Enabling MAC Protocol Implementations on Software-Defined Radios

George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, Peter Steenkiste
Carnegie Mellon University

Abstract

Over the past few years a range of new Media Access Control (MAC) protocols have been proposed for wireless networks. This research has been driven by the observation that a single one-size-fits-all MAC protocol cannot meet the needs of diverse wireless deployments and applications. Unfortunately, most MAC functionality has traditionally been implemented on the wireless card for performance reasons, thus, limiting the opportunities for MAC customization. Software-defined radios (SDRs) promise unprecedented flexibility, but their architecture has proven to be a challenge for MAC protocols.

In this paper, we identify a minimum set of core MAC functions that must be implemented close to the radio in a high-latency SDR architecture to enable high performance and efficient MAC implementations. These functions include: precise scheduling in time, carrier sense, backoff, dependent packets, packet recognition, fine-grained radio control, and access to physical layer information. While we focus on an architecture where the bus latency exceeds common MAC interaction times (tens to hundreds of microseconds), other SDR architectures with lower latencies can also benefit from implementing a subset of these functions closer to the radio. We also define an API applicable to all SDR architectures that allows the host to control these functions, providing the necessary flexibility to implement a diverse range of MAC protocols. We show the effectiveness of our *split-functionality* approach through an implementation on the GNU Radio and USRP platforms. Our evaluation based on microbenchmarks and end-to-end network measurements, shows that our design can simultaneously achieve high flexibility and high performance.

1 Introduction

Over the past few years, a range of new Media Access Control (MAC) protocols have been proposed for use in wireless networks. Much of this increased activity has been driven by the observation that a single one-size-fits-all MAC protocol cannot meet the needs of diverse wireless deployments and applications and, thus, MAC protocols need to be specialized (e.g. for use on long-distance links, mesh networks). Unfortunately, the development and deployment of new MAC designs has been slow due to the limited programmability of traditional wireless network interface hardware. The reason is that key MAC functions are implemented on the network interface card (NIC) for performance reasons, which often uses proprietary software and custom hardware, making the MAC hard, if even possible, to modify.

Software-defined radios (SDRs) have been proposed as an attractive alternative. SDRs provide simple hardware that translates signals between the RF and the digital domains. SDRs implement most of the network interface functionality (e.g., the physical layer and link layer) in software and, as a result, they make it feasible for developers to modify this functionality. SDR architectures [19, 6, 17, 20, 9] typically distribute processing of the digitized signals across several processing units – including FPGAs and CPUs located on the SDR device, and the CPU of the host. The platforms differ in the precise nature of the processing units that are provided, how those units are connected, and how computation is distributed across them.

Unfortunately, the high degree of flexibility offered by SDRs does not automatically lead to flexibility in the MAC implementation. The reason is that, in the SDR architecture we are addressing, the use of multiple heterogeneous processing units with interconnecting buses, introduces large delays and jitter into the processing path of packets. Processing, queuing, and bus transfer delays can easily add up to hundreds of microseconds [14]. Unfortunately, the delay limits how quickly the MAC can respond to incoming packets or changes in channel conditions, and the jitter prevents precise control over the timing of packet transmissions. These restrictions severely reduce the performance of many MAC protocols.

This paper presents a set of techniques that makes it possible to implement diverse, high performance MAC protocols that are easy to modify and customize from the host. The key idea is a novel way of splitting core MAC functionality between the host processing unit and pro-

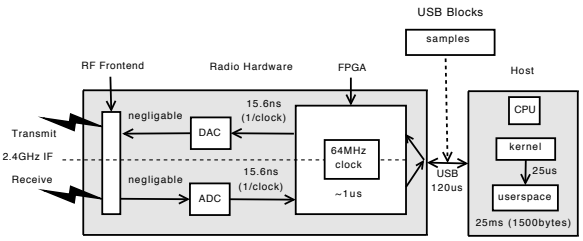

Figure 1: Generic SDR Architecture

cessing units on the hardware (e.g., FPGA). The paper makes the following contributions:

- We identify a set of core MAC functions that must be implemented close to the radio for performance and efficiency reasons.
- We define a *split-functionality* architecture that allows the functions to be implemented near the radio hardware, while maintaining control on the host CPU through an API.
- We present an implementation of our architecture using the GNU Radio [6] and USRP [17] SDR platform. We also use our implementation to characterize the performance-flexibility tradeoffs for key MAC features. For example, our results show three orders of magnitude greater precision for the scheduling of packets and carrier sense, along with a high level of accuracy in fast packet detection.
- Finally, we use our implementation for an end-to-end evaluation of the split-functionality architecture. We show how the system can support diverse high-performance MAC implementations by implementing 802.11-like and Bluetooth-like protocols for experimentation over the air.

The rest of the paper is organized as follows. We discuss current radio architecture and its impact on MAC protocol development in Section 2. In Sections 3 and 4, we explore the core MAC requirements and introduce our *split-functionality* architecture. Section 5 provides details for each component implementation with evaluation results. Finally, we present end-to-end evaluation results, related work, and a summary of our results in Sections 6 through 8.

2 MAC Implementation Choices

A number of different software-defined radio architectures have been developed. One common architecture is shown in Figure 1. The frontend is responsible for converting the signal between the RF domain and an intermediate frequency, and the A/D and D/A components convert the signal between the analog and the digital domain. Physical and higher layer processing of the digitized signal are executed on one or more processing units. Typically, there is at least an FPGA or DSP close to the frontend. The frontend, D/A, A/D, and FPGA are usually placed on a network card that is connected to the host CPU by a standard bus (e.g., USB).

The distribution of functionality across the processing units significantly impacts the radio's performance, flexibility, and ease of reprogramming. To achieve a high level of flexibility and reprogramming, the majority of processing (i.e., modulation) can be placed on the host CPU where the functionality is easy to modify. We refer to this architecture as *host-PHY*. This architecture is exemplified by GNU Radio [6] and the USRP [17], which place the majority of functionality in userspace, shown in Figure 1. For greater performance, processing can be implemented in the radio hardware on the FPGA or DSP. We refer to this architecture as *NIC-PHY*. The WARP platform [20] implements this architecture, placing the PHY and MAC layers on the radio hardware for performance reasons. It is fairly straightforward however, to parameterize PHY layers (e.g. to control the frequency band and coding an modulation options). Thus, it is possible control many aspects of the PHY layer from the host, no matter where it is implemented.

Unfortunately, MAC protocols are less structured and SDRs have fallen short in providing high-performance flexible MAC implementation. The MAC is either implemented near the radio hardware for performance, or near the host for flexibility. We propose a novel split of MAC functionality across the processing units in a *host-PHY* architecture such that we can achieve a high level of performance, while maintaining flexibility at both the MAC and PHY layers. This is especially significant in a *host-PHY* architecture, which has been considered incapable of supporting even core MAC protocol functions (e.g., carrier sense) due to the large processing delays inherent to the architecture [14, 18]. In addition, our design can enable many cross-layer optimizations, such as those proposed between the MAC and PHY layers [5, 8, 7]. Such optimizations have used the *host-PHY* architecture for easy PHY modifications, but given the lack of MAC support, they typically "fake" the MAC layer (e.g., by combining the SDR with a commodity 802.11 NIC to do the MAC processing [5]) or omit it all together [7, 8]. Although our work focuses on a *host-PHY* architecture, several of the components we will present can be applied to a *NIC-PHY* architecture.

In the next section, we explore delay and jitter measurements in the *host-PHY* architecture, which are the major limiting factor on performance of MAC implementations. The measurements are important in understanding the proper split of MAC functionality across the heterogeneous processing units of an SDR.

	Avg	SDev	Min	Max
User–>Kernel (μs)	24	10	22	213
Kernel–>User (μs)	27	89	13	7000
4096 Kernel<–>FPGA (μs)	291	62	204	360
512 Kernel<–>FPGA (μs)	148	35	90	193
GNU Radio<–>FPGA (μs)	612	789	289	9000

Table 1: Kernel level delay measurements.

