DOI:10.1145/1866739.1866760

Sora: High-Performance Software Radio Using General-Purpose Multi-Core Processors

By Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M. Voelker

Abstract

This paper presents Sora, a fully programmable software radio platform on commodity PC architectures. Sora combines the performance and fidelity of hardware software-defined radio (SDR) platforms with the programmability and flexibility of general-purpose processor (GPP) SDR platforms. Sora uses both hardware and software techniques to address the challenges of using PC architectures for high-speed SDR. The Sora hardware components consist of a radio front-end for reception and transmission, and a radio control board for high-throughput, low-latency data transfer between radio and host memories. Sora makes extensive use of features of contemporary processor architectures to accelerate wireless protocol processing and satisfy protocol timing requirements, including using dedicated CPU cores, large low-latency caches to store lookup tables, and SIMD processor extensions for highly efficient physical layer processing on GPPs. Using the Sora platform, we have developed a few demonstration wireless systems, including SoftWiFi, an 802.11a/b/g implementation that seamlessly interoperates with commercial 802.11 NICs at all modulation rates, and SoftLTE, a 3GPP LTE uplink PHY implementation that supports up to 43.8Mbps data rate.

1. INTRODUCTION

Software-defined radio (SDR) holds the promise of fully programmable wireless communication systems, effectively supplanting current technologies which have the lowest communication layers implemented primarily in fixed, custom hardware circuits. Realizing the promise of SDR in practice, however, has presented developers with a dilemma.

Many current SDR platforms are based on either programmable hardware such as field programmable gate arrays (FPGAs)[8, 10] or embedded digital signal processors (DSPs).[6, 12] Such hardware platforms can meet the processing and timing requirements of modern high-speed wireless protocols, but programming FPGAs and specialized DSPs are difficult tasks. Developers have to learn how to program to each particular embedded architecture, often without the support of a rich development environment of programming and debugging tools. Such hardware platforms can also be expensive.

In contrast, SDR platforms based on general-purpose processor (GPP) architectures, such as commodity PCs, have the opposite set of trade-offs. Developers program to a familiar architecture and environment using sophisticated tools, and radio front-end boards for interfacing with a PC are relatively inexpensive. However, since PC hardware and software have not been designed for wireless signal processing, existing GPP-based SDR platforms can achieve only limited performance.[1, 7] For example, the popular USRP/GNU Radio platform is reported to achieve only 100kbps throughput on an 8-MHz channel,[18] whereas modern high-speed wireless protocols like 802.11 support multiple Mbps data rates on a much wider 20-MHz channel. These constraints prevent developers from using such platforms to achieve the full fidelity of state-of-the-art wireless protocols while using standard operating systems and applications in a real environment.

In this paper we present Sora, a fully programmable software radio platform that provides the benefits of both SDR approaches, thereby resolving the SDR platform dilemma for developers. With Sora, developers can implement and experiment with high-speed wireless protocol stacks, e.g., IEEE 802.11a/b/g and 3GPP LTE, using commodity general-purpose PCs. Developers program in familiar programming environments with powerful tools on standard operating systems. Software radios implemented on Sora appear like any other network device, and users can run unmodified applications on their software radios with the same performance as commodity hardware wireless devices.

An implementation of high-speed wireless protocols on general-purpose PC architectures must overcome a number of challenges that stem from existing hardware interfaces and software architectures. First, transferring high-fidelity digital waveform samples into PC memory for processing requires very high bus throughput. For example, existing 802.11a/b/g requires 1.2Gbps system throughput to transfer digital signals for a single 20-MHz channel, while the latest 802.11n standard needs near 10Gbps as it uses even wider band and multiple-input–multiple-output (MIMO) technology. Second, physical layer (PHY) signal processing requires high computation for generating information bits from the large amount of digital samples, and vice versa, particularly at high modulation rates; indeed, back-of-the-envelope calculations for processing requirements on GPPs have instead

The original version of this paper was published in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. This work was performed when Ji Fang and He Liu were visiting students and Geoffrey M. Voelker was a visiting researcher at Microsoft Research Asia.

motivated specialized hardware approaches in the past.[14, 16] Lastly, wireless PHY and media access control (MAC) protocols have low-latency real-time deadlines that must be met for correct operation. For example, the 802.11 MAC protocol requires precise timing control and ACK response latency on the order of tens of microseconds. Existing software architectures on the PC cannot consistently meet this timing requirement.

Sora addresses these challenges with novel hardware and software designs. First, we have developed a new, inexpensive radio control board (RCB) with a radio front-end for transmission and reception. The RCB bridges an RF front-end with PC memory over the high-speed and low-latency PCIe bus. With this bus standard, the RCB can support 16.7Gbps (×8 mode) throughput with sub-microsecond latency, which together satisfies the throughput and timing requirements of modern wireless protocols while performing all digital signal processing on host CPU and memory.

Second, to meet PHY processing requirements, Sora makes full use of various features of widely adopted multi-core architectures in existing GPPs. The Sora software architecture explicitly supports streamlined processing that enables components of the signal processing pipeline to efficiently span multiple cores. Further, we change the conventional implementation of PHY components to extensively take advantage of lookup tables (LUTs), trading off computation for memory. These LUTs substantially reduce the computational requirements of PHY processing, while at the same time taking advantage of the large, low-latency caches on modern GPPs. Finally, Sora uses the Single Instruction Multiple Data (SIMD) extensions in existing processors to further accelerate PHY processing.

Lastly, to meet the real-time requirements of high-speed wireless protocols, Sora provides a new kernel service, *core dedication*, which allocates processor cores exclusively for real-time SDR tasks. We demonstrate that it is a simple yet crucial abstraction that guarantees the computational resources and precise timing control necessary for SDR on a multi-core GPP.

We have developed a few demonstration wireless systems based on the Sora platform, including: (1) SoftWiFi, an 802.11a/b/g implementation that supports a full suite of modulation rates (up to 54Mbps) and seamlessly interoperates with commercial 802.11 NICs, and (2) SoftLTE, a 3GPP LTE uplink PHY implementation that supports up to 43.8Mbps data rate.

