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

Kun Tan[†] Jiansong Zhang[†] Ji Fang [‡] He Liu [§] Yusheng Ye[§]
Shen Wang[§] Yongguang Zhang[†] Haitao Wu[†] Wei Wang[†] Geoffrey M. Voelker[♮]

[†]*Microsoft Research Asia, Beijing, China* [§] *Tsinghua University, Beijing, China*
[‡] *Beijing Jiaotong University, Beijing, China* [♮] *UCSD, La Jolla, USA*

Abstract

This paper presents Sora, a fully programmable software radio platform on commodity PC architectures. Sora combines the performance and fidelity of hardware 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 demonstration radio system called SoftWiFi. SoftWiFi seamlessly interoperates with commercial 802.11a/b/g NICs, and achieves equivalent performance as commercial NICs at each modulation.

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) [6, 11] or embedded digital signal processors (DSPs) [5, 13]. 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. Hardware platforms can also be expensive; the WARP [6] educational price, for example, is over US$9,750.

In contrast, SDR platforms based on general-purpose processor (GPP) architectures, such as commodity PCs, have the opposite set of tradeoffs. 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, 22]. For example, the popular GNU Radio platform [1] achieves only a few Kbps throughput on an 8MHz channel [21], whereas modern high-speed wireless protocols like 802.11 support multiple Mbps data rates on a much wider 20MHz channel [7]. 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, 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. Existing GPP platforms like GNU Radio use USB 2.0 or

This work was performed when Ji Fang, He Liu, Yusheng Ye, and Shen Wang were visiting students and Geoffrey M. Voelker was a visiting researcher at Microsoft Research Asia.

Gigabit Ethernet [1], which cannot satisfy this requirement for high-speed wireless protocols. Second, physical layer (PHY) signal processing has very high computational requirements for generating information bits from waveforms, and vice versa, particularly at high modulation rates; indeed, back-of-the-envelope calculations for processing requirements on GPPs have instead motivated specialized hardware approaches in the past [17, 19]. 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 uses both hardware and software techniques to address the challenges of using PC architectures for high-speed SDR. 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 [8]. With this bus standard, the RCB can support 16.7Gbps (x8 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 also 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 SIMD (Single Instruction Multiple Data) extensions in existing processors to further accelerate PHY processing. With these optimizations, Sora can fully support the complete digital processing of 802.11b modulation rates on just one core, and 802.11a/g on two cores.

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 GPP.

We have developed a demonstration radio system, SoftWiFi, based on the Sora platform. SoftWiFi currently supports the full suite of 802.11a/b/g modulation rates, seamlessly interoperates with commercial 802.11 NICs, and achieves equivalent performance as commercial NICs at each modulation.

In summary, the contributions of this paper are: (1) the design and implementation of the Sora platform and its high-performance PHY processing library; (2) the design and implementation of the SoftWiFi radio system that can interoperate with commercial wireless NICs using 802.11a/b/g standards; and (3) the evaluation of Sora and SoftWiFi on a commodity multi-core PC. To the best of our knowledge, Sora is the first SDR platform that enables users to develop high-speed wireless implementations, such as the IEEE 802.11a/b/g PHY and MAC, entirely in software on a standard PC architecture.

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 presents the design and implementation of SoftWiFi, a fully functional software WiFi radio based on Sora, and we evaluate its performance in Section 7. Finally, Section 9 describes related work and Section 10 concludes.

2 Background and Requirements

In this section, we briefly review the physical layer (PHY) and media access (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 packet or a MAC frame) into a time sequence of *baseband signals*. Baseband signals are then passed to the radio front-end, where they are multiplied by a high frequency *carrier* and transmitted into the wireless channel. At the receiver side, the radio front-end detects signals in the channel and extracts the baseband signal by removing the high-frequency carrier. The extracted baseband signal is then fed into the receiver's PHY layer to be *demodulated* into the original message.

Advanced communication systems (*e.g.*, IEEE 802.11a/b/g, as shown in Figure 1) contain multiple

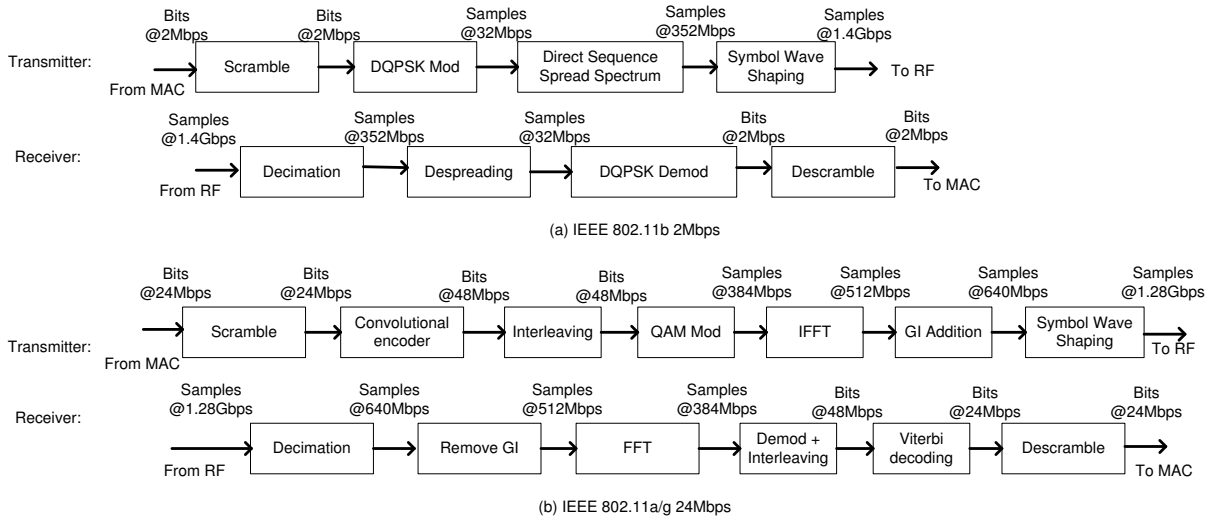

Figure 1: PHY operations of IEEE 802.11a/b/g transceiver.

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 which uses two 16-bit numbers to represent the in-phase and quadrature (I/Q) components.