2.1 Delay Measurements

Schmid et al [14] present delay measurement for SDRs and their impact on MAC functionality in a *host-PHY* architecture. However, they focus on user-level measurements, largely ignoring precise measurement of delays between the kernel and userspace, and kernel and the radio hardware. Such measurements are important, since they can provide insight into whether implementing MAC functions in the kernel is sufficient to overcome the performance problems associated with user level implementations. To obtain precise user and kernel-level measurements, we modified the Linux kernel's USB Request Block (*URB*) and USB Device Filesystem URB (*USBDEVFS_URB*) to include nanosecond precision timestamps taken at various times in the transmission and receive process. All user level timestamps are taken in user space right before or after a URB is submitted (write) or returned (read). At the kernel level, the measurement is taken at the last point in the kernel's USB driver before the DMA write request is generated, or after a DMA read request interrupts the driver. This is as close to the bus transfer timing as possible.

We measured the round trip time between GNU Radio (in user space) and the FPGA using a ping command on a control channel that we implement (Section 4.2). Using the measurements described above, we are also able to identify the sources of the delay by calculating the user to kernel space delay, kernel to user space delay, and round trip time between the kernel and FPGA based on ping. We ran the user process at the highest priority to minimize scheduling delay. We used the default 4096 byte USB transfer block size for all experiments, and then perform an additional kernel to FPGA RTT experiment using a 512 byte transfer block size, the minimum possible, in an attempt to minimize queuing delay.

The results presented in Table 1 are averaged over 1000 experiments. Focusing on the average times, we see the cost of a GNU Radio ping is dominated by the kernel-FPGA roundtrip time (291 out of 612 μs). The user-kernel and kernel-user times are relatively modest (24 and 27 μs). The remaining time (270 μs) is spent in the GNU Radio chain. The high latency of the kernel-FPGA roundtrip time is somewhat surprising, given that the effective measured rate of the USB with the USRP is 32MB/s. The difference between the latencies for 4KB and 512B shed some light on this. The difference in latency is only a factor of two, suggesting that the set up cost for transfers contributes significantly to the delay. The kernel-FPGA time also includes the time it takes for the data to pass through the USRP USB FX2 controller buffers, and to be copied into the FPGA for parsing. The time taken for the data to pass through the USRP USB FX2 controller buffers and copied into the FPGA for parsing also contributes to the kernel-FPGA RTT.

The standard deviations and the min/max values paint a different picture. The user-to-kernel and kernel-FPGA times fall in a fairly narrow range, so they only contribute a limited amount of jitter. The kernel-to-user times however have a very high standard deviation, which results in a high standard deviation for the GNU Radio ping delays. This is clearly the result of process scheduling.

2.2 MAC Design Space

As discussed briefly in Section 2, the processing units in the above SDR architecture have very different properties. Focusing on Figure 1, the host CPU is easy to program and is readily accessible to users and developers. However, the path between the host CPU and the radio front end has both high delay and jitter, as shown by the measurements presented in Section 2.1. The round trip times between the device driver on the host and the FPGA is about 300 μs for 4KB of data, with relatively modest jitter. The roundtrip from GNU Radio is about double, but with significantly more jitter. As a result, a host-based MAC protocol (be it in user space or in the kernel) will not be able to precisely control packet timing, or implement small, precise inter-frame spacings, which will hurt the performance of many MAC protocols. We conclude that, *time critical radio or MAC functions should not be placed on the host CPU*.

Processing close to the radio performed by a FPGA or CPU on the NIC has the opposite properties. It has a low latency path to the frontend (see USRP latencies in Figure 1), making it attractive for delay sensitive functions. Unfortunately, code running on the radio hardware is much harder to change because it is often hardware-specific and requires a more complex development environment. Moreover, history shows that vendors do not provide open access to their NICs, even if they are programmable. Access to the processors on the NIC is restricted to its manufacturer and possibly large customers who can, under license, customize the NIC code. This is of course not a problem for research groups using research platforms, which is why many researchers are moving to software radios, but it is an important consideration for widespread deployment. We conclude that *in order to be widely applicable, the control of flexible MAC implementations should reside on the host.*

Interesting enough, the SDR NIC architecture in Figure 1 is not unlike the architecture of traditional NICs (e.g., 802.11 cards). Today's commodity NICs use analog hardware to perform physical layer processing, but they typically also have a CPU, FPGA, or custom processor. These commodity devices exhibit the same tradeoffs we identified above for software radios: the delay between the processing on the host and the (analog) frontend is substantially higher and less predictable than between the NIC processor and the front end.

Experience with commercial 802.11 cards supports the conclusions we highlighted above. First, time sensitive MAC functions such as sending ACKs are always performed on the NIC, and only functions that are not delay sensitive such as access point association are handled by the host processor. Moreover, although most of the MAC functionality on the NIC is implemented in software, it can only be modified by a small number of vendors (i.e. in practice the NIC is a black box). Researchers have had some success in using commodity cards for MAC research by moving specific MAC functions to the host [13, 16, 10, 15], but the results are often unsatisfactory. The host can only take control over certain functions (e.g. interframe spacings must be longer than 60 microseconds), precision is limited (e.g. cannot eliminate all effects of jitter), and the host implementation is inefficient (as a result of polling) and is susceptible to host loads.

The different properties of the host and NIC processing units means that the placement of MAC functionality will fundamentally affect four key MAC performance metrics, including network performance, flexibility in MAC implementation and runtime control, and ease of development. Unfortunately, as discussed above, these performance goals are in conflict with each other and achieving the highest level for each is not possible. In this paper, we present a split-functionality architecture that implements key MAC functions on the radio hardware, but provides full control to the host. This allows us to simultaneously score very high on all four metrics, and it also allows developers and users to make tradeoffs across the metrics. While developers will always have to make tradeoffs, the negatives associated with specific design choices are significantly reduced in our design. Note that this does not imply that our design can support any arbitrary or even all existing MAC designs. However, we believe that it is capable of supporting most of the critical features of modern MAC designs.

The focus of the paper is on SDR platforms because they provide maximal flexibility in key research areas such as cross-layer MAC and PHY optimization (e.g., [5, 7, 8]). Our evaluation is based on a platform that uses the host-PHY architecture, but is not critical. Even in NIC-PHY architectures that have good support for the MAC on the NIC (e.g., in the form of a general-purpose CPU), it is important to maintain control over the MAC and PHY on the host to ensure easy customization. As a result, the techniques we propose can be useful across the entire spectrum of NIC designs.

3 Core MAC Functions

An ideal wireless protocol platform should support the implementation of well-known MAC protocols as well as novel MAC research designs. A study of current wireless protocols, including WiFi (both Distributed and Point Coordination Function), Zigbee, Bluetooth, and various research protocols shows that they are based on a common, core set of techniques such as contention-based access (CSMA), TDMA, CDMA, and polling. In this section, we identify key core functions that a platform must implement efficiently in order to support a wide range of MAC protocols.

Precise Scheduling in Time: TDMA-based protocols require precise scheduling to ensure that transmissions occur during time slots. Imprecise timing can be tolerated by using long guard periods; however, this degrades performance. Surprisingly, modern contention-based protocols also require precise scheduling to implement inter-frame spacing (i.e. DIFS, SIFS, PIFS), contention windows, back-off periods, etc.

Carrier Sense: Contention-based protocols often use carrier sense to detect other transmissions. Carrier sense may use simple power detection (e.g., using signal strength) or may use actual bit decoding. Network interfaces need to transmit shortly after the channel is detected to be idle. Additional delay increases both the frequency of collision and also the minimum packet size required by the network.

Backoff: When a transmission fails in a contention-based protocol, a backoff mechanism is used to reschedule the transmission under the assumption that the loss was caused by a collision. Backoff is related to precise scheduling, but focuses more closely on fast-rescheduling of a transmission without the full packet transmission process (e.g., modulation).

Fast Packet Recognition: Many MAC performance optimizations could use the ability to quickly detect an incoming packet and identify that it is relevant to the local node in a timely and accurate manner. For example, detecting and identifying an incoming packet before the demodulation procedure can reduce resource use on the processing units and on the bus.

