Experiences With a Platform for Frequency-Agile Techniques

Jeff Fifield, Paul Kasemir, Dirk Grunwald and Douglas Sicker
Dept. of Computer Science
University of Colorado, Boulder
Boulder, CO 80309-0430

Abstract—**One of the challenges impeding the development of frequency-agile interfaces is the lack of a robust platform that allows fine grain frequency control. As part of an NSF program to develop an experimental platform for software defined radio we have developed a FPGA-based software defined radio using an OFDM waveform. The current platform can transmit OFDM packets using a 256-FFT with sufficient robustness to be received by commodity 802.11g radios. Now that ”backward compatibility” has been implemented, the platform is being evolved to provide a number of profiles more suitable for frequency-agile experimentation. In particular, we provide interfaces to control individual subcarriers using specific modulations (BPSK, QPSK, QAM) and to aggregate ”bundles” of subcarriers into distinct radio interfaces that can be controlled at the application layer. This paper focuses on the architectural and design choices made on the transmit end of the radio to support this functionality; suprisingly, the system is difficult to build using existing SDR development frameworks such as GNU Radio. The end result is a radio system that lets applications “dial down” the bandwidth they use or to sense and react to subcarrier specific fades and interference. We believe this platform will enable expanded experimentation in frequency agile research.**

I. Introduction

Software defined radios [1], [2] have long been touted as a technology that enables cognitive radios [3], or radios that sense and respond to their environments. At their ideal, a software defined radio would use a general purpose processor and a programming environment that simplifies the implementation of a specific waveform. However, most extant implementations of software radios either use a combination of DSP or FPGA accelerators and a general purpose processor or make exclusive use of accelerators and implement the full waveform as an embedded application. Some platforms use an FPGA simply to convert the data to a form suitable for an upconverter (*e.g.,* performing a simple interpolation), while other platforms use FPGAs to do more extensive signal processing.

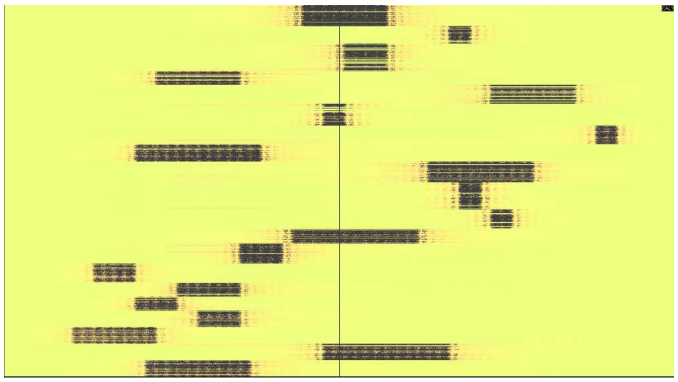

Fig. 1. Screenshot of a power-spectral plot showing spectrum agility using the OFDM platform described in the paper. The x-axis is frequency and the y-axis is time. Darker color shows higher measured power while lighter color indicates low or no measured power.

This paper reports on our experience in implementing a flexible orthogonal frequency division multiplexing (OFDM) waveform using a software defined radio platform. While our experiences are not ’novel’ in the sense that we have developed a new waveform or signal processing technique, our experience is a useful synthesis of the current state of the software and hardware development processes that would be encountered by many seeking to build such a software radio.

In particular, the use of an OFDM waveform requires a re-examination of the software design of common software processing packages, such as GNU Software Radio; this occurs because OFDM transmits “frames” or sets of OFDM symbols rather a single continuously modulated waveform. This highlights the tension between “frame” and “stream” processing, which also occurs when converting “network frames” (*e.g.,* packets) into “RF streams”. This simple change has implications about the software interface between a host based processor and FPGA accelerator.

Moreover, our use of OFDM requires the transmission of multiple, simultaneous streams – we want to be able to have an access point transmit to multiple stations simultaneously and without coordination between the different receiving stations. This requirement further complicates the buffer management and the I/O interface between the host processor and the FPGA accelerator.

Lastly, because we are transmitting a wide bandwidth (up to 35MHz), the waveform is computationally demanding. Current general purpose processors keep up with smaller bandwidths, but we found that the processor I/O subsystem, rather than the CPU, is likely to be the greater source of bottlenecks for the immediate future, particularly as general purpose processors either expand to multiple cores or incorporate hardware units such as graphics processors.