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 [19], we estimate that a direct implementation of 802.11b may require 10Gops while 802.11a/g needs at least 40Gops. These requirements are very demanding for software processing in GPPs.

PHY processing blocks directly operate on the digital waveforms after modulation on the transmitter side and before demodulation on the receiver side. Therefore, high-throughput interfaces are needed to connect these processing blocks as well as to connect the PHY and radio front-end. The required throughput linearly scales with the bandwidth of the baseband signal. For example, the channel bandwidth is 20MHz in 802.11a. It requires a data rate of at least 20M complex samples per second to represent the waveform [14]. These complex samples normally require 16-bit quantization for both I and Q components to provide sufficient fidelity, translating into 32 bits per sample, or 640Mbps for the full 20MHz channel. Over-sampling, a technique widely used for better performance [12], doubles the requirement to 1.28Gbps to move data between the RF frond-end and PHY blocks for one 802.11a channel.

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 CSMA (Carrier-Sense Multiple Access) MAC protocol to coordinate transmissions [7]. 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 tens of microseconds, 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

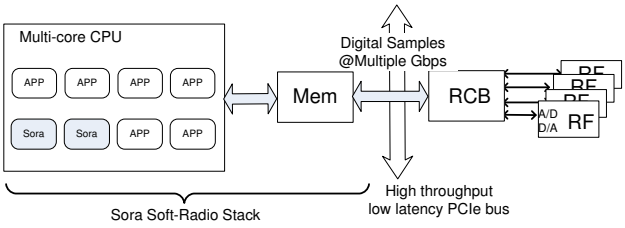

Figure 2: Sora system architecture. All PHY and MAC execute in software on a commodity multi-core CPU.

throughput to transfer high-fidelity digital waveforms. To support a 20MHz channel for 802.11, the interfaces must sustain at least 1.28Gbps. Conventional interfaces like USB 2.0 ($\leq$ 480Mbps) or Gigabit Ethernet ($\leq$ 1Gbps) cannot meet this requirement [1].

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 [9], it is now reasonable to dedicate computational power solely to 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 operation 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 μs. 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.

3.1 Hardware Components

The hardware components in the Sora architecture are a new radio control board (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. During receiving, the RF front-end acquires an analog waveform from the antenna, possibly down-converts it to a lower frequency, and then digitizes it into discrete samples before transferring them to the RCB. During transmitting, the RF front-end accepts a synchronous stream of software-generated digital samples and synthesizes the corresponding analog waveform before emitting it using the antenna. 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 [8]. With a maximum throughput of 64Gbps (PCIe x32) 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 on-board memory, to convert between synchronous and asynchronous streams and to smooth out bursty transfers between the RCB and host memory. The large on-board memory further allows caching pre-computed 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.

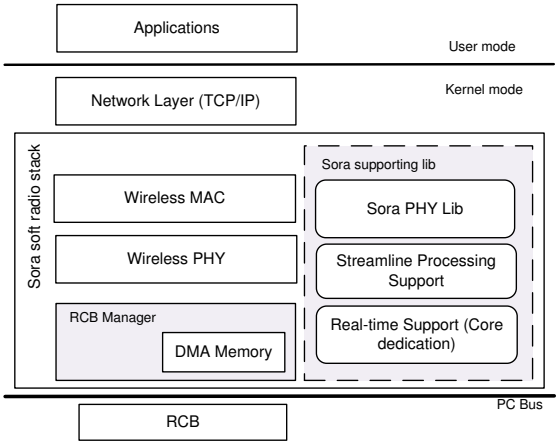

Figure 3: Software architecture of Sora soft-radio stack.

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, the Sora soft-radio stack 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 lookup tables (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.

4 High-Performance SDR Processing

In this section we describe the software techniques used by Sora to achieve high-performance SDR processing.

4.1 Efficient PHY processing

In a memory-for-computation tradeoff, Sora relies upon the large-capacity, high-speed cache memory in GPPs to accelerate PHY processing with pre-calculated lookup tables (LUTs). Contemporary modern CPU architectures, such as Intel Core 2, usually have megabytes of L2 cache with a low (10∼20 cycles) access latency. If we pre-calculate LUTs for a large portion of PHY algorithms, we can greatly reduce the computational requirement for on-line 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 pre-calculate 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 1.5x to 50x. 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. In Appendix A, we show an example of an optimized digital filter implementation using SSE instructions. As our measurements later show, such SIMD extensions substantially speed up PHY processing in Sora.

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.

One possible approach is to run multiple PHY pipelines on different cores (Figure 4(a)), and have the scheduler dispatch batches of digital samples to a

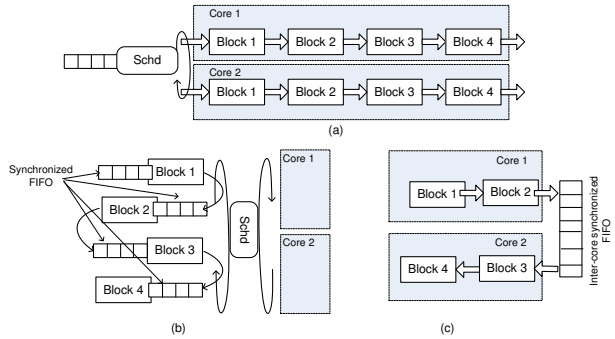

Figure 4: PHY pipeline scheduling: (a) parallel pipelines, (b) dynamic scheduling, (c) static scheduling.

pipeline. This approach, however, does not work well for SDR because wireless communication has strong dependencies in a data stream. For example, in convolutional encoding the output of each bit also depends on the seven preceding bits in the input stream. Without the scheduler knowing all of the data dependencies, it is difficult to produce an efficient schedule.

An alternative scheduling approach is to have only one pipeline and dynamically assign ready blocks to available cores (Figure 4(b)), in a way similar to thread scheduling in a multi-core system. Unfortunately, this approach would introduce prohibitively high overhead. On the one hand, any two adjacent blocks may be scheduled onto two different cores, thereby requiring synchronized FIFO (SFIFO) communication between them. On the other hand, most PHY processing blocks operate on very small data items, *e.g.*, 1–4 bytes each, and the processing only takes a few operations (several to tens of instructions). Such frequent FIFO and synchronization operations are not justifiable for such small computational tasks.