Dependent Packets: Dependent packets are explicit responses to received packets. A typical example is control packets that are associated with data packets, for example for error control (e.g., ACKs) or for improved

channel access (e.g., RTS/CTS). Network interfaces need to generate these packets quickly and transmit them with precise time scheduling relative to the previous packet.

Fine-grained Radio Control: Frequency-hopping spread spectrum protocols such as Bluetooth and the recently proposed MAXchop algorithm [11] require fine-grained radio control to rapidly change channels according to a pseudo-random sequence. Similarly, recent designs [1] for minimizing interference require the ability to control transmission power on a per-packet basis.

Access to physical layer information: Many MAC protocol optimizations could benefit from access to radio-level packet information. Examples include using a received signal strength indicator (RSSI) to improve access point handoff decisions and using information on the confidence of each decoded bit to implement partial packet recovery [7].

3.1 Implications

While it is difficult to argue that this (or any) list of core functions is the correct one and is complete, we believe that it is sufficient to implement a broad range of interesting MAC protocols. To provide some degree of confidence in this statement, we describe our implementation of an 802.11-like CSMA protocol and a Bluetooth-like TDMA protocol using our framework in Section 6. As such, this is a reasonable first "toolbox" that MAC protocol developers can extend over time.

4 Split Functionality Architecture

As discussed in Section 2, implementing flexible high-performance MAC protocols is challenging because the high delays and jitter between the host CPU and frontend affects the performance of the core MAC functions described in the previous section. For example, most protocols need either precise scheduling in time or dependent packets. However, the delays inherent in a host MAC implementation in the given SDR architecture would make these functions inefficient or ineffective. In this section, we first review the requirements associated with the core MAC functions identified above, and we then present an architecture that allows us to support high performance MACs while maintaining host control.

4.1 Core Requirements

Implementing the core MAC functions from Section 3 raises three challenges.

Bus delay: The delay introduced by transmission of data over the bus can be constant and predictable, depending on the technology. A constant delay is relatively easy to accommodate in supporting *precision scheduling*, as discussed in Section 5.1. However, the bus delay does impact the performance of *carrier sense*, *dependent packets*, and *fast packet recognition*. The effect of bus latency on performance for SDR NICs is discussed in previous work [14].

Queuing delay: The delay introduced by queues may be smaller than the bus transmission delay but has significant jitter, which makes precision scheduling difficult, if not impossible. The jitter can modify the inter-packet spacing through compression or dispersion as the data is processed in the host and at the ends of the bus. In Section 5.1.2, we present measurements that show that this compression can be so significant in the given architecture that spacing transmissions by under 1ms cannot be achieved reliably using host-CPU based scheduling.

Stream-based architecture of SDRs: The frontend operates on streams of samples, which can make *fine-grained radio control* and *access to physical layer information* from the host ineffective. The reason is that it adds complexity to the interaction between a MAC layer executing on a host CPU (or NIC CPU) and the radio frontend since it is difficult to associate control information or radio information with particular groups of samples (e.g., those belonging to a packet). This problem consists of two components: (1) how to propagate information within the software environment that performs physical and MAC layer processing, and (2) how to propagate the information between the host and the frontend, across the bus and SDR hardware. This first issue is being addressed in the GNU Radio design with the introduction of m-blocks [2], which is briefly discussed in Section 7, but we must address the second issue.

4.2 Overcoming the Limitations

We now present an architecture that overcomes the above limitations. The goal is to allow as much of the protocol to execute on the host as possible to achieve the flexibility and ease of development goals, both of which are important to a wireless platform for protocol development, as identified in Section 2. However, we must ensure that the high latency and jitter between the host and radio frontend does not result in poor performance and limited control, the other two criteria in Section 2. This is done by introducing two architectural features, **per-block meta-data** and a **control channel**, shown in Figure 2. The novelty is not in the two new architectural features, but in how we use them to implement the core MAC functions (Section 3) in such a way that we maintain flexibility, while increasing performance (Section 5). We first discuss both features in more detail.

Per block meta-data: Enabling the association of information with a packet is crucial to the support of nearly

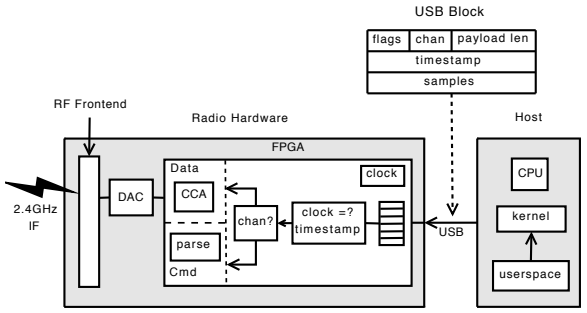

Figure 2: Split SDR architecture.

all of the core requirements in Section 3. Each packet is modulated into blocks of samples, for which we introduce per block meta-data. The meta-data stored in the header includes a timestamp (inbound and outbound), a channel flag (data/control), a payload length, and single bit flags to mark events such as overrun, underrun, or to request specific functions that we implement on the radio hardware. We limit the scope of the meta-data to the minimum needed to support the core requirements, thus minimizing the overhead on the bus.

Control Channel: The *control channel* allows us to implement a rich API between the host and radio hardware and allows for less frequent information to be passed. It consists of control blocks that are interleaved with the data blocks over the same bus. Control blocks carry the same meta-data header as data blocks but have the channel field in the header set to *CONTROL*. The control block payload contains one or more command subblocks. Each subblock specifies the command type, the length of the subblock, and information relevant to the specific command (e.g., a register number). Examples of commands include: reading or writing configuration registers on the SDR device, changing the carrier frequency, and setting the signal sampling rate.

With these two features, we can effectively partition the core MAC functions into a part that runs on the radio hardware close to the radio frontend, and a control part that runs on the host. Of course, meta-data and control channels are used in many contexts. The contribution lies in how we use them to partition the core MAC functions, which is the focus of the next section.

5 Core Component Design and Evaluation

We now examine how the split-functionality approach can be used to implement the core functions described in Section 3. We also evaluate the performance of the implementation of each core function. We focus our discussion on the GNU Radio and USRP platform.

5.1 Precise Scheduling in Time

Precision scheduling needs to be implemented close to the radio to achieve the fine-grained timing required for TDMA, spread spectrum, and contention based protocols. This is especially important when a large amount of jitter exists in the system from multiple stages of queuing and process scheduling, explored in Section 2.1.

For nodes to synchronize to the time of a global reference point, such as a beacon transmission for synchronization to the start of a round in a TDMA protocol, the nodes need to accurately estimate the reference point. Jitter at the transmitter can cause the actual transmission of the beacon to vary from its target time by δ_t, the maximum transmission jitter. Moreover, the estimated time of the beacon transmission as a global reference point will vary by δ_r, the maximum reception jitter. The maximum error is therefore $\delta_t + \delta_r$, which defines the minimum guard time needed by a TDMA protocol. By minimizing δ_t and δ_r, we increase channel capacity.

5.1.1 Precision Scheduling Design

Our delay measurements in Section 2.1 suggest that much of the delay jitter is created near the host. Therefore, the triggering mechanism for packet transmissions should reside beyond the introduction of the jitter. Likewise, to obtain an accurate local time at which a reception occurs, the time should be recorded prior to the introduction of the jitter on the RX path. To enable precision scheduling, we use a free running clock on the radio hardware to coordinate transmission/reception times as follows.

Transmit: To reduce the transmission jitter (δ_t), we insert a timestamp on all sample blocks sent from the host to the radio hardware. When the radio hardware receives the sample block, it waits until the local clock is equal to the *timestamp* value before transmitting the samples. This allows for timing compression or dispersion of data in the system with no effect on the precision scheduling of the transmission. The host must ensure the transmission reaches the radio hardware before the *timestamp* is equal to the hardware clock, else the transmission is discarded. The host is notified on failure, which can be treated as notification to schedule transmissions earlier. To support traditional best-effort streaming, we use a special timestamp value, called *NOW*, to transmit the block immediately.