The rest of the paper is organized as follows. Section 2 provides background on wireless communication systems. We then present the Sora architecture in Section 3, and we discuss our approach for addressing the challenges of building an SDR platform on a GPP system in Section 4. We then describe the implementation of the Sora platform in Section 5. Section 6 provides a quantitative evaluation of the radio systems based on Sora. Finally, Section 7 describes related work and Section 8 concludes.

2. BACKGROUND AND REQUIREMENTS

In this section, we briefly review the PHY and MAC components of typical wireless communication systems. Although different wireless technologies may have subtle differences among one another, they generally follow similar designs and share many common algorithms. In this section, we use the IEEE 802.11a/b/g standards to exemplify characteristics of wireless PHY and MAC components as well as the challenges of implementing them in software.

2.1. Wireless PHY

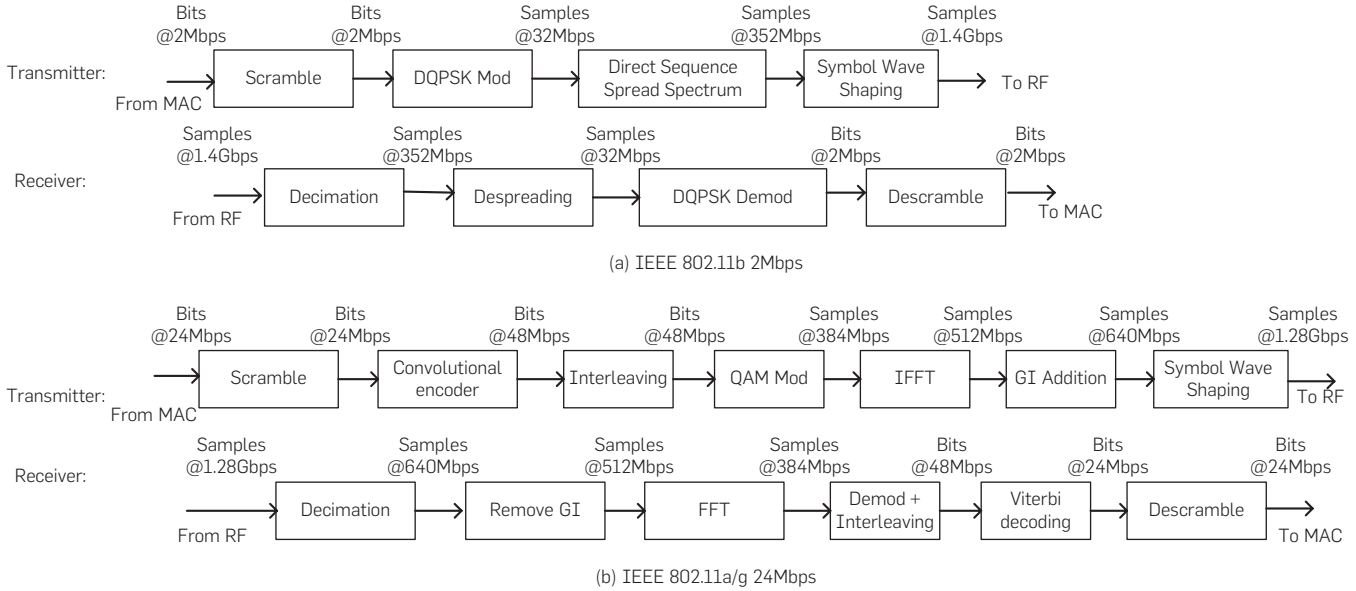
The role of the PHY layer is to convert information bits into a radio waveform, or vice versa. At the transmitter side, the wireless PHY component first *modulates* the message (i.e., a MAC frame) into a time sequence of *digital baseband signals*. Digital baseband signals are then passed to the radio front-end, where they are converted to analog waveform, multiplied by a high frequency *carrier* and transmitted into the wireless channel. At the receiver side, the radio front-end receives radio signals in the channel and extracts the baseband waveform by removing the high-frequency carrier. The extracted baseband waveform is digitalized and converted back into digital signals. Then, the digital baseband signals are fed into the receiver's PHY layer to be *demodulated* into the original message.

The PHY layer directly operates on the digital baseband signals after modulation on the transmitter side and before demodulation on the receiver side. Therefore, high-throughput interfaces are needed to connect the PHY layer and the radio front-end. The required throughput linearly scales with the bandwidth of the baseband signal as well as the number of antennas in a MIMO system. For example, the channel width is 20MHz in 802.11a. It requires a data rate of at least 20M complex samples per second to represent the waveform. These complex samples normally require 16-bit quantization for both in-phase and quadrature (I/Q) components to provide sufficient fidelity, translating into 32 bits per sample, or 640Mbps for the full 20 MHz channel. Oversampling, a technique widely used for better performance,[11] doubles the requirement to 1.28Gbps. With a 4×4 MIMO and 40-MHz channel, as specified in 802.11n, it will again quadruple the requirement to 10Gbps to move data between the RF frond-end and PHY for one channel.

Advanced communication systems (e.g., IEEE 802.11a/b/g, as shown in Figure 1) contain multiple functional blocks in their PHY components. These functional blocks are pipelined with one another. Data are streamed through these blocks sequentially, but with different data types and sizes. As illustrated in Figure 1, different blocks may consume or produce different types of data in different rates arranged in small data blocks. For example, in 802.11b, the scrambler may consume and produce one bit, while DQPSK modulation maps each two-bit data block onto a complex symbol, whose real and image components represent I and Q, respectively.

Each PHY block performs a fixed amount of computation on every transmitted or received bit. When the data rate is high, e.g., 11Mbps for 802.11b and 54Mbps for 802.11a/g, PHY processing blocks consume a significant amount of computational power. Based on the model in Neel et al.,[16] we estimate that a direct implementation of 802.11b may require 10GOPS while 802.11a/g needs at least 40GOPs.