We have experimented with multiple versions of the transmit chain for the OFDM waveform we describe in the next section. Figure 1 shows a "waterfall plot" of the OFDM waveform as we change both the frequency center and used bandwidth within a possible 35MHz band. Our final design allows flexible host-based configuration and control of individual subcarriers and should readily be adaptable to newer versions of the GNU Software Radio framework when they become available. Although we have a functioning receive chain, it is completely implemented in the FPGA and has not been subject to the same design considerations as the transmit chain that we describe in this paper.

In the next section, we describe enough about the OFDM waveform for a non-specialist to understand how it functions and the computation necessary; we also describe existing software radio frameworks. In §III we describe in more detail the extra functionality needed for our projects since that functionality has caused us to redesign the hardware-software interface. We also describe how our OFDM waveform is actually implemented and present an example application. Next, we reflect on the current state software radio frameworks and what can be done to improve them. Lastly, we survey related work and conclude.

II. Background

In this section, we provide background information on the OFDM waveform that is central to the SDR design described in the paper. We also describe the software flow of the GNU Radio software that is rapidly becoming a popular platform for developing SDR applications. While it is not important to understand the details of the GNU Radio software, it is important to understand the *design style* used in that software since it permeates many other SDR toolkits and frameworks. We will illustrate that point by also briefly examining how "software radio chains" can be implemented using the MatLab Simulink tools. Lastly, we will describe how the software is combined with hardware to produce a functioning radio.

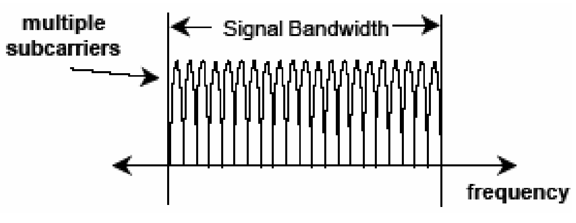

Fig. 2. Sample OFDM Waveform

All of this material is preamble to the design challenges we have and are facing; those challenges (and the solutions to them) are described in the following section.

A. OFDM

Orthogonal Frequency Division Multiplexing (OFDM) is a modulation technique that uses a large number of "subcarriers" allocated across a specific frequency range [4]. A sample waveform for OFDM is shown in Figure 2. OFDM is a popular modulation technique and is used in the 802.11a and g physical layers as well as the 802.16-2004 specification. Each subcarrier of an OFDM waveform is modulated at a fairly low rate, but the aggregate rate of all the subcarriers provides high throughput. For example the 802.11a PHY has a total of 52 data subcarriers modulated at 312.5KHz, and is able to transmit data at a rate of 54Mbits/sec. The symbol duration of an OFDM symbol is the number of subcarriers times the rate at which a single subcarrier is modulated. For 802.11a the symbol time is 3.2 μseconds, which is long compared to other high rate modulation techniques. The long symbol time means that the individual subcarriers are highly resistant to intersymbol interference (i.e., interference between consecutive symbols in time) caused by multipath, which occurs when multiple copies, or echos, of the signal arrive at the receiver at different times. This resistance to multipath also simplifies receiver design.

In most OFDM implementations, every subcarrier is modulated using the same waveform on each subcarrier – for example, each subcarrier may be modulated using a BPSK (binary phase shift keying) modulation that is robustly resistant to noise but offers a low throughput. Many standards employ an adaptive modulation method, which allows all of the subcarriers (as a group) to move between different waveforms (such as BPSK, QPSK or QAM encodings). Tradeoffs are made between higher throughput (e.g., QAM) or more resistance to noise (e.g., BPSK). In theory, different subcarriers can be modulated using different waveforms [5] - for example, if an environment has narrowband noise in a small part of the frequency space, it may be possible to use BPSK on those subcarriers and use high rate modulation on other subcarriers or to even omit certain portions of the subcarriers. One of the goals of our implementation of OFDM is to allow implementation of such an adaptive waveform. This goal increases the complexity of the transmitter because a differing number of bits need to be converted to symbols (*e.g.,* a BPSK modulation might encode 1 bit within a binary symbol, while a 64-QAM modulation might encode 6 bits in a single symbol).