In practice, the samples for a packet will be fragmented across multiple blocks. To make sure that a single packet's transmission is continuous and that if the packet is dropped all fragments are dropped, we implement *start of packet* and *end of packet* flags in the block headers. The first block carrying the packet will have the *start of packet* flag set and the timestamp for transmis-

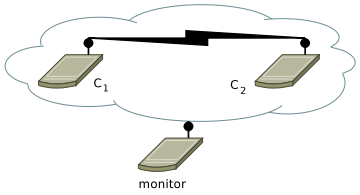

Figure 3: Evaluation setup using 3 USRPs.

sion. All remaining blocks carry a timestamp value of *NOW* to ensure continuous transmission. The hardware detects the last fragment using the *end of packet* flag, and can also report underruns to the host by detecting a gap between fragments.

A common solution to achieve precise transmission spacing from the host is to leave the transmitter enabled at all times and space transmissions with 0 valued samples. This solution is inefficient since it wastes both host CPU cycles and bus bandwidth, and it does not eliminate jitter on the receive side.

Receive: To reduce the receiver jitter (δ_r), the radio hardware *timestamps* all incoming sample blocks with the radio clock time at which the first sample in the block was generated by the ADC. Given that the sampling rate is set by the host, the host knows the exact spacing between samples. It can therefore calculate the exact time at which any sample was received, eliminating δ_r and allowing for full synchronization between transmitter and receiver.

5.1.2 Precision Scheduling Evaluation

To evaluate precision scheduling, we compare the timestamp-based release of packets using the split-functionality approach with a timer-based implementation in GNU Radio and in the kernel. We enable the real-time scheduling mechanism, which sets the GNU Radio processes to the highest priority. Our experiment transmits a frame used as a logical time reference, and then attempts to transmit another frame at a controlled spacing over the air. With no error, the actual spacing over the air is equal to the targeted spacing. We measure the actual spacings achieved using a monitoring node (Figure 3). A USRP on the monitoring node measures the magnitude of received complex samples at 8 megasamples per second, resulting in a precision of 125 nanoseconds. With no transmission jitter (δ_t), the spacing between beacons will exactly match their transmission rate, while any variability in scheduling will affect the spacings. The nodes are connected via coaxial cable to avoid the impact of external signals.

We compare the measured spacing of 50 transmissions with targeting spacings from 100ms to 1μs. Figure 4 shows the host and kernel based implementations to have approximately 1ms and 35μs of error, respectively. The timestamp-based mechanism achieves exact spacing to our monitoring node's precision. Therefore, moving timestamps to the kernel improves accuracy, but the error is still at least an order of magnitude greater than in the split-functionality design. Section 6.1 quantifies the benefits further through the implementation of a Bluetooth-like TDMA protocol. In the evaluation, we also measure δ_r with the split-functionality approach to be within 312ns. The average results show one-sided error, illustrating that compression of data across the bus dominates over dispersion. This is likely due to the multiple stages of buffers, including the buffers on the radio hardware to read the data from the FX2 controller. While dispersion is recorded, it occurs infrequently.

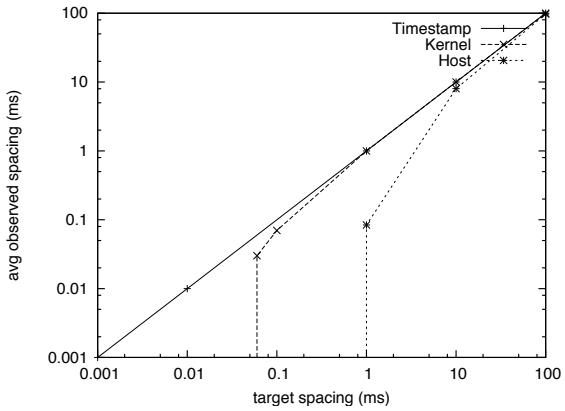

Figure 4: Split-functionality vs. host scheduling.

5.2 Carrier Sense

The performance of carrier sense is crucial to CSMA protocols: the longer it takes to transmit a packet after the channel goes idle, the greater the chance of collision. This turnaround time is referred to as the carrier sense 'blind spot" by Schmid et al. [14]. This blind spot has 4 components: signal propagation delay, the delay between the radio hardware and host for incoming samples, the processing delay involved in carrier detection at the host, and the complete transmission delay once the medium is detected idle at the host; this includes modulation of a packet and transferring the samples to the radio hardware for transmission.

5.2.1 Carrier Sense Design

To significantly reduce the size of the carrier sense blind spot, we must avoid the associated delays by placing the decision at the radio hardware. However, the decision process should be controlled by software running on the host CPU to maintain flexibility. The first assumption we can make is that if carrier sense is to be performed, the host has data to transmit and can modulate it and pass

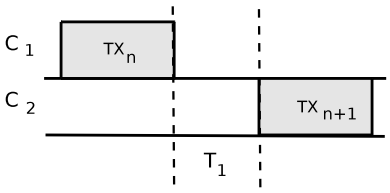

Figure 5: Carrier sense blind spot measurement.

it to the radio hardware to pend on carrier sense. The per block meta-data for the transmission has a single bit flag set to indicate the block should be held until there is no carrier using a locally computed RSSI value. The host can control the carrier sense threshold via the control channel. We use an RSSI value recorded in the radio hardware to implement a simple RSSI threshold carrier sense mechanism.

5.2.2 Carrier Sense Evaluation

We now present an evaluation of the *carrier sense* component in comparison to performing carrier sense at the host. In the host implementation, the received signal strength is estimated from the incoming sample stream and uses thresholds to control outgoing transmissions. We use the evaluation setup in Figure 3, described in Section 5.1.2, to achieve a 125 nanosecond resolution in measuring the archived carrier sense blind spot. The two contending nodes exchange the channel using carrier sense 100 times and we measure the spacing between each transmission, as illustrated in Figure 5. The first contending node, C_1, finishes transmission TX_n, and C_2 takes T_1 time to detect the channel as idle and begin transmission TX_{n+1}. T_1 represents the carrier sense turnaround time, or blind spot.

We plot two example channel exchanges using both implementations in Figure 6. Time is relative in the figure and we align the contending node's end of transmission at time 100. We highlight the gap in both implementations, and present the average gap observed across 100 exchanges: $1.5\mu s$ and 1.98ms for the split-functionality and host implementations, respectively. The host based latency could be reduced closer to 1ms, or on the order of tens of microseconds, by splitting the functionality to the USRP device driver, or the kernel, respectively. In our evaluation, the times were recorded at a higher-level block in GNU Radio where a MAC protocol would reside. These measurements illustrate our design's ability to reduce the carrier sense blind spot by *three orders of magnitude*, while maintaining host control on a per-packet basis. This can significantly increase the capacity in the channel by reducing the time it takes to detect it is idle. The host can even control the threshold on a per-packet basis by placing a control packet with a new threshold on the bus before the data packet.

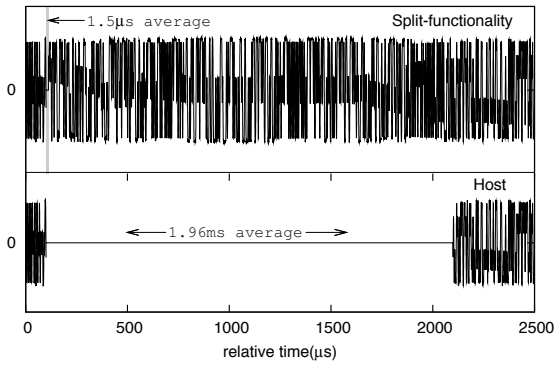

Figure 6: Measured carrier sense blind spots.

5.3 Backoff

In contention based protocols, backoff is used to reduce collisions and increase fairness. Although the technique varies by protocol, a common implementation is to reduce collisions by forcing a transmission delay and to increase fairness by making this delay random. The various delay components in SDRs prevent fine-grained backoff at the host. As shown in Section 5.1, a host backoff of less than 1ms is unachievable and values between 1ms and 100ms would be unpredictable. Therefore, backoff at the host would require a large minimum backoff time, which decreases channel capacity.