Instead, 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 (Figure 4(c)). 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 from different blocks are still connected with an 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 [10]. 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).

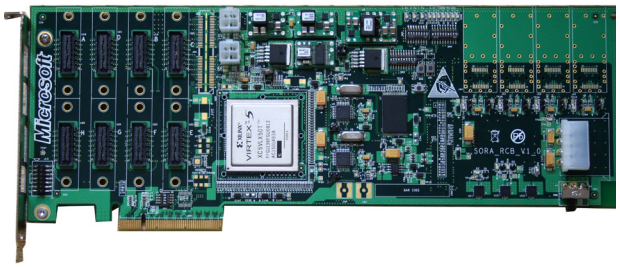

Figure 5: Sora radio control board.

5 Implementation

We have implemented both the hardware and software components of Sora. This section describes our hardware prototype and software stack, and presents microbenchmark evaluations of Sora components.

5.1 Hardware

We have designed and implemented the Sora radio control board (RCB) as shown in Figure 5. It contains a Virtex-5 FPGA, a PCIe-x8 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, developed by Rice University [6], that is capable of transmitting and receiving a 20MHz channel at 2.4GHz or 5GHz.

Figure 6 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 regis-

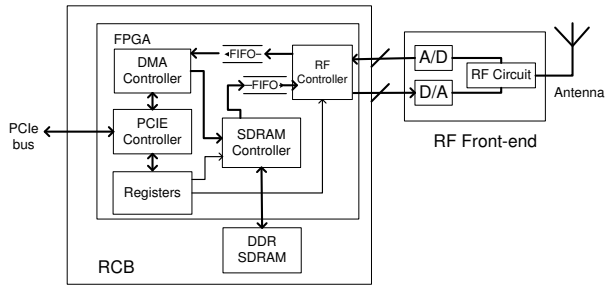

Figure 6: Hardware architecture of RCB and RF.

ters. The RCB uses its on-board 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 in a DMA burst (128 bytes). 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 pre-calculating 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. One naive solution is to disable cached accesses to memory regions used for DMA, but doing so will cause a significant degradation in memory access throughput.

We solve this problem with a *smart-fetch* strategy, enabling Sora to maintain cache coherency with DMA memory without drastically sacrificing throughput. 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 cleared 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.

5.2 Software

The Sora software is written in C, with some assembly for performance-critical processing. The entire Sora software stack 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.

The Sora software currently consists of 23,325 non-blank lines of C code. Of this total, 14,529 lines are for system support, including driver framework, memory management, streamline processing, etc. The remaining 8,796 lines comprise the PHY processing library.

5.2.1 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 pre-calculations, 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 used in 802.11a/g is around 200KB and 310KB in 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. SSE2 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 2.66GHz 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; AMD's Enhanced 3DNow!, for instance, includes SSE instructions plus a set of DSP extensions. We expect that our optimization techniques will directly apply to these other GPP architectures as well. In Appendix A, we show a simple example of a functional block using SIMD instruction optimizations.

Table 1 summarizes some key PHY processing algorithms we have implemented in Sora, together with the optimization techniques we have applied. The table also

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

Table 1: Key algorithms in IEEE 802.11b/a and their performance with conventional and Sora implementations.

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.

5.2.2 Lightweight, synchronized FIFOs

Sora allows different PHY processing blocks to streamline across multiple cores while communicating with one another through shared memory FIFO queues. If two blocks are running on different cores, their access to the shared FIFO must be synchronized. The traditional implementation of a synchronized FIFO uses a *counter* to synchronize the writer and reader, which we refer to as a counter-based FIFO (CBFIFO) and illustrate in Figure 7(a). However, this counter is shared by two processor cores, and every write to the variable by one core will cause a cache miss on the other core. Since both the producer and consumer modify this variable, two cache misses are unavoidable for each datum. It is also quite common to have very fine data granularity in PHY (*e.g.*, 4–16 bytes as summarized in Table 1). Therefore, such cache misses will result in significant overhead when synchronization has to be performed very frequently (*e.g.*, once per micro-second) for such small pieces of data.

In Sora, we implement another synchronized FIFO that removes the sole shared synchronization variable. 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, as outlined in Figure 7(b). This chasing-pointer FIFO (CPFIFO) largely mitigates the overhead even for very fine-grained synchronization. If the speed of the producer and consumer is

```
// producer:
void write_fifo ( DATA_TYPE data ) {
  while (cnt >= q_size); // spin wait
  q[w_tail] = data;
  w_tail = (w_tail+1) % q_size;
  InterlockedIncrement (cnt); // increase cnt by 1
 }
// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
  while (cnt==0); // spin wait
  * pdata = q[r_head];
  r_head = (r_head+1) % q_size;
  InterlockedDecrement(cnt); // decrease cnt by 1
 }
```

(a)

```
// producer:
void write_fifo ( DATA_TYPE data ) {
  while (q[w_tail].flag>0); // spin wait
  q[w_tail].data = data;
  q[w_tail].flag = 1; // occupied
  w_tail = (w_tail+1) % q_size;
}
// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
  while (q[r_head].flag==0); // spin
  *data = q[r_head].data;
  q[r_head].flag = 0; // release
  r_head = (r_head + 1) % q_size;
}
```

(b)

Figure 7: Pseudo-code for synchronized (a) CBFIFOs and (b) CPFIFOs.

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

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

Table 2: DMA throughput performance of the RCB.

Method	Memory Throughput
Cache Disabled	707.2Mbps
Smart-fetch	10.1Gbps

Table 3: Memory throughput.

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 is not the bottleneck element.

5.2.3 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.

5.3 Evaluation

We measure the performance of the Sora implementation with microbenchmark experiments. We perform all measurements on a Dell XPS PC with an Intel Core 2 Quad 2.66GHz CPU (Section 7.1 details the complete hardware configuration).

Throughput and latency. To measure PCIe throughput, we instruct the RCB to read/write a number of descriptors from/to main memory via DMA, and measure the time taken. Table 2 summarizes the results, which agree with the hardware specifications.