In addition to the basic subcarrier modulation, an effective OFDM waveform must introduce *guard intervals* (also called a *cyclic prefix*) to reduce the effect of any multipath on OFDM symbols. Despite the fact that individual subcarriers are resistant to multipath, the FFT decoder is sensitive to multipath. Moreover, some of subcarriers may be devoted to *pilot tones* to assist in estimating frequency specific fading. For example, in the 802.11a and 802.11g OFDM waveform, the base symbol time is $T_s = 3.2$ μseconds with a $\Delta = 0.8$ μseconds guard interval for a total symbol time of $T = 4$ μseconds. Of the $K = 52$ subcarriers, four carriers are used as pilot tones. A 64 element FFT is used to convert the time-series samples into frequency samples. These parameters (K, T, T_S and Δ) are chosen for specific applications of OFDM based on the bandwidth available, physical properties of RF near the center frequency and estimates of both mobility and multipath in different applications; tradeoffs in the parameters influence total performance and throughput – for example, in 802.11a, the guard period Δ may be set to 0.4μseconds in environments without significant multipath, and this can increase throughput by up to 11%.

Before data is modulated onto subcarriers, it may be “scrambled” in an effort to reduce long runs of 0’s and 1’s - this is done to improve the accuracy of detecting values from the resulting analog signal. A similar protocol design involves interleaving and coding to reduce the effect of noise in the channel. Interleaving attempts to move adjacent bits in the data stream to be further away in time or frequency space in the resulting waveform. Coding adds additional redundancy to allow the recovery of missing information. By both spreading information out across time/space and adding redundancy, it is possible to reduce the impact of bursts of errors or noise in the channel. Again, both interleaving and coding operate on “bits” rather than “symbols”.

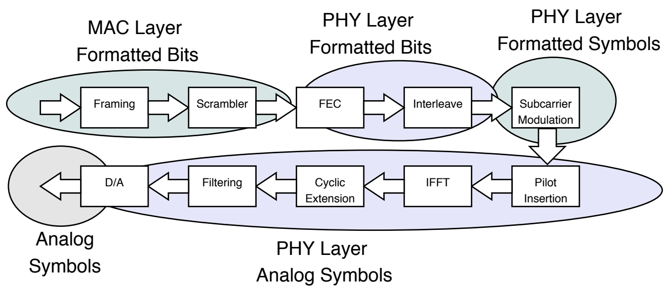

Fig. 3. Sample processing flow for OFDM waveform

Beyond the physical layer considerations, applications using OFDM can be broken into “continuous” and “frame” based protocols. A continuous protocol, such as digital audio broadcasting (DAB), streams data continuously with interspersed information used to synchronize receivers; such protocols are usually received by a number of stations and a fixed (and conservative) physical protocol is assumed. A frame based protocol, such as 802.11a/g prepends a “preamble” prior to the data payload; the preamble is used to synchronize clocks on the receiver, estimate channel conditions and to indicate the modulation used by the data payload; that payload may have different encodings based on the intended receiver or the number of retransmissions.

All of the processing steps can be visually represented by a processing data flow diagram such as shown in Figure 3. This diagram is particular to the 801.11a/g use of the OFDM waveform. The data transmitted by the user undergoes a series of transformations; initially, the user data is framed (adding control information) and then scrambled. That data is then expanded by adding forward error correction and the resulting data is then interleaved. The expanded data is then converted from “bits” to “symbols” as dictated by the subcarrier modulation being used. The resulting data is then converted to fixed point “analog values” that will be processed by the digital-to-analog (D/A) converters. The total volume of data increases as it moves along each processing step. For example, in 802.11a, to transmit data in a 10MHz band, the A/D converter must be provided at least 20 million samples per second (Msps); each “sample” is a complex number composed of two values that are typically represented as 16-bit quantity, for a total of 80MBytes/second at the last stage of the processing

```
src0 = gr.sig_source_f (sample_rate,
             gr.GR_SIN_WAVE, 650, ampl)
dst = audio.sink (sample_rate,
                  options.audio_output,
                  options.ok_to_block)
self.connect (src0, (dst, 0))
```

Fig. 4. Sample GNU Radio code snippet

pipeline. Meanwhile, the data entering the processing pipeline may vary in rate from 6Mbit/s (0.75MByte/s) to 54Mbit/s (6Mbyte/s). One of the challenges in designing a flexible software radio interface is determining on which processing element different data representations will be manipulated.

B. The Software In Software Defined Radios

Radio processing pipelines such as that shown in Figure 3 are more than simply illustrations of the processing for common radio waveforms - they also represent the most common software architecture used to implement software radios.