Despite our timestamping mechanism achieving microsecond level accuracy (Section 5.1.2), such a mechanism alone is insufficient. If a new backoff time is to be computed once a failure is reported to the MAC on the host, the retransmission would incur at least a radio-to-host RTT after the previous transmission, meaning the minimum backoff in a host implementation is an RTT. The average RTT measured in Section 2.1 was $612\mu s$ with a standard deviation was $789\mu s$ and a maximum observed value of 9ms. This is insufficient by current protocol standards. Placing the backoff algorithm on the radio hardware would require developers to make low level changes. We therefore explore a split-functionality approach for backoff.

5.3.1 Backoff Design

To enable flexible fine-grained backoff we build upon the precision scheduling mechanism (Section 5.1) to introduce a technique that leaves the backoff algorithm and computations at the host, and the actual transmission delay on the radio hardware. The key observation that enables our technique is that all backoff times, from the initial transmission n_0 to $n_{\text{MAX\_RETRIES}}$, can be precalculated by the host. The host calculates the backoff time for transmission n_0, and then assuming failure cal-

culates all remaining backoffs from 1 to *MAX_RETRIES*, including each in the per packet meta-data.

A flag is set in the per block meta-data for the radio hardware to interpret the timestamp value as the maximum number of retries (M), and the first M 32-bit words pre-pended in the data payload to be interpreted as backoff times for each retransmission. Each value is interpreted as a time-to-wait, where the transmission is scheduled at *current_clock+backoff*. Moreover, we implement a control channel command that allows the host to configure the interpretation of a backoff value as an absolute time-to-wait, or a channel idle time-to-wait (most common).

This technique does not affect scheduling of future transmissions, as for example in 802.11 the contention window is reset to the minimum on a successful transmission. This means that the host can fully schedule a transmission and before a success/failure notification is given by the hardware, it can prepare the next transmission and buffer it on the radio hardware.

5.3.2 Backoff Evaluation

Given that the backoff technique uses the precision scheduling mechanism, its accuracy is the same as the precision scheduling mechanism and on the order of microseconds. We also use the backoff technique in our split-functionality 802.11-like protocol evaluation found in Section 6.

5.4 Fast Packet Recognition

Traditional software-defined radios, in the receive state, stream captured samples at some decimated rate between the radio hardware and the host. For many MAC protocols, such as CSMA-style designs, the radio cannot determine when packets for the attached node will arrive. As a result, the radio must remain in the receiving state. The downside to this is that the demodulation process uses significant memory and processor resources despite the fact that incoming packets destined for the radio are infrequent. As such radios become more ubiquitous and common for implementation, resource usage will become increasingly important, especially for energy-constrained devices such as the battery-powered Kansas University Agile Radio [9].

One simple solution would be to send samples when the RSSI is above some threshold. However, this does not filter out transmissions destined to other hosts and external signals. A better solution would be to have the radio hardware look for the packet preamble and the destination address, then transfer a maximum packet size worth of samples to the host after any match. At first glance, it may seem that fast packet recognition

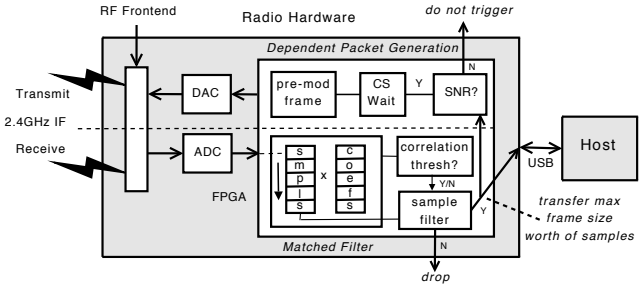

Figure 7: Matched filter & dependent packet design.

is not a "necessary" function for implementing MAC protocols, especially since the CPU and bus bandwidth resource consumption can become insignificant rather quickly (i.e., due to Moore's Law). However, trends in bus delay do not have this same property. As we will discuss further in Section 5.5, the ability to identify packets and process them partially on the SDR hardware is critical to supporting low-latency MAC interactions (e.g., packet/ACK exchanges or RTS/CTS) in a high-latency architecture.

5.4.1 Fast Packet Recognition Design

Our goal is to accurately detect packets at the radio hardware without demodulating the signal (to keep flexibility), for which we perform signal detection. The most relevant work in signal detection comes from the area of radar and sonar system design. From this area, we borrow a well-known technique, called a *matched filter*, to detect incoming packets at the radio hardware without the demodulation stage. For the purpose of design discussion, we refer to the bottom half of Figure 7.

Matched filter: A matched filter is the optimal linear filter that maximizes the output signal to noise ratio for use in correlating a known signal to the unknown received signal. For use in packet detection, the known signal would be the time-reversed complex conjugate of the modulated framing bits. This known signal is stored as the coefficients of the matched filter (Figure 7). The received sample stream is convolved with the coefficients to perform cross-correlation, where the output can be treated as a correlation score between the unknown and known signals. The correlation score is then compared with a threshold to trigger the transfer of samples to the host. The matched filter is flexible to different modulation schemes (e.g., GMSK, PSK, QAM), but requires a Fast Fourier transform for OFDM, given that the symbols are in the frequency domain. This would require an FFT implementation on the radio hardware.

To also detect that the frame is destined to the particular host, two different methods that have mathematically different properties can be used. *Single Stage*: Use a frame format where the destination address is the

first field after the framing bits, and use this complete modulated sequence as the matched filter coefficients. *Dual Stages*: detect the framing bits first, then change the coefficients to the modulated destination address. Our implementation uses the single stage approach for simplification. However, a dual stage is more appropriate for monitoring multiple addresses such as a local address and a broadcast address.

5.4.2 Fast Packet Recognition Evaluation

We evaluate the effectiveness of the matched filter at detecting incoming sequences using simulations where we can control the noise level. Results are presented from over the air experiments with the presence of interference, multipath, and fading in Section 5.5.

To evaluate the effectiveness of the matched filter with varying signal quality, we first run experiments with controlled signal-to-noise ratios (SNR) using the GNU Radio software. We introduce additive white Gaussian noise (AWGN) to control the SNR in terms of dB:

$$SNR(dB) = 10 * log_{10} * \frac{Power_{signal}}{Power_{noise}} \quad (1)$$

To introduce noise, we compute the noise power based on the specified *snr* and power in the signal:

$$SNR = 10^{(snr/10)}$$
$$Power_{signal} = |Signal_{ampl}|^2$$
$$Power_{noise} = \frac{Power_{signal}}{SNR}$$

For evaluation, 1000 frames of 1500 bytes are encoded using the Gaussian minimum-shift keying (GMSK) modulation scheme. These frames are used as the ground truth and mixed with the noise. We require that the matched filter detect the framing bits *and* that the transmission is destined for the attached host using the single-stage scheme (Section 5.4.1). The success rate is defined as the number of detected frames over the total number of frames in the dataset (1000). For comparison, we also include the success rate of the full GMSK decoder. At a high noise level, even the full decoder will fail at detecting the frames. The success rate, as a function of the SNR, is shown in Figure 8. The results show that the matched filter can detect the frames at a much higher success rate than the decoder can, even at low SNR levels where the noise power is greater than the signal power.

Given these results, and further real-world results presented in Section 5.5, we conclude that using the matched filter for detecting relevant packets is accurate enough that the host will never miss an actual frame due to the filter. In fact, the filter triggering samples to the host can been seen from a different perspective as providing further confidence to the host that there is actually a frame within the sample stream. The host could then perform additional processing in an attempt to decode the frame successfully.

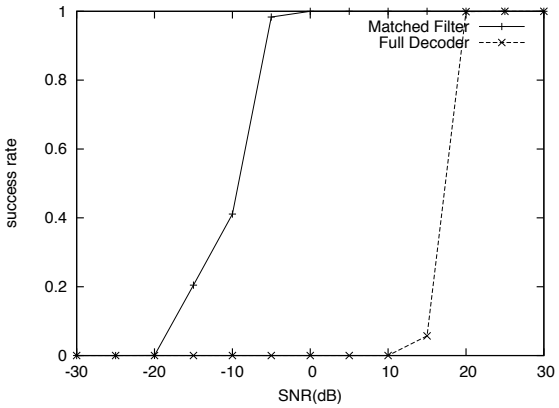

Figure 8: Success rate of the matched filter.