To precisely measure PCIe latency, we instruct the RCB to read a memory address in host memory. We measure the time interval between issuing the request and receiving the response data in hardware. Since the *memory read* operation accesses the PCIe bus using a round trip operation, we use half of the measured time to estimate the one-way delay. This one-way delay is $360ns$ with a worst case variation of $4ns$. We also confirm that the RCB hardware itself induces negligible delay except for buffers on the data path. However, such delay is tiny when the buffer is small. For example, the DMA burst size is 128 bytes, which causes only $76ns$ latency in PCIe-x8.

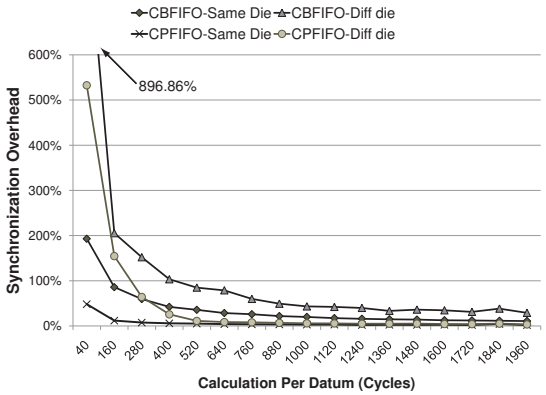

Figure 8: Overhead of synchronized FIFOs.

Table 3 compares measured memory throughput in two different cases. The first row shows the read throughput of uncacheable memory. It is only 707Mbps, which is insufficient for 802.11 processing. The second row shows the performance of the smart-fetch technique. With smart-fetch, the memory throughput is a factor of 14 greater compared to the uncacheable case, and sufficient for supporting high-speed protocol processing. We note, however, that it is still slower than reading from normal cacheable memory without having to be consistent with DMA operations. This reduction is due to the overhead of additional cache-line invalidations.

Synchronized FIFO. To measure the overhead of the synchronized CBFIFO and CPFIFO implementations, we process ten thousand data inputs through the FIFOs first on one core, and then on two cores. We also vary the number of cycles to process each datum to change the ratio of synchronization time with processing time. When processing with two cores, we allocate the same computation to each core. Denote t_1 and t_2 as the completion times of processing on one core and two cores, respectively. We then define the overhead of a synchronized FIFO as $\frac{t_2 - t_1/2}{t_1/2}$.

Figure 8 shows the results of this experiment. The x-axis shows the total processing cycles required for each datum, and the y-axis shows the overhead of the syn-

chronized FIFO. We make following observations from these results. First, partitioning work across cores gives different overheads depending upon whether the cores are on the same die. Two cores on the same die share the same L2 cache, while cores on different dies are connected via a shared front-side bus. Thus, streaming data between functional blocks across cores on the same die has significantly less overhead than streaming between cores on different dies.

Second, the overhead decreases as the computation time per datum increases, as expected. When the computation per datum is very short, the communication overhead between cores dominates. The Intel CPU requires about 10 cycles to access its local L2 cache, and 100 cycles to access a remote cache. Therefore, when there are 40 cycles per datum, the overhead is at least $\frac{10}{20} = 50\%$ when two cores are on one die, and $\frac{100}{20} = 500\%$ when two cores are on different dies. The CPFIFO almost achieves this lower bound. When there is more computation required per datum, however, the data transfer can be overlapped with computation, enabling the overhead to be hidden. Finally, the CBFIFO generally has significantly higher overhead compared to the CPFIFO due to the additional synchronization overhead on the shared variable, which the CPFIFO avoids.

6 Case study: SoftWiFi

To demonstrate the use of Sora, we have developed a fully functional WiFi transceiver on the Sora platform called SoftWiFi. Our SoftWiFi stack supports all IEEE 802.11a/b/g modulations and can communicate seamlessly with commercial WiFi network cards.

Figure 9 illustrates the Sora SoftWiFi implementation. The MAC state machine (SM) is implemented as an ethread. Since 802.11 is a half-duplex 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 streamlined using a CPFIFO. An additional thread, *Snd_thread*, modulates the outgoing frames into waveform samples in the background. These modulated waveforms can be pre-stored in the RCB's memory to facilitate transmission. The *Completion_thread* monitors the *Rcv_buf* and notifies upper software layers of any correctly received frames. This thread also cleans up the *snd* and *rcv* buffers after they are used.

SoftWiFi implements the basic access mode of 802.11. The detailed MAC SM is shown in Figure 10. Normally, the SM is in the *Frame Detection* (FD) state. In that state, the RCB constantly writes samples into the *Rx_buf*. The SM continuously measures the average energy to determine whether the channel is clean or whether there is an incoming frame.

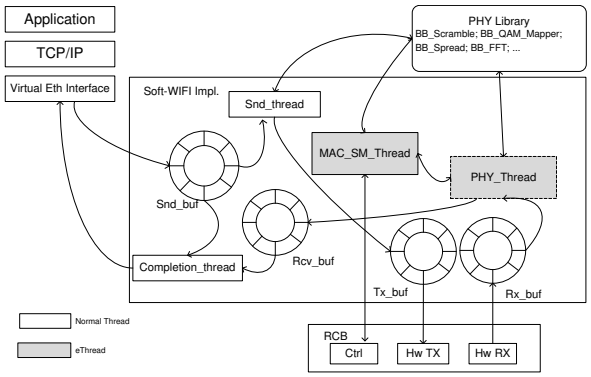

Figure 9: SoftWiFi implementation.

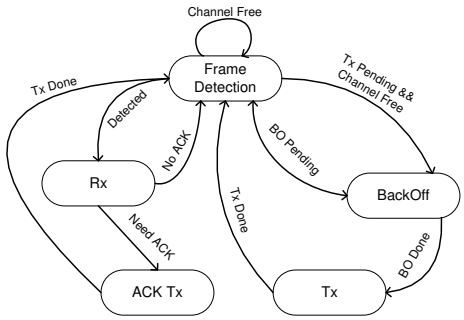

Figure 10: State machine of the SoftWiFi MAC.

The transmission of a frame follows the CSMA mechanism. When there is a pending frame, the SM first needs to check if the energy on the channel is low. If the channel is busy, the transmission should be deferred and a backoff timer started. Each time the channel becomes free, the SM checks if any backoff time remains. If the timer goes to zero, it transmits the frame.