Figure 1. PHY operations of IEEE 802.11a/b/g transceiver.



These requirements are very demanding for software processing in GPPs.

2.2. Wireless MAC

The wireless channel is a resource shared by all transceivers operating on the same spectrum. As simultaneously transmitting neighbors may interfere with each other, various MAC protocols have been developed to coordinate their transmissions in wireless networks to avoid collisions.

Most modern MAC protocols, such as 802.11, require timely responses to critical events. For example, 802.11 adopts a carrier sense multiple access (CSMA) MAC protocol to coordinate transmissions. Transmitters are required to sense the channel before starting their transmission, and channel access is only allowed when no energy is sensed, i.e., the channel is free. The latency between *sense* and *access* should be as small as possible. Otherwise, the sensing result could be outdated and inaccurate. Another example is the link-layer retransmission mechanisms in wireless protocols, which may require an immediate acknowledgement (ACK) to be returned in a limited time window.

Commercial standards like IEEE 802.11 mandate a response latency within 16 μs, which is challenging to achieve in software on a general-purpose PC with a general-purpose OS.

2.3. Software radio requirements

Given the above discussion, we summarize the requirements for implementing a software radio system on a general PC platform:

High-system throughput. The interfaces between the radio front-end and PHY as well as between some PHY processing blocks must possess sufficiently high throughput to transfer high-fidelity digital waveforms.

Intensive computation. High-speed wireless protocols require substantial computational power for their PHY processing. Such computational requirements also increase proportionally with communication speed. Unfortunately, techniques used in conventional PHY hardware or embedded DSPs do not directly carry over to GPP architectures. Thus, we require new software techniques to accelerate high-speed signal processing on GPPs. With the advent of many-core GPP architectures, it is now reasonable to aggregate computational power of multiple CPU cores for signal processing. But, it is still challenging to build a software architecture to efficiently exploit the full capability of multiple cores.

Real-time enforcement. Wireless protocols have multiple *real-time deadlines* that need to be met. Consequently, not only is processing throughput a critical requirement, but the processing latency needs to meet response deadlines. Some MAC protocols also require precise timing control at the granularity of microseconds to ensure certain actions occur at exactly pre-scheduled time points. Meeting such real-time deadlines on a general PC architecture is a non-trivial challenge: time sharing operating systems may not respond to an event in a timely manner, and bus interfaces, such as Gigabit Ethernet, could introduce indefinite delays far more than a few microseconds. Therefore, meeting these real-time requirements requires new mechanisms on GPPs.

3. ARCHITECTURE

We have developed a high-performance software radio platform called Sora that addresses these challenges. It is based on a commodity general-purpose PC architecture. For flexibility and programmability, we push as much communication functionality as possible into software, while keeping hardware additions as simple and generic as possible. Figure 2 illustrates the overall system architecture.

Figure 2. Sora system architecture. All PHY and MAC execute in software on a commodity multi-core CPU.

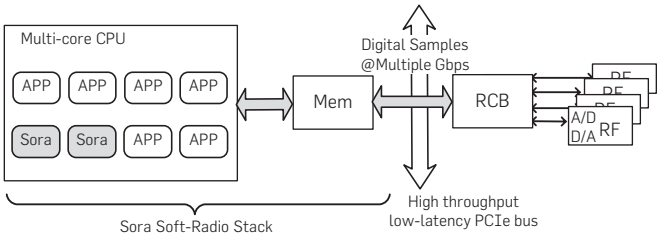
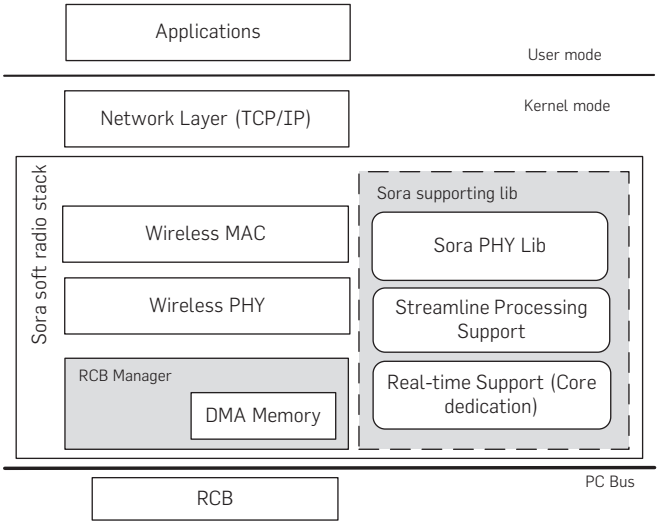


Figure 3. Software architecture of Sora soft-radio stack.



3.1. Hardware components

The hardware components in the Sora architecture are a new RCB with an interchangeable radio front-end (RF front-end). The radio front-end is a hardware module that receives and/or transmits radio signals through an antenna. In the Sora architecture, the RF front-end represents the well-defined interface between the digital and analog domains. It contains analog-to-digital (A/D) and digital-to-analog (D/A) converters, and necessary circuitry for radio transmission. Since all signal processing is done in software, the RF front-end design can be rather generic. It can be implemented in a self-contained module with a standard interface to the RCB. Multiple wireless technologies defined on the same frequency band can use the same RF front-end hardware, and the RCB can connect to different RF front-ends designed for different frequency bands.

The RCB is a new PC interface board for establishing a high-throughput, low-latency path for transferring high-fidelity digital signals between the RF front-end and PC memory. To achieve the required system throughput discussed in Section 2.1, the RCB uses a high-speed, low-latency bus such as PCIe. With a maximum throughput of 64Gbps (PCIe $\times$ 32) and sub-microsecond latency, it is well suited for supporting multiple gigabit data rates for wireless signals over a very wide band or over many MIMO channels. Further, the PCIe interface is now common in contemporary commodity PCs.

Another important role of the RCB is to bridge the synchronous data transmission at the RF front-end and the asynchronous processing on the host CPU. The RCB uses various buffers and queues, together with a large onboard memory, to convert between synchronous and asynchronous streams and to smooth out bursty transfers between the RCB and host memory. The large onboard memory further allows caching precomputed waveforms, adding additional flexibility for software radio processing.