SimuLink [6] is a rapid-prototyping environment that can be used to simulate and implement a radio processing chain. SimuLink directly implements a "dataflow pipeline" between different processing blocks, effectively realizing the processing pipeline described in the prior section. Blocks are interconnected using "edges" that are actually implemented using wires or queues implemented using memory. Since SimuLink is built on top of Matlab, the common data structure within SimuLink is the vector or array; data is transported between computational blocks as vectors of samples.

Additional tools can translate a SimuLink design into a hardware specification for a processing pipeline; for example, when SimuLink is combined with tools such as Xilinx System Generator, designers can go from simulating to evaluating a radio processing pipeline very quickly. Because of these benefits, Simulink is used in many SDR implementations in the literature [7], [8]. In most of those designs, different "configurations" of a radio processing chain are typically represented by an increased number of processing blocks in the RF chain; this occurs because of the pipeline model underlying SimuLink and because most physical systems used to implement software defined radios cannot be dynamically reconfigured when new components are needed or older components are no longer needed.

An alternative software development environment is GNU Radio, an open source development environment for software defined radio. GNU Radio also uses a pipeline processing model, but uses the Python scripting language as a programable configuration language for the pipeline; the sample code snippet in Figure 4 creates a continuous sine wave tone and an "audio sink" (*i.e.,* a sound card) and connects the sound source to one channel of the audio sink. The GNU Radio framework is based on "blocks" that process *streams* of data; each "GNU radio block" is designed to process a varying number of samples above some specified minimum (*e.g.,* an FFT module may can specify that it must have at least 64 samples, but may be expected to process more than that minimum – it would do so in increments of 64 if that was the size of the FFT). A number of different waveforms, including HD-TV reception, GSM and BPSK and so on have been implemented using the GNU Radio framework.

Although specific details are not available, the Vanu Software Radio framework [9] appears be structured in a similar manner to the GNU Radio framework. The software is composed of the Sprockit middleware layer that implements signal processing blocks and the Radio Definition Language (RDL) that defines the structure of the software radio or composition of blocks to form a complete radio chain. As with the GNU software, the *stream* abstraction appears to be central to the design of Vanu software stack.

C. Software Radio Hardware Platforms

The hardware platforms needed to implement a software defined radio depend dramatically on the desired bandwidth, computational demands of the processing pipeline and system architecture. There are three main variants, depending both on which information is being communicated to a device as well as the system architecture. The first variant is "pure" software defined radio where all computational processing is performed by a CPU and complex I/Q samples are sent to an A/D converter for transmission. Such a system is limited by the I/O device connecting the CPU and the A/D converters. Current interconnects support a range of throughput; the PCI bus standard supports a nominal 133MByte/s throughput, while the USB 2.0 standard supports a nominal 60MByte/s. In practice, PCI interfaces can deliver in excess of 100MByte/s in either direction (CPU→A/D or A/D→CPU), although the throughput is reduced when data is moved in both directions or when short message sequences are used. USB interfaces typically provide 20-40MByte/s of delivered throughput depending on a range of factors including the chipset implementing the USB protocol. Assuming a 14-bit A/D converter (a common configuration) using complex samples (I/Q), a 100MByte/s channel could sample at most a 12MHz bandwidth; a 20MByte/s connection could sample a 2MHz bandwidth. The next-generation I/O standard, PCI-Express, transfers data in scalable "lanes" - each "lane" can transfer 250MByte/s of data, or enough for a $\approx$30MHz bandwidth using such raw I/Q samples.

The second platform choice combines a programable digital signal processor (DSP) with A/D hardware; this allows the host processor to process and transmit "bits"' to the DSP using the relatively low-bandwidth connections available and then have the DSP transmit the high-rate data to the A/D converters using an internal bus. The third organization uses an FPGA for essentially the same task. Both of these platforms also benefit from the computation acceleration offered by DSP's and FPGA's – however, we later show that modern CPU's can (almost) keep up with a wide-band PHY such as 802.11g.

III. An OFDM Software Radio

We had two goals in building an OFDM-based software radio. The first reason was that one goal of one of our NSF funded projects (NSF Project #0435297 - "NeTS:ProWiN: Programmable Radio Platforms for Highly Dynamic Networks") was to develop a software radio infrastructure suitable for experimentation and classroom use and we needed to provide sample waveforms and development paths for that radio; the prototype hardware is pictured in Figure 6 and its capabilities are described later.