SoftWiFi starts to receive a frame if it detects a high energy in the FD state. In 802.11, it takes three steps in the PHY layer to receive a frame. First, the PHY layer needs to synchronize to the frame, *i.e.*, find the starting point of the frame (timing synchronization) and the frequency offset and phase of the sample stream (carrier synchronization). Synchronization is usually done by correlating the incoming samples with a pre-defined preamble. Subsequently, the PHY layer needs to demodulate the PLCP (Physical Layer Convergence Protocol) header, which is always transmitted using a fixed low-rate modulation mode. The PLCP header contains the length of the frame as well as the modulation mode, possibly a higher rate, of the frame data that follows. Thus, only after successful reception of the PLCP header will the PHY layer know how to demodulate the remainder of the frame.

After successfully receiving a frame, the 802.11 MAC standard requires a station to transmit an ACK frame in a timely manner. For example, 802.11b requires that an

ACK frame be sent with a $10\mu s$ delay. However, this ACK requirement is quite difficult for an SDR implementation to achieve 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. For example, in 802.11 all data in an ACK frame is fixed except for the sender address of the corresponding data frame. Thus, in SoftWiFi, we can precalculate most of an ACK frame (19 bytes), and update only the address (10 bytes). Further, we can do it early in the processing, immediately after demodulating the MAC header, and without waiting for the end of a frame. We then prestore the waveform into the memory of the RCB. Thus, the time for ACK generation and transferring can overlap with the demodulation of the data frame. After the MAC SM demodulates the entire frame and validates the CRC32 checksum, it instructs the RCB to transmit the ACK, which has already been stored on the RCB. Thus, the latency for ACK transmission is very small.

In rare cases when the incoming data frame is quite small (*e.g.*, the frame contains only a MAC header and zero payload), then SoftWiFi cannot fully overlap ACK generation and the DMA transfer with demodulation to completely hide the latency. In this case, SoftWiFi may fail to send the ACK in time. We address this problem in SoftWiFi by maintaining a cache of previous ACKs in the RCB. With 802.11, all data frames from one node will have exactly the same ACK frame. Thus, we can use pre-allocated memory slots in the RCB to store ACK waveforms for different senders (we currently allocate 64 slots). Now, when demodulating a frame, if the ACK frame is already in the RCB cache, the MAC SM simply instructs the RCB to transmit the pre-cached ACK. With this scheme, SoftWiFi may be late on the first small frame from a sender, effectively dropping the packet from the sender's perspective. But retransmissions, and all subsequent transmissions, will find the appropriate ACK waveform already stored in the RCB cache.

We have implemented and tested the full 802.11a/g/b SoftWiFi tranceivers, which support DSSS (Direct Sequence Spreading: 1 and 2Mbps in 11b), CCK (Complementary Code Keying: 5.5 and 11Mbps in 11b), and OFDM (Orthogonal Frequency Division Multiplexing: 6, 9 and up to 54Mbps in 11a/g). It took one student about one month to develop and test 11b on Sora, and another student one and half months to code and test 11a/g; these efforts also include the time for implementing the corresponding algorithms in the PHY library.

7 Evaluations

In this section we evaluate the end-to-end application performance delivered by Sora. Our goals are to show that Sora interoperates seamlessly with commercial 802.11 devices, and that the Sora SoftWiFi implementation achieves equivalent performance. As a result, we show that Sora can process signals sufficiently fast to achieve full channel utilization, and that it can satisfy all timing requirements of the 802.11 standards with a software implementation on a GPP. We also characterize the CPU utilization of the software processing. In the following, we sometimes use the label 11a/g to present data for both 11a/g, since 11a and 11g have exactly the same OFDM PHY specification.

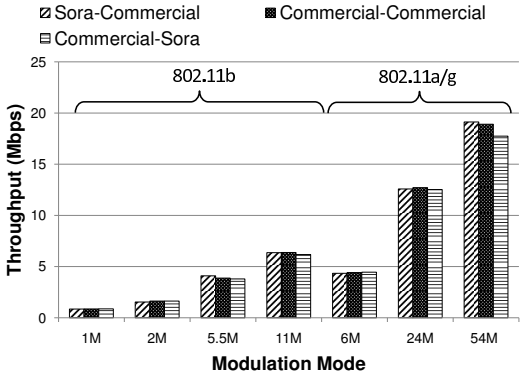

Figure 11: Throughput of Sora when communicating with a commercial WiFi card. *Sora–Commercial* presents the transmission throughput when a Sora node sends data. *Commercial–Sora* presents the throughput when a Sora node receives data. *Commercial–Commercial* presents the throughput when a commercial NIC communicates with another commercial NIC.

7.1 Experimental setup

The experimental setup consists of two high-end Dell XPS PCs (Intel Core 2 Quad 2.66GHz CPU, 4GB DDR2 400MHz SDRAM, and two PCIe-16x slots) and two laptops, all running Window XP. Each Dell PC equips a Sora radio control board (RCB) with an 802.11 RF board (Section 5) and runs Sora and the SoftWiFi implementation. Each CPU core has 32KB instruction and 32KB data L1 caches and a 2MB L2 cache. The Dell laptops use commercial WiFi NICs. We have used several different WiFi NICs in our experiments, including Netgear, Cisco and Intel devices. All give similar results. Thus, we present results just for the Netgear WAG511 device (based on the Atheros AR5212 chipset).

7.2 Throughput

Figure 11 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 11 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.