Finally, the RCB provides a low-latency control path for software to control the RF front-end hardware and to ensure it is properly synchronized with the host CPU. Section 5.1 describes our implementation of the RCB in more detail.

3.2. Sora software

Figure 3 illustrates Sora's software architecture. The software components in Sora provide necessary system services and programming support for implementing various wireless PHY and MAC protocols in a general-purpose operating system. In addition to facilitating the interaction with the RCB, Sora provides a set of techniques to greatly improve the performance of PHY and MAC processing on GPPs. To meet the processing and real-time requirements, these techniques make full use of various common features in existing multi-core CPU architectures, including the extensive use of LUTs, substantial data-parallelism with CPU SIMD extensions, the efficient partitioning of streamlined processing over multiple cores, and exclusive dedication of cores for software radio tasks. We describe these software techniques in details in the next section.

4. HIGH-PERFORMANCE SDR SOFTWARE

4.1. Efficient PHY processing

In a memory-for-computation trade-off, Sora relies upon the large-capacity, high-speed cache memory in GPPs to accelerate PHY processing with precalculated LUTs. Contemporary modern CPU architectures usually have megabytes of L2 cache with a low (10–20 cycles) access latency. If we precalculate LUTs for a large portion of PHY algorithms, we can greatly reduce the computational requirement for online processing.

For example, the *soft demapper* algorithm used in demodulation needs to calculate the *confidence level* of each bit contained in an incoming symbol. This task involves rather complex computation proportional to the modulation density. More precisely, it conducts an extensive search for all modulation points in a constellation graph and calculates a ratio between the minimum of Euclidean distances to all points representing one and the minimum of distances to all points representing zero. In this case, we can precalculate the confidence levels for all possible incoming symbols based on their I and Q values, and build LUTs to directly map the input symbol to confidence level. Such LUTs are not large. For example, in 802.11a/g with a 54Mbps modulation rate (64-QAM), the size of the LUT for the soft demapper is only 1.5KB.

As we detail later in Section 5.2.1, more than half of the common PHY algorithms can indeed be rewritten with LUTs, each with a speedup from 1.5× to 50×. Since the size of each LUT is sufficiently small, the sum of all LUTs in a processing path can easily fit in the L2 caches of contemporary GPP cores. With *core dedication* (Section 4.3), the possibility of cache collisions is very small. As a result, these LUTs are almost always in caches during PHY processing.

To accelerate PHY processing with data-level parallelism, Sora heavily uses the SIMD extensions in modern GPPs, such as SSE, 3DNow! and AltiVec. Although these extensions were designed for multimedia and graphics applications, they also match the needs of wireless signal processing very well because many PHY algorithms have fixed computation structures that can easily map to large vector operations.

4.2. Multi-core streamline processing

Even with the above optimizations, a single CPU core may not have sufficient capacity to meet the processing requirements of high-speed wireless communication technologies. As a result, Sora must be able to use more than one core in a multi-core CPU for PHY processing. This multi-core technique should also be scalable because the signal processing algorithms may become increasingly more complex as wireless technologies progress.

As discussed in Section 2, PHY processing typically contains several functional blocks in a pipeline. These blocks differ in processing speed and in input/output data rates and units. A block is only *ready* to execute when it has sufficient input data from the previous block. Therefore, a key issue is how to schedule a functional block on multiple cores when it is ready.

Sora chooses a static scheduling scheme. This decision is based on the observation that the schedule of each block in a PHY processing pipeline is actually static: the processing pattern of previous blocks can determine whether a subsequent block is ready or not. Sora can thus partition the whole PHY processing pipeline into several sub-pipelines and statically assign them to different cores. Within one sub-pipeline, when a block has accumulated enough data for the next block to be ready, it explicitly schedules the next block. Adjacent sub-pipelines are still connected with a synchronized FIFO (SFIFO), but the number of SFIFOs and their overhead are greatly reduced.

4.3. Real-time support

SDR processing is a time-critical task that requires strict guarantees of computational resources and hard real-time deadlines. As an alternative to relying upon the full generality of real-time operating systems, we can achieve real-time guarantees by simply dedicating cores to SDR processing in a multi-core system. Thus, sufficient computational resources can be guaranteed without being affected by other concurrent tasks in the system.

This approach is particularly plausible for SDR. First, wireless communication often requires its PHY to constantly monitor the channel for incoming signals. Therefore, the PHY processing may need to be active all the time. It is much better to always schedule this task on the same core to minimize overhead like cache misses or TLB flushes. Second, previous work on multi-core OSes also suggests that isolating applications into different cores may have better performance compared to symmetric scheduling, since an effective use of cache resources and a reduction in locks can outweigh dedicating cores.[9] Moreover, a *core dedication* mechanism is much easier to implement than a real-time scheduler, sometimes even without modifying an OS kernel. For example, we can simply raise the priority of a kernel thread so that it is pinned on a core and it exclusively runs until termination (Section 5.2.3).

5. IMPLEMENTATION

5.1. Hardware

We have designed and implemented the Sora RCB as shown in Figure 4. It contains a Virtex-5 FPGA, a PCIe-×8 interface, and 256MB of DDR2 SDRAM. The RCB can connect to various RF front-ends. In our experimental prototype, we use a third-party RF front-end that is capable of transmitting and receiving a 20 MHz channel at 2.4 or 5 GHz.

Figure 5 illustrates the logical components of the Sora hardware platform. The DMA and PCIe controllers interface with the host and transfer digital samples between the RCB and PC memory. Sora software sends commands and reads RCB states through RCB registers. The RCB uses its onboard SDRAM as well as small FIFOs on the FPGA chip to bridge data streams between the CPU and RF front-end. When receiving, digital signal samples are buffered in on-chip FIFOs and delivered into PC memory when they fit

Figure 4. Sora radio control board.