The second, and more compelling, reason was that we are developing a very low-latency multi-hop networking layer that will be based on an OFDM waveform. We propose to reduce the latency of mesh networks by adopting a modified *cut through* or *wormhole* networking approach [10]. A similar proposal was recently made by Ramanathan [11] in a keynote address to the MOBICOMM conference. Our design is based on spatially distributed and coordinated temporal and frequency diversity operating on the level of symbol frames. This is similar to radio stations that employ a "translator network" that rebroadcast transmitters signal. Similarly, many first-responder networks employ sophisticated relay networks to increase the range of emergency telecommunications. These networks use static frequency allocations to relay traffic; we intend to extend this model to include mechanisms that allow more efficient spectral re-use and traffic engineering.

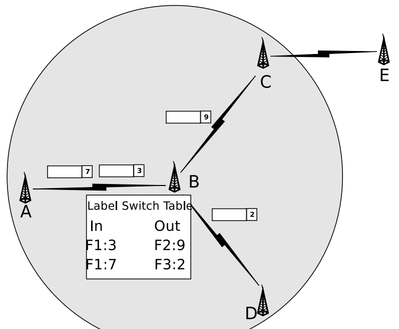

Fig. 5. Schematic diagram of a wireless label switch. Individual nodes in the multi-hop mesh network communicate over specific frequency bands of varying bandwidth.

We propose to adopt the network management and traffic engineering mechanisms implemented in Multiprotocol Label Switching (MPLS) and Generalized MPLS (GMPLS) to establish "label switch paths" (LSPs); those LSPs can operate on a variety of "labels" to reduce the routing and switching delay. For example, we could imagine a frequency range as being a connection over which labeled frames would be transmitted, as illustrated in Figure 5. Each frame is preceded by a "label" and the relay node implements a *wireless label switch*. In this diagram, the incoming label (indicated by the number preceding a frame) contains sufficient information to rapidly direct the remainder of the frame on the out-going connection. In this example, frames received on frequency 1 with label #3 are retransmitted on frequency 2 with label #9, and frames with label #7 are transmitted on frequency 3 with label #2.

There are 6 requirements for a PHY layer for such a radio design:

1) The PHY needs to be able to simultaneously transmit multiple "streams" to different receivers (corresponding to connections between nodes in Figure 5),
2) the PHY needs to be able allocate varying amounts of bandwidth to the different "streams",
3) the individual streams may use different modulations and coding rates,
4) the individual streams can "stop" and "start" at varying times,
5) the PHY should be able to operate over a sizable bandwidth,
6) and, the system should be practical to build.

Some of these design constraints are similar to those developed for 802.16-2004 and 802.16e [12], and the sum of these requirements place many requirements on

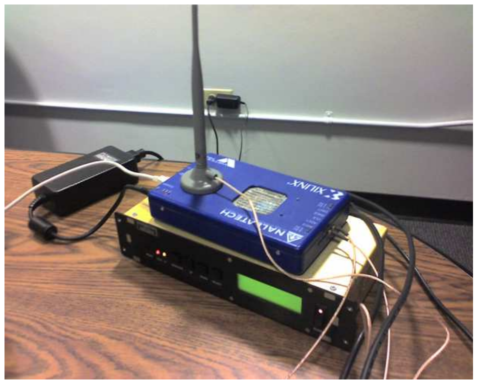

Fig. 6. Software Radio Platform

OFDM Parameter	Allowable Values
sample rate	80 MHz
IFFT size	1-256
subcarrier spacing	.3125 MHz
OFMD symbol period	3.2 μs
subcarrier modulation	BPSK, QPSK, 16-QAM, 64-QAM
guard interval	0-3.2 μs

TABLE I
OFDM MODULATOR PARAMETERS

the software radio design. Our design depends on having a radio processing pipeline for each "stream" and we felt that significant processing should be done by the host processor to simplify development – it also allows us to "virtualize" the highly variable parts of the processing pipelines on the CPU rather than attempting such fine control using FPGA resources.

We now describe the resulting interface and software radio architecture for the transmit path of this system; we then demonstrate the system functionality by instantiating a single 802.11g radio stream. That PHY layer serves as a benchmark both for the ability to correctly transmit the OFDM waveform and to evaluate the performance tradeoffs of the SDR design. Our software layer cannot use current software radio frameworks such as GNU Radio; we follow in §IV with observations on why this is the case and why up-coming changes in GNU Radio (and similar systems) are needed.