5.5 Dependent Packets

Dependent packets are packets generated in response to another packet (e.g., an ACK or RTS packet). MAC protocols often leave the channel idle during the dependent packet exchanges such as RTS-CTS and data-ACK exchanges. As a result, reducing the turnaround time of such exchanges can significantly increase overall capacity. In a host-based MAC, three sources contribute to the delay associated with dependent packet generation: bus transmission delay, queuing delay, and processing time. In this section, we explore the use of a matched filter along with additional techniques for triggering dependent packet responses on the radio hardware. The technique minimizes processing time by placing the packet detection as close to the radio as possible and avoids bus transmission and queuing delays by triggering a pre-modulated packet stored on the radio hardware.

5.5.1 Decoding Delay at the Host

We begin by quantifying the processing delay associated with host-based dependent packet generation. Note that we have already quantified bus delays in Section 2.1. We measure decode time for various frame sizes at the maximum supported decoding rate of the USRP: 2Mbps. The larger frame sizes would be representative of processing time for data/ACK exchanges, and the smaller frame sizes for RTS/CTS exchanges.

We use two 3.0GHz Pentium 4 machines running GNU Radio with their USRPs transmitting/receiving using the GMSK modulation scheme. Using host based timers, we record the minimum, average, and maximum time to decode 6 different frame sizes seen in Figure 9. The average decoding time is close to the mini-

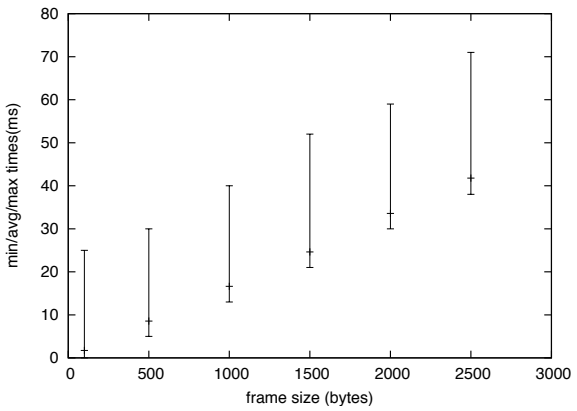

Figure 9: Decode times for various frame sizes.

mum recorded times for each frame size, however, rather large delays can be experienced at each frame size, likely due to the jitter introduced by queuing delays and process scheduling. Therefore, if one were to implement the matched filter at the radio hardware to detect incoming dependent packets and generate responses, anywhere from several milliseconds to 70 milliseconds can be saved solely in host processing.

5.5.2 Generating Fast-Dependent Packets

As an optimization to circumvent the decoding delays described, we develop a mechanism for fast-dependent packet generation in the radio hardware. This is not necessarily limited to *host-PHY* architectures. Although bus delay is reduced in *NIC-PHY* architectures, they typically use slower processors that increases decoding delays. Fast-dependent packet generation has three stages: (1) fast-packet detection of the initiating packet (e.g., RTS), (2) conditionals specific to the protocol that trigger the dependent packet, and (3) transmission of a pre-modulated dependent packet. We discuss stages 2 and 3 in this section. **Stage 1** was detailed in Section 5.4, although it is important to point out that by running multiple matched filters in parallel, it is possible to detect and respond to different initiating packets.

Stage 2: To introduce protocol dependent behavior after stage 1 detects the initiating packet and its end of transmission (the incoming signal drops to the noise floor), protocol developers can introduce a set of conditionals that control when a dependent packet is generated. In our current implementation this must be written in a hardware description language (Verilog), which has primitives similar to those in C/C++ (e.g., if, else, case, etc.). A simple example is the conditional for generating a CTS in Verilog. It checks that the receiver and channel are idle: *if(!receiving && RSSI < carrier_sense_thresh)*.

A more interesting example is the fast-ACK generator developed for our 802.11-like protocol (Section 6.3). We write 3 simple conditional statements around an SNR value. If any of the conditionals *pass* during the transmission, the radio hardware concludes that the host would not have been able to decode the packet, and a fast-ACK should not be triggered. The following are the 3 conditionals, with reasons as to why the fast-ACK should not be generated based on the conditional passing. (1) *if(SNR < lowest_thresh)*: interference throughout the transmission. (2) *if(last_SNR_val - SNR < drop_thresh)*: interference at the tail of the transmission, or fading. (3) *if(SNR - last_SNR_val > increase_thresh)*: interference at the head of the transmission, or multipath. The technique is illustrated in the overall system in Figure 7, where the correlation threshold for a data packet raises a signal which streams the samples to the SNR monitor. The final conditional is to detect the carrier as idle; then the fast-ACK is generated.

Stage 3: To satisfy fast-dependent packet generation, the dependent packet must be pre-modulated and stored on the radio hardware, for which we provide a mechanism on the control channel. Pre-modulation restricts the dependent packet to not contain fields dependent on the initiating packet (e.g., a MAC address). However, it still permits many dependent packets like those in current protocol standards (e.g., ACKs, RTS/CTS). For example, despite 802.11's requirement for a destination address in an ACK packet, we can still develop and evaluate an 802.11-like protocol where senders assume the destination of the ACK based on data transmissions. We remind the reader that a goal of our work is to enable MAC implementations and building blocks for novel MAC designs, not to necessarily support every current protocol to its specification. Future work could be in the development of a technique which extracts part of an incoming signal (e.g., destination address) and then performs additional processing to use this raw signal in a pre-modulated dependent packet. This would essentially enable dynamic fast-dependent packets, without the interaction of the host. We do not explore this in the scope of our work.

Fast-Dependent Packet Evaluation: To illustrate the fast-dependent packet generator, we evaluate an implementation of the fast-ACK generator outlined in the description of *stage 2*. First, we use the control channel to setup a matched filter which detects the framing bits and the attached node's address (satisfying stage 1). Then, we pre-modulate an ACK that uses the broadcast address as the destination address for all active nodes to parse it (satisfying stage 3).

To evaluate the SNR monitoring technique, and further evaluate the matched filter's ability to detect packets in a real world scenario, we use a 2 USRP-node setup in the ISM band for presence of 802.11 and Bluetooth devices, incorporating real world interference in our re-

sults. We detected 6 active 802.11 devices within interference range, but ensured that none were within 40 feet of either node. To test in adversarial conditions with multipath interference, the two USRPs were placed in separate rooms with no direct line of sight. The matched filter and fast-ACK technique are enabled at the receiver, for which we transmit 10000 frames to at 1Mbps. These frames are considered the ground truth for the matched filter, which we are trying to determine the accuracy of in detecting the frames. Full decoding of the data packets at the host is used as the ground truth for the fast-ACK generator. If the full decoder successfully decodes the frame, and the SNR monitor triggers a fast-ACK, it is considered success. If the SNR monitor chose to not generate a fast-ACK in this scenario, it is considered failure. An additional failure scenario is triggering a fast-ACK when the host could not decode the frame.

For the 10000 frames transmitted, we find that the matched filter is able to detect the transmissions with 100% success rate, reinforcing the simulation results from Section 5.4.2 with real world signal propagation properties. Of the 10000 frames, 460 transmissions were not decodable. Using the SNR monitoring technique *we detect 457 of the corrupted frames for a failure rate of 0.6%*. Inspection of the 3 misses could not determine the cause of transmission failure. The error rate of not generating an ACK, when one should have been, is 4%.

There are implications to incorrectly generating ACKs, which the MAC can be designed to recover from, or higher layers such as TCP can be relied on. Our evaluation further explores the matched filter's accuracy and illustrates the ability to implement fast-dependent packets. Reducing the error rates seen by our technique is future work, either by improving the SNR monitoring technique, or introducing other fast-ACK techniques. An example for improvement would be detecting multipath during SNR monitoring, which is a property that can reduce decoding probability.