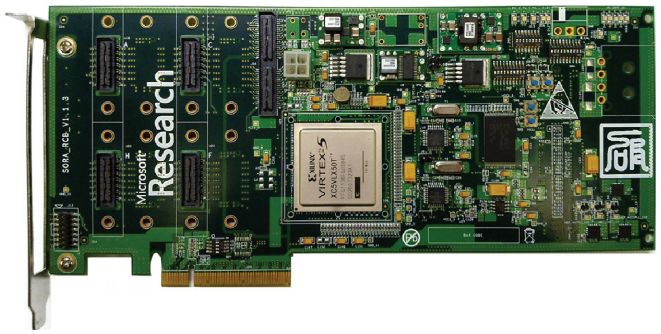
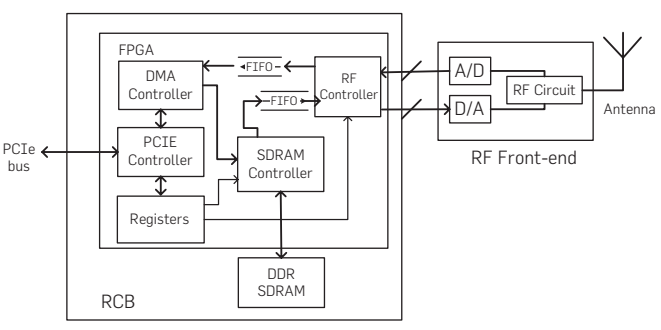


Figure 5. Hardware architecture of RCB and RF.



in a DMA burst (128B). When transmitting, the large RCB memory enables Sora software to first write the generated samples onto the RCB, and then trigger transmission with another command to the RCB. This functionality provides flexibility to the Sora software for precalculating and storing several waveforms before actually transmitting them, while allowing precise control of the timing of the waveform transmission.

While implementing Sora, we encountered a consistency issue in the interaction between DMA operations and the CPU cache system. When a DMA operation modifies a memory location that has been cached in the L2 cache, it does not invalidate the corresponding cache entry. When the CPU reads that location, it can therefore read an incorrect value from the cache.

We solve this problem with a *smart-fetch* strategy, enabling Sora to maintain cache coherency with DMA memory without drastically sacrificing throughput if disabling cached accesses. First, Sora organizes DMA memory into small slots, whose size is a multiple of a cache line. Each slot begins with a *descriptor* that contains a flag. The RCB sets the flag after it writes a full slot of data, and clears it after the CPU processes all data in the slot. When the CPU moves to a new slot, it first reads its descriptor, causing a whole cache line to be filled. If the flag is set, the data just fetched is valid and the CPU can continue processing the data. Otherwise, the RCB has not updated this slot with new data. Then, the CPU explicitly flushes the cache line and repeats reading the same location. This next read refills the cache line, loading the most recent data from memory.

Table 1 summarizes the RCB throughput results, which agree with the hardware specifications. To precisely measure PCIe latency, we instruct the RCB to read a memory address in host memory, and measure the time interval between issuing the request and receiving the response in hardware. Since each *read* involves a round trip operation, we use half of the measured time to estimate the one-way delay. This one-way delay is 360 ns with a worst case variation of 4 ns.

5.2. Software

The Sora software is written in C, with some assembly for performance-critical processing. The entire Sora software is implemented on Windows XP as a network device driver and it exposes a virtual Ethernet interface to the upper TCP/IP stack. Since any software radio implemented on Sora can appear as a normal network device, all existing network applications can run unmodified on it.

PHY Processing Library: In the Sora PHY processing library, we extensively exploit the use of look-up tables (LUTs) and SIMD instructions to optimize the performance of PHY algorithms. We have been able to rewrite more than half of the PHY algorithms with LUTs. Some LUTs are straightforward precalculations, others require more sophisticated implementations to keep the LUT size small. For the soft-demapper example mentioned earlier, we can greatly reduce the LUT size (e.g., 1.5KB for the 802.11a/g 54Mbps modulation) by exploiting the symmetry of the algorithm. In our SoftWiFi implementation described below, the overall size of the LUTs is around 200KB for 802.11a/g and 310KB for 802.11b, both of which fit comfortably within the L2 caches of commodity CPUs.

We also heavily use SIMD instructions in coding Sora software. We currently use the SSE2 instruction set designed for Intel CPUs. Since the SSE registers are 128-bit wide while most PHY algorithms require only 8-bit or 16-bit fixed-point operations, one SSE instruction can perform 8 or 16 simultaneous calculations. SSE also has rich instruction support for flexible data permutations, and most PHY algorithms, e.g., FFT, FIR Filter and Viterbi, can fit naturally into this SIMD model. For example, the Sora Viterbi decoder uses only 40 cycles to compute the branch metric and select the shortest path for each input. As a result, our Viterbi implementation can handle 802.11a/g at the 54Mbps modulation with only one 2.66 GHz CPU core, whereas previous implementations relied on hardware implementations. Note that other GPP architectures, like AMD and PowerPC, have very similar SIMD models and instruction sets, and we expect that our optimization techniques will directly apply to these other GPP architectures as well.

Table 2 summarizes some key PHY processing algorithms we have implemented in Sora, together with the optimization techniques we have applied. The table also compares the performance of a conventional software implementation (e.g., a direct translation from a hardware implementation) and the Sora implementation with the LUT and SIMD optimizations.

Lightweight, Synchronized FIFOs: Sora allows different PHY processing blocks to streamline across multiple cores, and we have implemented a lightweight, synchronized FIFO to connect these blocks with low contention overhead. The idea is to augment each data slot in the FIFO with a header that indicates whether the slot is empty or not. We pad each data slot to be a multiple of a cache line. Thus, the consumer is always chasing the producer in the circular buffer for filled slots. If the speed of the producer and consumer is the same and the two pointers are separated by a particular offset (e.g., two cache lines in the Intel architecture), no cache miss will occur during synchronized streaming since the local cache will prefetch the following slots before the actual access. If the producer and the consumer have different processing speeds, e.g., the reader is faster than the writer, then eventually the consumer will wait for the producer to release a slot. In this case, each time the producer writes to a slot, the write will cause a cache miss at the consumer. But the producer will not suffer a miss since the next free slot will be prefetched into its local cache. Fortunately, such cache misses experienced by the consumer will not cause significant impact on the overall performance of the streamline processing since the consumer

Table 1. DMA throughput performance of the RCB.