A. OFDM Modulator Design

Our software defined radio platform is shown is composed of a Xilinx XtremeDSP development kit manufactured by Nallatech and a 2.4GHz radio transceiver manufactured by Fidelity Comtech. The XtremeDSP development kit features a Nallatech Benadda Pro PCI/USB card including a Virtex II Pro FPGA, A/D and D/A converters. Although the figure shows the device in its USB configuration, we typically use it as a PCI card in a standard PC.

In order to implement an OFDM waveform on our platform, we had to first partition the design between the host processor and the FPGA. Our base FPGA design simply transfers samples from the PCI bus to the D/A converters on the card. There is enough space remaining on the FPGA to implement significant DSP pipelines. Thus we can implement all processing on the host, all processing on the FPGA, or choose a hybrid design somewhere in between.

We believe that the correct partitioning is to place bit manipulation operations such as framing, coding, interleaving and subcarrier assignment in host software while subcarrier modulation, the IFFT operation and guard interval insertion are performed on the FPGA. To retain software control over these functions, the OFDM modulation on the FPGA is kept as general and configurable as possible. This partitioning puts enough functionality on the FPGA to significantly reduce computation and I/O subsystem requirements but only places functions on the FPGA that tend to be universal across OFDM protocols. For example, different protocols may have very different data framing, error coding, and preamble schemes, but they will all eventually divide frames into subcarriers and run an IFFT on them. To see why the I/O requirements are reduced, note that the input to the subcarrier modulators is as small as one bit for BPSK or as large as six bits for 64-QAM. However, the output of the IFFT is a pair of 16 bit numbers. We send unmodulated subcarriers to the FPGA as bytes and the I/O requirements are reduced by a factor of four.

A summary of the fixed and configurable parameters of our OFDM modulator is shown in table I. We choose a 256 point IFFT and a sample rate of 80MHz for our design. Although the sample rate can be changed by altering the FPGA design, it is viewed as fixed by software. We picked these values so that the subcarrier spacing of our design is compatible with 802.11a receivers. The 256 subcarriers are individually modulated before they are sent to the inputs of the IFFT. A subcarrier can be disabled or modulated using BPSK, QPSK, 16-QAM, or 64-QAM. By enabling or disabling subcarrier modulators, software can effectively reconfigure the IFFT size to be anywhere between one and 256. The length of the cyclic extension for the guard interval insertion step can be configured to be between 0 and 100 percent of the OFDM symbol duration.

Subcarrier bits and configuration data are sent from software to hardware using a packet-like format. Packets

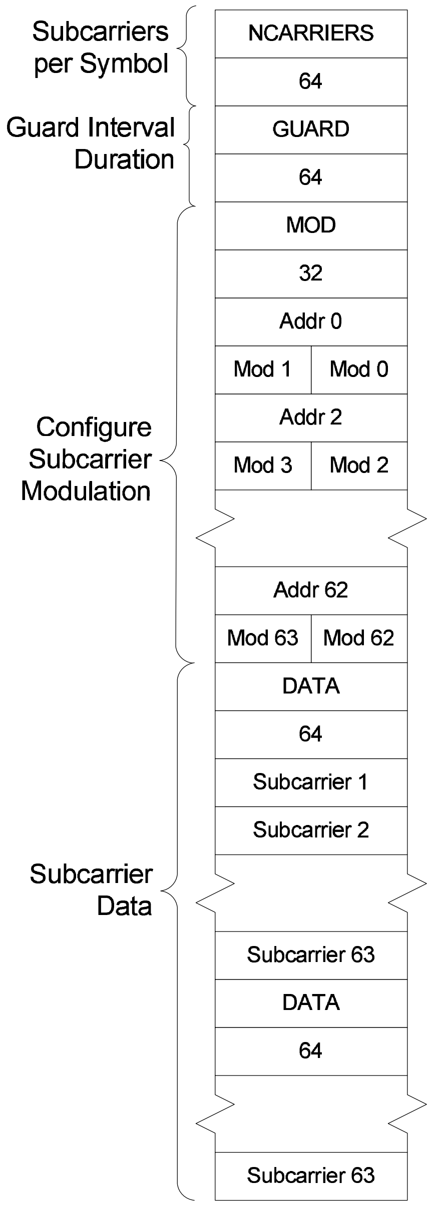

Fig. 7. Sample packet sent to hardware OFDM modulator

are subdivided to contain zero or more control blocks and zero or more data blocks. A data block starts with a block identifier followed by block specific data. There are four block types.