7.3 CPU Utilization

What is the processing cost of onloading all digital signal processing into software on the host? Figure 12 shows the CPU utilization of a Sora SoftWiFi node to support modulation/demodulation at the corresponding rate. We normalize the utilization to the processing capability of one core. For receiving, higher modulation rates require higher CPU utilization due to the increased computational complexity of demodulating the higher rates. We can see that one core of a contemporary multicore CPU can comfortably support all 11b modulation modes. With the 11Mbps rate, Sora SoftWiFi requires roughly 70% of the computational power of one core for real-time SDR processing. However, 802.11a/g PHY processing is more complex than 11b and may require two cores for receive processing. In our software implementation, the Viterbi decoder in 11a/g is the most computationally-intensive component. It alone requires more than 1.4 Gcycles/s at modulation rates higher than 24Mbps (Table 1). Therefore, it is natural to partition the receive pipeline across two cores, with the Viterbi decoder on one core and the remainder on another. With the parallelism enabled by this streamline processing,

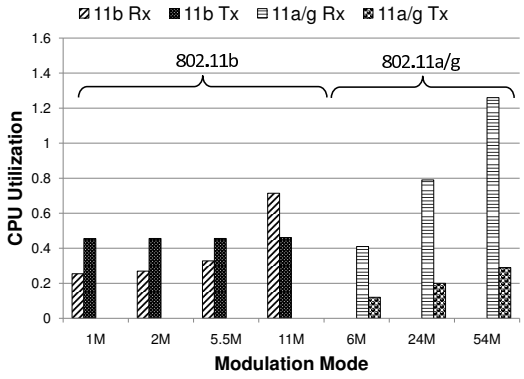

Figure 12: CPU Utilization of Sora.

we reduce the delay to process one 11a/g symbol from $4.8\mu s$ to $3.9\mu s$, meeting the requirement of the standard (*i.e.* $4\mu s$) for 54Mbps. Note that the CPU utilization is not completely linear with the modulation rates in 11b because the 5.5/11Mbps rates use a different modulation scheme than with 1/2Mbps.

The CPU utilization for transmission, however, is generally lower than the receiving case. Note that the utilization is constant for all 11b rates. Since the transmission part of 11b can be optimized effectively with LUTs, for different rates we just use different LUTs. In 11a/g, since all samples need to pass an IFFT, the computation requirements increase as the rate increases.

7.4 Detailed processing costs

The results in Figure 12 presented the overall CPU utilization for a Sora SoftWiFi receiving node. As discussed in Section 6, a complete receiver has a number of stages: frame detection, frame synchronization, and demodulators for both the PLCP header and its data depending on the modulation mode. How does CPU utilization partition across these stages? Figure 13 shows the computational cost for each component for receiving a 1400-byte UDP packet in each modulation mode; again, we show only three representative modulation rates for 11a/g. Frame detection (FD) has the lowest utilization (11% of a 2.66GHz core for 11b and only 3.2% for 11a/g) and is constant across all modulation modes in each standard. Note that frame detection needs to execute even if there is no communication since a frame may arrive at any time. When Sora detects a frame, it uses 29% of a core to synchronize to the start of a frame (SYNC) for 11b, and it uses 20% of a core to synchronize to an 11a/g frame. Then Sora can demodulate the PLCP header, which is always transmitted using the lowest modulation rate. It requires slightly less (27.5%) computation overhead than synchronization for 11b; but it needs much more computation (44%) for 11a. Demodulation of the data (DATA) at the higher rates is the most computationally expensive step in a receiver. It re-

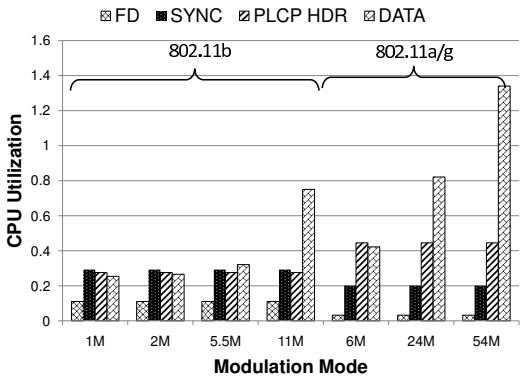

Figure 13: Detailed processing costs in WiFi PHY.

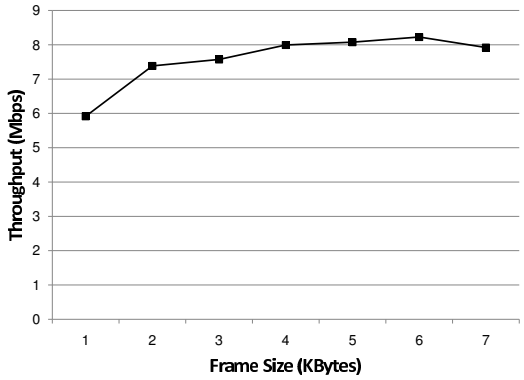

Figure 14: Throughput with Jumbo Frames between two Sora SoftWiFi nodes.

quires 75% of a core at 11Mbps for 11b, and the utilization reaches exceeds one core (134%) for processing at 54Mbps in 11a/g. This result indicates that we need to streamline the processing to at least two cores to support this modulation.

8 Extensions

The flexibility of Sora allows us to develop interesting extensions to current WiFi protocol.

8.1 Jumbo Frames

If the channel conditions are good, transmitting data with larger frames can reduce the overhead of MAC/-PHY headers, preambles and the per frame ACK. However, the maximal frame size of 802.11 is fixed at 2304 bytes. With simple modifications (changes in a few lines), SoftWiFi can transmit and receive jumbo frames with up to 32KB. Figure 14 shows the throughput of sending UDP packets between two Sora SoftWiFi nodes using the jumbo frame optimization across a range of frame sizes (with 11b using the 11Mbps modulation mode). When we increase the frame size from 1KB to 6KB, the end-to-end throughput increase 39% from 5.9Mbps to 8.2Mbps. When we further increase the frame size to 7KB, however, the throughput drops because the frame error rate also increases with the size. So, at some point, the increasing error will offset the gain of reducing the overhead. Note that our default commercial NIC rejects frames larger than 2304 bytes, even if those frames can be successfully demodulated.

In this experiment, we place the antennas close to each other, clearly a best-case scenario. Our goal, though, is not to argue that jumbo frames for 802.11 are necessarily a compelling optimization. Rather, we want to demonstrate that the full programmability offered by Sora makes it both possible and straightforward to explore such "what if" questions on a GPP SDR platform.

	10ms	50ms	100ms
$\epsilon/\sigma(\mu s)$	0.85/0.5	0.96/0.54	0.98/0.46
Outlier	0.5%	0.4%	0.4%

Table 4: Timing error of Sora in TDMA.

8.2 TDMA MAC

To evaluate the ability of Sora to precisely control the transmission time of a frame, we implemented a simple TDMA MAC that schedules a frame transmission at a predefined time interval. The MAC state machine (SM) runs in an ethread, and it continuously queries a timer to check if the pre-defined amount of time has elapsed. If so, the MAC SM will instruct the RCB to send out a frame. The modification is simple and straightforward with about 20 lines of additional code.