Mode	Rx (Gbps)	Tx (Gbps)
PCIe-x4	6.71	6.55
PCIe-x8	12.8	12.3

Table 2. Key algorithms in IEEE 802.11b/a and their performance with conventional and Sora implementations.

Algorithm	Configuration	I/O Size (bit)		Optimization Method	Computation Required (Mcycles/s)		
		Input	Output		Conventional Implementation	Sora Implementation	Speedup
IEEE 802.11b							
Scramble	11Mbps	8	8	LUT	96.54	10.82	8.9×
Descramble	11Mbps	8	8	LUT	95.23	5.91	16.1×
Mapping and spreading	2Mbps, DQPSK	8	44 × 16 × 2	LUT	128.59	73.92	1.7×
CCK modulator	5Mbps, CCK	8	8 × 16 × 2	LUT	124.93	81.29	1.5×
	11Mbps, CCK	8	8 × 16 × 2	LUT	203.96	110.88	1.8×
FIR filter	16-bit I/Q, 37 taps, 22MSps	16 × 2 × 4	16 × 2 × 4	SIMD	5,780.34	616.41	9.4×
Decimation	16-bit I/Q, 4× Oversample	16 × 2 × 4 × 4	16 × 2 × 4	SIMD	422.45	198.72	2.1×
IEEE 802.11a							
FFT/IFFT	64 points	64 × 16 × 2	64 × 16 × 2	SIMD	754.11	459.52	1.6×
Conv. encoder	24Mbps, 1/2 rate	8	16	LUT	406.08	18.15	22.4×
	48Mbps, 2/3 rate	16	24	LUT	688.55	37.21	18.5×
	54Mbps, 3/4 rate	24	32	LUT	712.10	56.23	12.7×
Viterbi	24Mbps, 1/2 rate	8 × 16	8	SIMD+LUT	68,553.57	1,408.93	48.7×
	48Mbps, 2/3 rate	8 × 24	16	SIMD+LUT	117,199.6	2,422.04	48.4×
	54Mbps, 3/4 rate	8 × 32	24	SIMD+LUT	131,017.9	2,573.85	50.9×
Soft demapper	24Mbps, QAM 16	16 × 2	8 × 4	LUT	115.05	46.55	2.5×
	54Mbps, QAM 64	16 × 2	8 × 6	LUT	255.86	98.75	2.4×
Scramble and descramble	54Mbps	8	8	LUT	547.86	40.29	13.6×

is not the bottleneck element.

Real-Time Support: Sora uses *exclusive threads* (or *ethreads*) to dedicate cores for real-time SDR tasks. Sora implements ethreads without any modification to the kernel code. An ethread is implemented as a kernel-mode thread, and it exploits the *processor affiliation* that is commonly supported in commodity OSes to control on which core it runs. Once the OS has scheduled the ethread on a specified physical core, it will raise its IRQL (interrupt request level) to a level as high as the kernel scheduler, e.g., dispatch_level in Windows. Thus, the ethread takes control of the core and prevents itself from being preempted by other threads.

Running at such an IRQL, however, does not prevent the core from responding to hardware interrupts. Therefore, we also constrain the *interrupt affiliations* of all devices attached to the host. If an ethread is running on one core, all interrupt handlers for installed devices are removed from the core, thus prevent the core from being interrupted by hardware. To ensure the correct operation of the system, Sora always ensures core zero is able to respond to all hardware interrupts. Consequently, Sora only allows ethreads to run on cores whose ID is greater than zero.

6. EXPERIENCE

To demonstrate the use of Sora, we have developed two wireless systems fully in software in a multi-core PC, namely SoftWiFi and SoftLTE. The performance we report for SoftWiFi is measured on an Intel Core Duo 2 (2.67 GHz), and the performance reported for SoftLTE is measured on an Intel Core i7-920 (2.67 GHz).

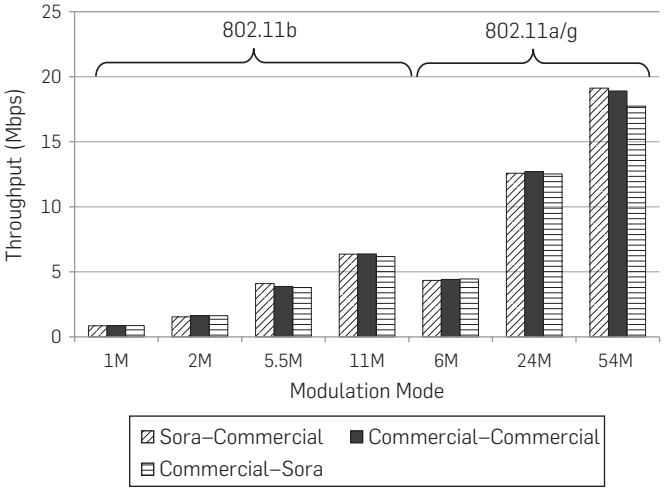
6.1. SoftWiFi

SoftWiFi implements the basic access mode of 802.11. The MAC state machine (SM) is implemented as an ethread. Since 802.11 is a simplex radio, the demodulation components can run directly within a MAC SM thread. If a single core is insufficient for all PHY processing (e.g., 802.11a/g), the PHY processing can be partitioned across two ethreads. These two ethreads are streamed using a synchronized FIFO. Two additional auxiliary threads modulate the outgoing frames in the background and transfer the demodulated frames to upper layers, respectively.

