15] and pre-faulting the stack to reduce the possibility of a stack fault [16]. Furthermore, for a comparison we included an application configuration running with normal user space scheduling and no other preventive methods.

During the tests an idle processor was used to determine minimal latencies and jitter values. To simulate a large background load, we the canonical kernel-hacking workload [15] of a parallel kernel build. This latter test provides insight into how well the system can maintain stability under load. With respect to a software defined radio environment, the test characterizes how well the processor and operating system can maintain high priority MAC and physical layer timing constraints while dealing with a lower priority background load. The results are shown in Table 1 and Table 2 which show the externally measured round trip times under idle and loaded conditions respectively. We defined jitter as the width of the time interval for 99.9% of samples. Maximum latencies, or statistical outliers, were recorded over an interval of 60 seconds.

V. RESULTS

Our initial test results displayed in Table V and Table V show that the process of triggering the processor based on an external event and handling the interrupt is a stable operation when no other tasks are contending for the CPU. Do to the idle load and no contention on the specific interrupt used for the parallel port, which likely removed any detectable variance due to the APIC, or, in general, anything in the signaling path to the CPU, the interrupt latency is stable and predictable within the time frames that we measured. The Intel APIC is known for interrupt latency jitter though this behavior was not detected in the idle testing [14].

	Median	Jitter	Max
IRQ Context	6	¡1	9
User Mode	11	4	100
RT_PREEMPT	10	1	20

TABLE I
LOW UTILIZATION (μs)

	Median	Jitter	Max
IRQ Context	8	6	53
User Mode	30	1ms	1ms
RT_PREEMPT	12	13	200

TABLE II
UNDER APPLIED LOAD (μs)

Real-time priorities are not applicable to interrupt handlers since they run in interrupt context. More specifically the basic driver that we used ran in hard context with interrupts disabled so thread priorities were not applicable. On the other hand, our signaling interrupts can be delayed by other interrupts for the same reason, which we will describe next.

Moving the handler to a user process raised the maximum delay by a large amount with smaller changes in average latency and jitter. The small changes in average latency can be attributed to the change from interrupt to process context and additional associated function calls. The significant change in maximum latency is due to the fact that user space threads preemptable by interrupts and in the case of non-RT scheduling, any other running process. Given the structure of the Linux kernel used, compared to industrial real-time operating systems, these results fall within expectations with the following conclusions. Most significantly, system load is the primary factor. The actual processing time of responding to an interrupt is an extremely short and largely deterministic operation even from user space and delays occur when other processor, or other interrupts, prevent or preempt the handler from running during periods of processor contention.

VI. OUTLOOK

Given our initial results, the Linux kernel offers the potential to support timing sensitive cooperative applications under certain general conditions. First, the timing requirements of the application can not exceed fundmental limitations of the hardware platform and operating system. Second, a stable operating environment must be maintained. The first condition, which can only be determined through thorough testing, is intuitive and not specific to Linux. The second condition, however, is due to the fact the Linux with preemptive and real-time capabilities enabled is still a general purpose operating system. As such, Linux does not achieve the the level of process separation and allow dedicated processor utilization as done in hard real-time operating systems.

As previously mentioned, the stability of the operating environment is dependent on a number factors with system load being the primary concern. More specifically this limitation is a result of contention over shared resources by multiple processes. With the basic kernel configuration that we used, operating under load with real-time priorities or even in interrupt context was not sufficient to guarantee an upper bound on execution time for our basic signaling task even if application requirements are relaxed by dropping worst-case instances and statistical outliers. Consequently, our current kernel configuration may not be sufficient for the cooperative signal processing and computing tasks when relative processing loads are high. One possible solution in this case is to maintain adequate headroom and open availability of resources though this approach comes at the detriment of efficiency goals.