5.6 Access to Physical Layer Information and Fine-grained Radio Control

The underlying radio hardware in an SDR platform has many controls that are not configured by the transmitted sample stream (e.g., transmission frequency and power), and can make many observations that are not easily derived from the input sample stream (e.g., RSSI). We use our control channel between the SDR hardware and host to expose these controls and physical layer information to the MAC protocol implementation. Many existing network interface use similar designs for setting the transmission channel and obtaining RSSI measurements. One key difference is that our interface operates on blocks of samples instead of packets.

Physical Layer Information: Access to physical layer information at all other layers in the processing chain is important for supporting common cross-layer optimizations. This can be seen through recent work where per-bit confidence levels are used to perform partial packet recovery [7]. In our design, information from the SDR can be sent to the host using either the control channel or per block meta-data. We use this mechanism to report RSSI to the host. Note that the host could calculate RSSI using the raw samples, but an RSSI value which takes into account the gain or attenuation in the RF stages is only available at the radio hardware. The control protocol is easily modified to support reporting additional properties, however, developers must reprogram the FPGA to report the desired values.

Radio Control: We implement a set of radio hardware control messages on the control channel (Section 4.2) that can be synchronized with packet transmissions using the timestamp. For example, by placing a control block with a timestamp T before a data packet on the bus, which uses a *NOW* timestamp, the radio will be reconfigured at time T and the data packet will be transmitted immediately after the reconfiguration. This can be used to implement common techniques such as rapid frequency hopping. Unfortunately on the USRP, the daughterboards are tuned directly from the FX2 USB controller using the I^2C bus, which has no connection to the FPGA. Therefore, we cannot issue daughterboard commands from the FPGA using the control channel and hardware clock to implement rapid frequency hopping. The USRP2 tunes the daughterboards directly from the FPGA. Therefore, if our design was implemented on the USRP2, unavailable at the time, rapid frequency hopping could be achieved.

6 MAC Evaluation

We now provide end-to-end results for a Bluetooth-like TDMA protocol and 802.11-like CSMA protocol. The protocols use the *split-functionality* design described in Section 5 and we compare their performance with that of full host-based implementations.

6.1 Bluetooth-like TDMA Protocol

To illustrate the effectiveness of the overall system design, we implement a tightly timed Bluetooth-like TDMA protocol. Like Bluetooth, the network (piconet) consists of a master and a maximum of 7 slaves. The slaves communicate with the master in a round-robin fashion within a slot time of $625\mu s$. Unlike Bluetooth, our protocol fixes its frequency instead of hopping (a

limitation of the USRP discussed in Section 5.6), varies slightly in synchronization (bypasses *pairing*), and the slot guard time is varied for evaluation.

Each slave in the network synchronizes with the start of a round by listening for the master's beacon, and calculates the start of transmission (Section 5.1) as the logical synchronization time T. The beacon frame also carries the total number of registered slaves (N) and the guard time (T_g). The slave can then compute the total round time, which must account for the master: $T_r = N + 1 * (T_s + T_g)$, where T_s is the slot time ($625\mu s$). The start of round k is computed as: $T_k = T + T_r * k$. We remind the reader that this is a logical time kept at each node, taken from the beacon frame which is a global reference point. Global hardware clock synchronization is explored in Section 6.2. Finally, each slave's slot offset is computed from its node ID (n), $\delta_n = n * (T_s + T_g)$, which is then used to compute the local start time of slave n's slot in round k: $T_{n(k)} = R_k + \delta_n$.

6.1.1 TDMA Results

We use two metrics in our evaluation: ability to maintain tight synchronization and overall throughput. The synchronization error at the master is 15ns, computed by measuring the actual spacing of 1000 beacons using a monitoring node (discussed in Section 5.1.2). This illustrates the tight timing of the master's beacon transmissions. To measure the synchronization error at the slaves, we record the calculated timestamps of 1000 beacons at 4 slaves. Each timestamp should be exactly T_r apart from the next. The absolute error in spacing represents shifts in the slave's calculation of the start of the round. We find the maximum error of the 1000 beacons at all 4 slaves to be 312 nanoseconds, with an average of 140ns. This answers the question of our platform's ability to obtain tight synchronization at both transmitters (master) and receivers (slaves).

We compare a split-functionality implementation to a host implementation, which differ in their guard times. A guard time of $1\mu s$ is used for the split-functionality implementation, which is nearly 3 times the maximum error. We use our round trip host and radio hardware delay measurements from Section 2.1, which accounts for both transmissions and reception timing variability, to estimate the host guard time needed. A guard time of 9ms would be needed to account for the maximum error, however, this delay occurs rarely and we therefore present results using a generous guard time of 3ms (approximately $3 * \text{sdev}$) and a more realistic guard time of 6ms based on our recorded delay distribution.

We perform 100KB file transfers, varying the number of registered slaves and presenting averaged results across 100 transfers in Figure 10. The *split-functionality* implementation is able to achieve an average of 4 times the throughput of the host based implementation. While we had only been able to answer the question of obtaining synchronization, we find that throughout the full transfers no slave drifts into another slot period using only the initial beacon for synchronization, illustrating the ability to *maintain* tight synchronization. These results are promising for the development of TDMA protocols on the platform.

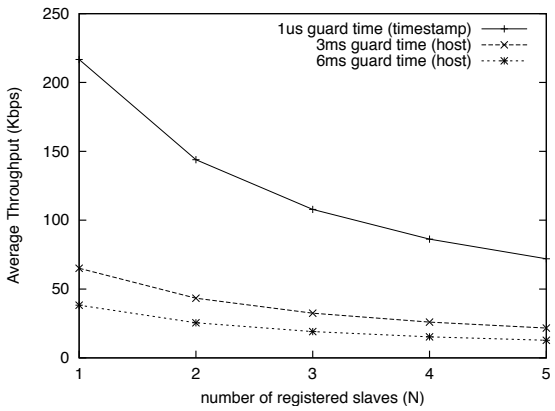

Figure 10: TDMA throughput comparison results.

6.2 Additional TDMA Protocols

Another common TDMA implementation is the use of global clock synchronization. We extend the Bluetooth-like protocol to use global clock synchronization on the platform rather than the logical clock. The implementation design is as follows. The global clock in the network is the clock of the master, to which all slaves synchronize via beacon frames. In addition to the information sent in each beacon frame described in Section 6.1, the master includes the timestamp at which the beacon is locally scheduled for transmission.

For global synchronization, the slave takes its estimated local time of the master's beacon transmission and subtracts the incoming global clock timestamp included in the beacon to calculate δ, the local clock offset from the master. The error is within 312ns plus over-the-air propagation delay. The MAC framework can now synchronize to the global clock with a command packet (Section 4.2) which adds δ to the local clock. Another option is to use a timestamp transformation where the MAC adds δ to all timestamps. Using this methodology, we are able to achieve measurement results similar to those in Figure 10 using global synchronization.

6.3 802.11-like CSMA Protocol

We implemented two 802.11-like CSMA MAC protocols, one fully on the host CPU and one using our

	pairs	Avg (Kbps)	min	max
platform	1	408	387	415
host	1	215	190	240
platform	2	205	201	210
host	2	112	101	130

Table 2: 802.11-like CSMA protocol per-pair results.

split-functionality optimizations including on-board carrier sense (Section 5.2), dependent packet ACK generation (Section 5.5), and backoff (Section 5.3). The MAC implements 802.11's clear channel assessment (CCA), exponential backoff, and ACK'ing. Our protocol does not implement SIFS and DIFS periods; this work is in progress. For space reasons, we focus our description on how the 802.11-like protocol uses our architecture.

The host-based implementation places all functionality on the host CPU, including carrier sense, ACK generation, and the backoff. The optimized implementation uses the matched filter and SNR monitoring for ACK generation, and performs carrier sense and backoff on the radio hardware. We configure the USRPs for a target rate of 0.5Mbps, and run 100 1MB file transfers for each implementation using a center frequency of 2.485GHz in an attempt to avoid 802.11 interference. This allows us to present results that highlight the differences in the implementation without the effect of uncontrolled interference. We also vary the number of nodes in the network, where each pair of nodes performs a transfer.