In idle state, the SM continuously measures the average energy to determine whether the channel is clean or there is an incoming frame. If it detects a high energy, SoftWiFi starts to demodulate a frame. After successfully receiving a frame, the 802.11 MAC standard requires a station to transmit an ACK frame in a timely manner (10 μs for 802.11b and 16 μs for 802.11a). This ACK requirement is quite difficult for an SDR implementation in software on a PC. Both generating and transferring the waveform across the PC bus will cause a latency of several microseconds, and the sum is usually larger than mandated by the standard.

Fortunately, an ACK frame generally has a fixed pattern with only a few dynamic fields (i.e., sender address). Thus, we can precalculate most of an ACK frame (19B), and update only the address (10B) on the flight. We can further do it immediately after demodulating the MAC header, and without waiting for the end of a frame. We then prestore the waveform in the memory of the RCB. Thus, the time for ACK

Figure 6. Throughput of Sora when communicating with a commercial WiFi card. *Sora–Commercial* presents the transmission throughput when a Sora node sends data. *Commercial–Sora* presents the throughput when a Sora node receives data. *Commercial–Commercial* presents the throughput when a commercial NIC communicates with another commercial NIC.



generation and transferring can overlap with the demodulation of the data frame. After the entire frame is demodulated and validated, SoftWiFi instructs the RCB to transmit the ACK which has already been stored in the RCB. Thus, the latency for ACK transmission is very small.

Figure 6 shows the transmitting and receiving throughput of a Sora SoftWiFi node when it communicates with a commercial WiFi NIC. In the "Sora–Commercial" configuration, the Sora node acts as a sender and generates 1400-byte UDP frames and unicast transmits them to a laptop equipped with a commercial NIC. In the "Commercial–Sora" configuration, the Sora node acts as a receiver, and the laptop generates the same workload. The "Commercial–Commercial" configuration shows the throughput when both sender and receiver are commercial NICs. In all configurations, the hosts were at the same distance from each other and experienced very little packet loss. Figure 6 shows the throughput achieved for all configurations with the various modulation modes in 11a/b/g. We show only three selective rates in 11a/g for conciseness. The results are averaged over five runs (the variance was very small).

We make a number of observations from these results. First, the Sora SoftWiFi implementation operates seamlessly with commercial devices, showing that Sora SoftWiFi is protocol compatible. Second, Sora SoftWiFi can achieve similar performance as commercial devices. The throughputs for both configurations are essentially equivalent, demonstrating that SoftWiFi (1) has the processing capability to demodulate all incoming frames at full modulation rates, and (2) it can meet the 802.11 timing constraints for returning ACKs within the delay window required by the standard. We note that the maximal achievable application throughput for 802.11 is less than 80% of the PHY data rate, and the percentage decreases as the PHY data rate increases. This limit is due to the overhead of headers at different layers as well as the MAC overhead to coordinate channel access (i.e., carrier sense, ACKs, and backoff), and is a well-known property of 802.11 performance.

6.2. SoftLTE

We have also implemented the 3GPP LTE Physical Uplink Shared Channel (PHUSC) on the Sora platform.[13] LTE is the next generation cellular standard. It is more complex than 802.11 since it uses a higher-order FFT (1024-point) and advanced coding/decoding algorithms (e.g., Turbo coding). Our SoftLTE implementation on Sora provides a peak data rate of 43.8Mbps with a 20-MHz channel, 16QAM modulation, and 3/4 Turbo coding. The most computationally intensive component of an LTE PHY is the Turbo decoder. Our current implementation can achieve 35Mbps throughput using one hardware thread of an Intel Core i7-920 core (2.66 GHz). Since Core i7 supports hyperthreading, though, we can execute the Turbo decoder in parallel on two threads, achieving an aggregated throughput of 54.8Mbps. We can achieve this performance because Turbo decoding is relatively balanced in the number of arithmetic instructions and memory accesses. Therefore, the two threads can overlap these two kinds of operations well and yield a 56% performance gain even though they share the same execution units of a single core. Thus, the whole SoftLTE implementation can run in real time with two Intel Core i7 cores.

7. RELATED WORK

Traditionally, device drivers have been the primary software mechanism for changing wireless functionality on general-purpose computing systems. For example, the MadWiFi drivers for cards with Atheros chipsets,[3] HostAP drivers for Prism chipsets,[2] and the rtx200 drivers for RaLink chipsets[5] are popular driver suites for experimenting with 802.11. These drivers typically allow software to control a wide range of 802.11 management tasks and non-time-critical aspects of the MAC protocol, and allow software to access some device hardware state and exercise limited control over device operation (e.g., transmission rate or power). However, they do not allow changes to fundamental aspects of 802.11 like the MAC packet format or any aspects of PHY.

SoftMAC goes one step further to provide a platform for implementing customized MAC protocols using inexpensive commodity 802.11 cards.[17] Based on the MadWiFi drivers and associated open-source hardware abstraction layers, SoftMAC takes advantage of features of the Atheros chipsets to control and disable default low-level MAC behavior. SoftMAC enables greater flexibility in implementing nonstandard MAC features, but does not provide a full platform for SDR. With the separation of functionality between driver software and hardware firmware on commodity devices, time critical tasks and PHY processing remain unchangeable.

GNU Radio is a popular software toolkit for building software radios using general-purpose computing platforms.[1] GNU Radio consists of a software library and a hardware platform. Developers implement software radios

by composing modular precompiled components into processing graphs using Python scripts. The default GNU Radio platform is the Universal Software Radio Peripheral (USRP), a configurable FPGA radio board that connects to the host. As with Sora, GNU Radio performs much of the SDR processing on the host itself. Current USRP supports USB2.0 and a new version USRP 2.0 upgrades to Gigabit Ethernet. Such interfaces, though, are not sufficient for high-speed wireless protocols in wide bandwidth channels. Existing USRP/GNU Radio platforms can only sustain low-speed wireless communication due to both the hardware constraints as well as software processing.[18] As a consequence, users must sacrifice radio performance for its flexibility.