Since our RCB can indicate to SoftWiFi when the transmission completes, and we know the exact size of the frame, we can calculate the exact time when the frame transmits. Table 4 summarizes the results with various scheduling intervals under a heavy load, where we copy files on the local disk, download files from a nearby server, and playback a HD video simultaneously. In the Table, ϵ presents the average error and σ presents the standard deviation of the error. The average error is less than $1\mu s$, which is sufficient for most wireless protocols. We also list outliers, which we define as packet transmissions that occur later than $2\mu s$ from the pre-defined schedule. Previous work has also implemented TDMA MACs on a commodity WiFi NIC [20], but their software architecture results in a timing error of near $100\mu s$.

8.3 Soft Spectrum Analyzer.

It is also easy for Sora to expose all PHY layer information to applications. One application we have found useful is a software spectrum analyzer for WiFi. We have implemented such a simple spectrum analyzer that can graphically display the waveform and modulation points

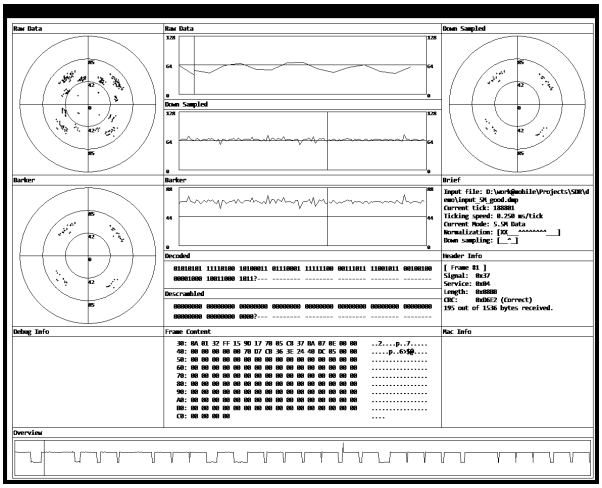

Figure 15: Software Spectrum Analyzer built on Sora.

in a constellation graph, as well as the demodulated results, as shown in Figure 15. Commercial spectrum analyzers may have similar functionality and wider sensing spectrum band, but they are also more expensive.

9 Related Work

In this section we discuss various efforts to implement software defined radio functionality and platforms.

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 [4] 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 [20]. 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 non-standard 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 on the device.

GNU Radio is a popular software toolkit for building software radios using general purpose computing platforms [1]. It is derived from an earlier system called SpectrumWare [22]. GNU Radio consists of a software library and a hardware platform. Developers implement software radios by composing modular pre-compiled 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 GNU Radio platforms can only sustain low-speed wireless communication due to both the hardware constraints as well as software processing [21]. As a consequence, users must sacrifice radio performance for its flexibility.

The WARP hardware platform provides a flexible and high-performance software defined radio platform [6]. Based on Xilinx FPGAs and PowerPC cores, WARP allows full control over the PHY and MAC layers and supports customized modulations up to 36 Mbps. 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 [18]. 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), which has a 1.4GHz Pentium M processor. Therefore, it allows some communication systems to be implemented completely in software on CPH. They have demonstrated some GNU Radio applications on KUAR. 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 [17]. 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.

10 Conclusions

This paper presents 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 radio implementation of the 802.11a/b/g protocols. We are planning and implementing additional software radios, such as 3GPP LTE (Long Term Evolution), W-CDMA, and WiMax using the Sora platform. We have started the implementation of 3GPP LTE in cooperation with Beijing University of Posts and Telecommunications, China, and we confirm the programming effort is greatly reduced with Sora. For example, it has taken one student only two weeks to develop the transmission half of LTE PUSCH(Physical Uplink Shared Channel), which can be a multi-month task on a traditional FPGA platform.

The flexibility provided by Sora makes it a convenient platform for experimenting with novel wireless protocols, such as ANC [16] or PPR [15]. Further, being able to utilize multiple cores, Sora can scale to support even more complex PHY algorithms, such as MIMO or SIC (Successive Interference Cancellation) [23].

More broadly, we plan to make Sora available to the wireless networking research community. Currently, we are collaborating with Xi'an Jiao Tong University, China, to design a new MIMO RF module that supports eight channels. We are planning moderate production of the Sora RCB and RF modules for use by other researchers. The estimated cost for Sora hardware is about $2,000 per set (RCB + one RF front-end). We also plan to release the Sora software to the wireless network research community. Our hope is that Sora can substantially contribute to the adoption of SDR for wireless networking experimentation and innovation.

Acknowledgements

The authors would like to thank Xiongfei Cai, Ningyi Xu and Zenlin Xia in the Platform and Devices Center 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.

References

[1] Gnu radio. *http://www.gnu.org/software/gnuradio/*.

[2] HostAP. *http://hostap.epitest.fi/*.

[3] Madwifi. *http://sourceforge.net/projects/madwifi*.

[4] Rt2x00. *http://rt2x00.serialmonkey.com*.

[5] Small form factor sdr development platform. *http://www.xilinx.com/products/devkits/SFF-SDR-DP.htm*.

[6] WARP: Wireless open access research platform. *http://warp.rice.edu/trac*.

[7] *ANSI/IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*. IEEE Press, 1999.

[8] *PCI Express Base 2.0 specification*. PCI-SIG, 2007.

[9] A. Agarwal and M. Levy. Thousand-core chips: the kill rule for multi-core. In *Proceedings of the 44th Annual Conference on Design Automations*, 2007.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI*, 2008.

[11] M. Cummings and S. Haruyama. FPGA in the software radio. *IEEE Communications Magazine*, 1999.

[12] J. V. de Vegte. *Fundamental of Digital Signal Processing*. Cambridge University Press, 2005.

[13] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *3rd Workshop on Application Specific Processors*, 2004.

[14] A. Goldsmith. *Wireless Communication*. Cambridge University Press, 2005.

[15] K. Jamieson and H. Balakrishnan. Ppr: Partial packet recovery for wireless networks. In *Proceedings of ACM SIGCOMM 2007*, April 2007.