If we assume that the primary means of maintaining reliable timing characteristics on a Linux based SDR is by maintaining low relative system loads, counter to the concerns of timing are a number of factors. Energy utilization and processor efficiency are two of these factors which are critical for any processor working with multiple functionalities. Ideally to maximize efficiency, a processor and software stack would allow operation at maximum utilization while maintaining timing capabilities required for communication. Currently, dedicated hard real-time operating systems can achieve these objectives to a much greater extent than the main Linux kernel.

The traditional approach in embedded systems is to move computationally heavy or critical applications into dedicated hardware routines on a DSP, FPGA, or simply a separate dedicated GPP [17]. This approach reveals the compromises of real-time performance requirements against design flexibility and solutions that are easy to modify and customize. Underpinning these approaches is the goal of giving high priority, time-critical tasks full utilization of the process in an uninterruptable manner, which leads to predictable operation. This is easily achievable on a processor with a single task, which is rarely the case in modern multitasking software systems where conflicts over shared processor resources readily occur.

Nevertheless, technology changes quickly so these considerations may change as the community progresses and the Linux kernel develops. Only recently has the Linux kernel been considered suitable for general real-time tasks without the use of major patches or extensions. Furthermore, current multi-core processors and platforms may be operable in configurations that could separate processes physically such that real-time response is achieved in a different manner. As a result, significant potential remains to be examined using Linux based systems in cooperative software radio systems.

VII. CONCLUSIONS

This paper considers the role of SDR in examining distributed cooperative applications. A basic GPP based hardware platform was described that included the use of the Linux operating system. The relevance of accurate timing capability was discussed with subsequent discussion about the use of the Linux kernel for a general outlined application. A test environment was described and latency measurements were provided along with alternatives and tradeoffs. Additional testing with the Linux kernel remains as well as further development of proposed application implementations.

ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research under Grant No N30001407010536.

REFERENCES

[1] B. Sirkeci-Mergen and A. Scaglione. Randomized space-time coding for distributed cooperative communication. *Signal Processing, IEEE Transactions on*, 55(10):5003–5017, October 2007.

[2] J. N. Laneman and G. W. Wornell. Distributed space-time-coded protocols for exploiting cooperative diversity in wireless networks. *Information Theory, IEEE Transactions on*, 49(10), 2003.

[3] Shuangqing Wei. DiversityMultiplexing Tradeoff of Asynchronous Cooperative Diversity in Wireless Networks. *Information Theory, IEEE Transactions on*, 53(11):4150–4172, November 2007.

[4] GNU Radio Website. Available at: *http://www.gnuradio.org*.
[5] Zhuocheng Yang Srinivasan Seshan Peter Steenkiste George Nychis, Thibaud Hottelier. Enabling mac protocol inplementations on software-defined radios. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, April 2009.
[6] Mani Srivastava Thomas Schmid, Oussama Sekkat. An experimental study of network performance impact of increased latency in software defined radios. In *WiNTECH '07*, September 2007.
[7]
[8] The OpenBTS Project. Available at: *http://openbts.sourceforge.org*.
[9] J.H. Reed. *Software Radio: A Modern Approach to Radio Engineering*. Prentice Hall, 2002.
[10] B.A. Fette. *Cognitive Radio Techonolgy*. Newnes, 2006.
[11] Ettus Research Website. Available at: *http://www.ettus.com*.
[12] Real-Time Linux Wiki. Available at: *http://rt.wiki.kernel.org*.
[13] S. Rostedt. Ftrace, April 2009.
[14] *Intel 64 and IA-32 Architectures Software Developers's Manual Volume 3A: System Programming Guide Part I, No. 253668-031US*, June 2009.
[15] P.E. McKenney. 'real time' vs. 'real fast'. In *Proceedings of the Linux Symposium*, July 2008.
[16] Dominic Duval. From fast to predictably fast. In *Proceedings of the Linux Symposium*, July 2009.
[17] I. Lee. *Handbook of Real-Time and Embedded Systems*. Chapman and Hall, 2008.