The WARP hardware platform provides a high-performance SDR platform.[8] Based on Xilinx FPGAs and PowerPC cores, WARP allows full control over the PHY and MAC layers and supports customized modulations up to 36Mbps. A variety of projects have used WARP to experiment with new PHY and MAC features, demonstrating the impact a high-performance SDR platform can provide. KUAR is another SDR development platform.[15] Similar to WARP, KUAR mainly uses Xilinx FPGAs and PowerPC cores for signal processing. But it also contains an embedded PC as the control processor host (CPH), enabling some communication systems to be implemented completely in software on the CPH. Sora provides the same flexibility and performance as hardware-based platforms, like WARP, but it also provides a familiar and powerful programming environment with software portability at a lower cost.

The SODA architecture represents another point in the SDR design space.[14] SODA is an application domain-specific multiprocessor for SDR. It is fully programmable and targets a range of radio platforms—four such processors can meet the computational requirements of 802.11a and W-CDMA. Compared to WARP and Sora, as a single-chip implementation it is more appropriate for embedded scenarios. As with WARP, developers must program to a custom architecture to implement SDR functionality.

8. CONCLUSION

This paper presented Sora, a fully programmable software radio platform on commodity PC architectures. Sora combines the performance and fidelity of hardware SDR platforms with the programmability of GPP-based SDR platforms. Using the Sora platform, we also present the design and implementation of SoftWiFi, a software implementation of the 802.11a/b/g protocols, and SoftLTE, a software implementation of the LTE uplink PHY.

The flexibility provided by Sora makes it a convenient platform for experimenting with novel wireless protocols. In our research group, we have extensively used Sora to implement and evaluate various ideas in our wireless research projects. For example, we have built a spatial multiplexing system with 802.11b.[19] In this work, we implemented not only a complex PHY algorithm with successive interference cancellation, but also a sophisticated carrier-counting multi-access (CCMA) MAC—implementations would not have been possible with previous PC-based software radio platforms.

Sora is now available for academic use as the MSR Software Radio Kit.[4] The Sora hardware can be ordered from a vender company in Beijing and all software can be downloaded for free from Microsoft Research website. Our hope is that Sora can substantially contribute to the adoption of SDR for wireless networking experimentation and innovation.

Acknowledgments

The authors would like to thank Xiongfei Cai, Ningyi Xu, and Zenlin Xia in the Hardware Computing group at MSRA for their essential assistance in the hardware design of the RCB. We also thank Fan Yang and Chunyi Peng in the Wireless Networking (WN) Group at MSRA; in particular we have learned much from their early study on accelerating 802.11a using GPUs. We would also like to thank all members in the WN Group and Zheng Zhang for their support and feedback. The authors also want to thank Songwu Lu, Frans Kaashoek, and MSR colleagues (Victor Bahl, Ranveer Chandra, etc.) for their comments on earlier drafts of this paper. C

References

1. GNU Radio. http://www.gnu.org/software/gnuradio/.
2. HostAP. http://hostap.epitest.fi/.
3. MadWifi. http://sourceforge.net/projects/madwifi.
4. Microsoft Research Software Radio Platform. http://research.microsoft.com/enus/projects/sora/academickit.aspx.
5. Rt2x00. http://rt2x00.serialmonkey.com.
6. Small Form Factor SDR Development Platform. http://www.xilinx.com/products/devkits/SFF-SDR-DP.htm.
7. Universal Software Radio Peripheral. http://www.ettus.com/.
8. WARP: Wireless Open Access Research Platform. http://warp.rice.edu/trac.
9. Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., Zhang Z. Corey: an operating system for many cores. In *OSDI* 2008.
10. Cummings, M., Haruyama, S. FPGA in the Software Radio. *IEEE Commun. Mag.* 1999.
11. de Vegte, J.V. *Fundamental of Digital Signal Processing*. Cambridge University Press, 2005.
12. Glossner, J., Hokenek, E., Moudgill, M. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors* (2004).
13. Li, Y., Fang, J., Tan, K., Zhang, J., Cui, Q., Tao, X. Soft-LTE: a software radio implementation of 3GPP long term evolution based on Sora platform. In *ACM Moicom 2009 (Demonstration)* (Beijing, 2009).
14. Lin, Y., Lee, H., who, M., Harel, Y., Mahlke, S., Mudge, T. SODA: a low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd International Symposium onComputer Architecture* (2006).
15. Minden, G.J., Evans, J.B., Searl, L., DePardo, D., Patty, V.R., Rajbanshi, R., Newman, T., Chen, Q., Weidling, F., Guffey, J., Datla, D., Barker, B., Peck, M., Cordill, B., Wyglinski, A.M., Agah, A. KUAR: a flexible software-defined radio development platform. In *DySpan* (2007).
16. Neel, J., Robert, P., Reed, J. A formal methodology for estimating the feasible processor solution space for a software radio. In *SDR'05: Proceedings of the SDR Technical Conference and Product Exposition* (2005).
17. Neufeld, M., Fifield, J., Doerr, C., Sheth, A., Grunwald, D. SoftMAC—flexible wireless research platform. In *HotNets'05* (2005).
18. Schmid, T., Sekkat, O., Srivastava, M.B. An experimental study of network performance impact of increased latency in software defined radios. In *WiNETCH'07* (2007).
19. Tan, K., Liu, H., Fang, J., Wang, W., Zhang, J., Chen, M., Voelker, G.M. SAM: enabling practical spatial multiple access in wireless LAN. In *MobiCom'09: Proceedings of the 15th Annual International Conference on Mobile Computing and Networking* (New York, NY, 2009), ACM, USA, 49–60.

Kun Tan (kuntan@microsoft.com), Microsoft Research Asia, Beijing, China.

He Liu (h8liu@ucsd.edu), University of California, San Diego, La Jolla, CA.

Jiansong Zhang (kuntan@microsoft.com), Microsoft Research Asia, Beijing, China.

Yongguang Zhang (ygz@microsoft.com), Microsoft Research Asia, Beijing, China.

Ji Fang (v-fangji@microsoft.com), Microsoft Research Asia and Beijing Jiaotong University, Beijing, China.

Geoffrey M. Voelker (voelker@cs.ucsd.edu), University of California, San Diego, La Jolla, CA.