[16] S. Katti, S. Gollakota, and D. Katabi. Embracing wireless interference: analog network coding. In *Proceedings of ACM SIGCOMM 2007*, pages 397–408. ACM Press, 2007.

[17] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, and T. Mudge. Soda: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[18] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Patty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah. Kuar: A flexible software-defined radio development platform. In *DySpan*, 2007.

[19] J. Neel, P. Robert, and J. Reed. 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.

[20] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. Softmac - flexible wireless research platform. In *HotNets 05*, 2005.

[21] T. Schmid, O. Sekkat, and M. B. Srivastava. An experimental study of network performance impact of increased latency in software defined radios. In *WiNETCH07*, 2007.

[22] D. L. Tennenhouse and V. G. Bose. Spectrumware-a software-oriented approach to wireless signal processing. In *MobiCom 95*, 1995.

[23] S. Verdu. *Multiuser Detection*. Cambridge University Press, 1998.

Appendix A: SIMD example for FIR Filter

In this appendix, we show a small example of how to use SSE instructions to optimize the implementation of a FIR (Finite Impulse Response) filter in Sora. FIR filters are widely used in various PHY layers. An n-tap FIR filter is defined as

$$y[t] = \sum_{k=0}^{n-1} c_k \cdot x[t-k],$$

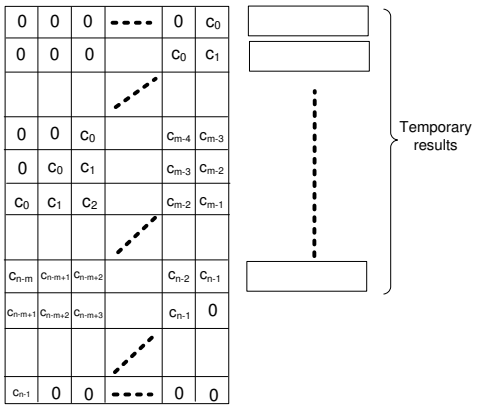

Figure 16: Memory layout of the FIR coefficients.

where $x[.]$ are the input samples, $y[.]$ are the output samples, and c_k are the filter coefficients. With SIMD instructions, we can process multiple samples at the same time. For example, Intel SSE supports a 128-bit packed-vector and each FIR sample takes 16 bits. Therefore, we can perform $m = 8$ calculations simultaneously. To facilitate SSE processing, the data layout in memory should be carefully designed. Figure 16 shows the memory layout of the FIR coefficients. Each row forms a packed-vector containing m components for SIMD operations. The coefficient vector of the FIR filter is replicated in each column in a zig-zag layout. Thus, the total number of rows is $(n + m - 1)$. There are also n temporary variables containing the accumulated sum up to each FIR tap for each sample.

Figure 17 shows the example code. It takes an array of input samples, a coefficient array, and outputs the filtered samples in an output sample buffer. The input contains two separate sample streams, with the even and odd indexed samples representing the *I* and *Q* samples, respectively. The coefficient array is arranged similarly to Figure 16, but with two sets of FIR coefficients for *I* and *Q* samples, respectively.

Each iteration, four *I* and four *Q* samples are loaded into an SSE register. It multiplies the data in each row and adds the result to the corresponding temporal accumulative sum variable (lines 59–68). A result is output when all taps are calculated for the input samples (lines 18–57). When the input sample stream is long, there are nm samples in the pipeline and m outputs are generated in each iteration. Note that the output samples may not be in the same order as the input — some algorithms do not always require the output to have exactly the same order as the input. A few shuffle instructions can be added to place the output samples in original order if needed.

```
1  int FirSSE ( PSAMPLE  pSrc,
2               PSAMPLE  pOutput,
3               int nSize, // number of complex samples
4               PSHORT    pCoff, // filter coeffs
5               int iTaps, // the highest index of tap (n-1)
6               PSAMPLE pTempBuf, // for temp value store
7             )
8  {
9    _asm {
10     mov esi, pSrc;
11     mov ecx, nSize;
12     mov ebx, pOutput;
13 outerloop:
14     mov edx, pCoff;
15     mov edi, pTempBuf;
16
17     ;// load samples 4-I and 4-Q
18     movdqa xmm0, [esi];
19
20     ; // result_0
21     movdqa xmm4, xmm0;
22     pmullw xmm4, [edx];
23     paddsw xmm4, [edi];
24     ; // result_1
25     movdqa xmm5, xmm0;
26     pmullw xmm5, [edx + 16];
27     paddsw xmm5, [edi + 16];
28     ; // result_2
29     movdqa xmm6, xmm0;
30     pmullw xmm6, [edx + 32];
31     paddsw xmm6, [edi + 32];
32     ; // result_3
33     movdqa xmm7, xmm0;
34     pmullw xmm7, [edx + 48];
35     paddsw xmm7, [edi + 48];
36
37     ; // xmm4, xmm5, xmm6, xmm7 contains output
38     ; // perform shuffle and horizontal additions
39     movdqa xmm1, xmm4;
40     punpckldq xmm1, xmm6;
41     punpckhdq xmm4, xmm6;
42     paddsw xmm4, xmm1;
43
44     movdqa xmm1, xmm5;
45     punpckldq xmm1, xmm7;
46     punpckhdq xmm5, xmm7;
47     paddsw xmm5, xmm1;
48
49     movdqa xmm1, xmm4;
50     punpckldq xmm1, xmm5;
51     punpckhdq xmm4, xmm5;
52     paddsw xmm4, xmm1;
53
54     ; // output
55     ; // additional instructions may be added to
56     ; // adjust the sample orders
57     movdqa [ebx], xmm4;
58
59     ; // update temp buffers
60     mov eax, iTaps;
61 innerloop:
62     movdqa xmm1, xmm0;
63     pmullw xmm1, [edx + 64];
64     paddsw xmm1, [edi + 64];
65     movdqa [edi], xmm1;
66
67     add edx, 16;
68     add edi, 16;
69     dec eax;
70     jnz innerloop;
71
72     ;// advance to next sample group
73     add esi, 16;
74     add ebx, 16;
75     sub ecx, 4;
76     jg outerloop;
77   }
78 }
```

Figure 17: Pseudo-code of SSE optimized FIR Filter.