The results for the two implementations are shown in Table 2. We see significant performance increases from the use of the *split-functionality* implementation. This nearly doubles the throughput on average, likely due to the time saved in decoding to generate the ACK, and the delays associated with carrier sense and backoff. We note that the matched filter detected every framing sequence, and the fast-ACK generation technique only failed 2 times over the total number of runs. To recover from these failures, we implemented a feedback mechanism on the host that checks the SNR monitoring technique's decision and retransmits. This is needed since we did not use a higher-layer recover mechanism like TCP.

7 Related Work

We review related work in the area of MAC development. Existing platforms mostly use the extremes of the design space where either the majority of functionality is fixed on the network card (*Traditional NICs*), or perform all processing at the host (*Software-defined Radios*).

7.1 Traditional NICs

Several efforts [13, 4, 16] have built new MAC protocols on top of existing commercial NICs (e.g., 802.11 cards). Unfortunately, commercial 802.11 cards implement the bulk of the MAC functionality in proprietary microcode on the card, limiting what functions can be changed by researchers. As a result, this approach is not very satisfactory: the range of MAC protocols that can be implemented is limited and performance (e.g. throughput, capacity) is often poor from the MAC needing to be implemented on the host. For example, past efforts have mostly implemented TDMA-based schemes.

7.2 Software-defined Radios

Software-defined radios (SDRs) provide a compelling architecture for flexible wireless protocol development since most aspects of both the MAC and physical layer are, by design, implemented in software and thus in principle, easy to modify. However, so far, SDR efforts have focused on implementing the physical layer [19] while MAC and higher layer protocol development has received little attention.

Recent work by Schmid et al [14] examines the impact of increased latency in software-defined radios using GNU Radio and the USRP. The authors address how the bus latency creates "blind spots" that increase collision rates when carrier sense is performed at the host, and how pre-computation of packets is not possible without fully demodulating (at the host), resulting in larger interframe spacing. Our design provides solutions for both of these issues in Sections 5.2 and 5.4, respectively. Bus delay measurements were also taken by Valentin et al [18].

On top of these hardware challenges, the original streaming-based design of GNU Radio and the fixed size data limitation on its blocks prevents packet processing. Dhar et al [3] take the approach of integrating the Click modular router [12] with GNU Radio. GNU Radio blocks are imported into Click to handle the physical layer, while Click is used to implement the MAC layer. Additionally, the authors interface with the USRP to provide a full SDR. Another approach extended the GNU Radio architecture with *m-blocks* [2], blocks that allow variable length data passing and include meta-data that can be used to represent packets. Our work is complementary to the above efforts: while they focus on a MAC development environment on the host, we focus on the partitioning of MAC layer processing between the host and radio hardware. Our architecture and results also do not depend on a particular environment on the host.

A number of groups have developed software radios with architectures that differ from the current GNU Radio and USRP design by including a CPU on the radio hardware (NC-CPU), either as a separate compo-

nent or as a core on the FPGA. Examples include the Rice University Wireless Open-Access Research Platform (WARP) [20] and USRP2. These designs are more expensive, but they offer additional flexibility for partitioning the MAC. However, there is still a non-trivial delay (compared with traditional radios) due to physical layer processing and queueing. The NC-CPU is also likely to be slower than the host CPU, increasing the processing delay. Finally, in deployed products based on this architecture, the NC-CPU is likely to be off-limit to users, similar to the current situation with commercial wireless cards. As a result, we expect that our architecture will be useful this type of platform as well.

8 Conclusions

In this paper, we presented a set of techniques that support the implementation of diverse, high-performance MAC protocols on software radios. The work is motivated by the observation that a single one-size fits all MAC protocol cannot meet the demands of increasingly diverse deployments and application loads. Software radios offer flexibility, but their architecture, specifically the delay between the host and the radio frontend, has traditionally been a problem for MAC protocols. We introduce a split-functionally approach, which addresses this problem, and show that it enables the implementation of a set of core MAC functions. An implementation for the USRP and GNU Radio, along with the implementation of an 802.11-like and Bluetooth-like protocol, shows the approach is effective. To our best knowledge, these protocol implementations are the first high-speed, bi-directional MAC implementations for the GNU software radio platform. For future work, we plan to implement a more diverse set of MAC protocols to further evaluate our design and implement the architecture on different SDR platforms to evaluate its generality.

Acknowledgments

We thank the GNU Radio community for the help provided, especially the support from Eric Blossom and Matt Ettus, and their collaboration in the design of the control channel. A sincere thank you to Brian Padalino for the constant feedback and guidance throughout our work. This work was supported by grant CNS-0626827 from the National Science Foundation.

References

[1] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self-management in chaotic wireless deployments. In *ACM MobiCom*, pages 185–199, 2005. ISBN 1-59593-020-5. doi: http://doi.acm.org/10.1145/1080829.1080849.

[2] BBN:ArchChanges. BBN Technologies Corperation, GNU Radio Architectural Changes (m-block). http://acert.ir.bbn.com/downloads/adroit/gnuradio-architectural-enhancements-3.pdf.

[3] R. Dhar, G. George, A. Malani, and P. Steenkiste. Supporting Integrated MAC and PHY Software Development for the USRP SDR. In *IEEE Workshop on Networking Technologies for Software Defined Radio (SDR) Networks*, Reston, 2006.

[4] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. C. Sicker, and D. Grunwald. MultiMAC - An Adaptive MAC Framework for Dynamic Radio Networking . In *IEEE DySPAN*, 2005.

[5] S. Gollakota and D. Katabi. Zigzag decoding: Combating hidden terminals in wireless networks. In *ACM SIGCOMM*, New York, NY, USA, 2008. ACM Press.

[6] GR. Gnu radio. http://www.gnu.org/software/gnuradio/.

[7] K. Jamieson and H. Balakrishnan. Ppr: partial packet recovery for wireless networks. *SIGCOMM Comput. Commun. Rev.*, 37 (4):409–420, 2007. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1282427.1282426.

[8] S. Katti, D. Katabi, H. Balakrishnan, and M. Medard. Symbol-level network coding for wireless mesh networks. In *ACM SIGCOMM*, New York, NY, USA, 2008. ACM Press.

[9] kuagile. Kansas university agile radio. https://agileradio.ittc.ku.edu/.

[10] M.-H. Lu, P. Steenkiste, and T. Chen. Flexmac: a wireless protocol development and evaluation platform based on commodity hardware. In *WiNTECH '08: Proceedings of the third ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 105–106, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-187-3. doi: http://doi.acm.org/10.1145/1410077.1410102.

[11] A. Mishra, V. Shrivastava, D. Agrawal, S. Banerjee, and S. Ganguly. Distributed channel management in uncoordinated wireless environments. In *ACM MobiCom*, pages 170–181, 2006. ISBN 1-59593-286-0. doi: http://doi.acm.org/10.1145/1161089.1161109.

[12] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. volume 33, pages 217–231, New York, NY, USA, 1999. ACM. doi: http://doi.acm.org/10.1145/319344.319166.

[13] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. SoftMAC - Flexible Wireless Research Platform. In *Fourth Workshop on Hot Topics in Networks (HotNets)*, 2005.

[14] T. Schmid, O. Sekkat, and M. B. Srivastava. An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios. In *WiNTECH'07*, 2007.

[15] A. Sharma and E. M. Belding. Freemac: framework for multi-channel mac development on 802.11 hardware. In *PRESTO*, pages 69–74, 2008.

[16] A. Sharma, M. Tiwari, and H. Zheng. MadMAC: Building a Reconfigurable Radio Testbed Using Commodity 802.11 Hardware. In *IEEE Workshop on Networking Technologies for Software Defined Radio Networks*, Reston, 2006.

[17] USRP. The universal software radio peripheral. http://www.ettus.com/.

[18] S. Valentin, H. von Malm, and H. Karl. Evaluating the gnu software radio platform for wireless testbeds. In *Technical Report TR-RT-06-273*, 2006.

[19] Vanu. Vanu software radio systems. http://www.vanu.com.

[20] WARP. Rice university wireless open-access research platform (warp). http://warp.rice.edu.