The NCARRIERS block tells the hardware how many of the IFFT inputs software will use. The block identifier is followed by the number IFFT inputs. Specifically, this variable tells the hardware how many subcarriers will be consumed by the subcarrier modulation unit during each OFDM symbol period.

The MOD block is used to configure the individual subcarrier modulators. The block identifier is followed by the number of data items contained in the block. Each data item, consisting of an address byte and a configuration byte, configures two subcarrier modulators. The address byte gives the address of the first modulator to be configured and the configuration byte contains 4 bits of configuration information for each subcarrier. The control bits enable or disable the subcarrier and also select the subcarrier modulation scheme to be used with that subcarriers. If a subcarrier is disabled, a zero is inserted into the corresponding IFFT input and no subcarrier data will be consumed for that subcarrier.

The GUARD block is used to configure the size of the guard interval. The block identifier is followed by the size of the guard interval.

The DATA block contains subcarrier bits to be modulated into OFDM symbols. The block identifier is followed by the number of subcarriers in the block. The format of the subcarrier data byte is defined by the modulation to be used and is generally just the appropriate number of bits for the modulation type.

Figure 7 shows an example input packet. The first part of this packet contains some control blocks to configure the modulator. Following the control blocks are two data blocks that will be modulated to form two OFDM symbols. It is not always necessary to include control blocks with data blocks. If the OFDM modulator is already configured appropriately, only the data blocks need to be sent. Similarly, control blocks do not need to be accompanied by data blocks.

B. An Example IEEE 802.11a/g Implementation

Our FPGA design has been tested by writing a software 802.11a/g implementation for it in C. Using this software, we are able to transmit 802.11g packets at 2.4GHz and receive them over the air using a vector signal analyzer as well as off the shelf 802.11g network adapters. An 802.11a/g frame exercises all the features of our design. It requires subcarriers to be reconfigured in the middle of a frame (between the preamble and payload), it requires that subcarriers are modulated differently within a single OFDM symbol (pilots subcarriers vs. data subcarriers), it requires the guard interval to be reconfigured in the middle of a frame (again, between preamble and payload), and it requires only a portion of the 256 subcarriers available.

1) The IEEE 802.11a Physical Layer: The IEEE 802.11a physical layer is specified in [13] for the 5GHz band. In [14], this physical layer was extended to the 2.4GHz band as part of 802.11g. This OFDM specification defines OFDM modulation parameters such as subcarrier spacing, subcarrier modulation, FFT width, and guard interval duration. It also defines the framing, scrambling, forward error correction coding and interleaving operations that are performed before OFDM modulation. These parameters and operations are combined to provide the eight data rate choices for standard 802.11g. Table II lists the timing related PHY parameters.

Figure 3 shows a block diagram of an 802.11a modulator. Data from the MAC layer is first padded with zero bits to aid coding and to ensure that the number of bits to be modulated is a multiple of the number of data bits per OFDM symbol. These bits are then scrambled by a simple polynomial to randomize the data. At the receiver, the same function is used to descramble the data. Scrambled data is subsequently encoded using a 1/2 rate convolutional code, generating two bits of output for every input bit. To achieve rates higher than 1/2, a puncturing step removes some of the coded bits from the data stream. Puncturing is reversed at the receiver by inserting zeros for the missing bits. After puncturing, an interleaving step ensures that adjacent bits are mapped onto nonadjacent subcarriers. The bits are then ready to be modulated.

The 802.11a standard defines an OFDM modulation like that described in section II. The scrambled, coded, interleaved bits are divided up into OFDM symbols of 48 subcarriers where 1, 2, 4 or 6 bits are mapped to each subcarrier depending on the modulation used. These subcarriers are individually modulated resulting in one complex number per subcarrier. Four pilot tones and 12 zero subcarriers are added for a total of 64 subcarriers per OFDM symbol. These vectors of 64 subcarriers are the input to the IFFT operation and 64 complex time domain samples are the output. The guard interval is added by copying the last 16 samples of each IFFT output to the start of the output giving 80 complex time domain samples per OFDM symbol.

Before transmitting the frame, two pieces of information are added. To help with signal detection, synchronization, and channel estimation, a $16\mu s$ preamble begins each frame. The preamble consists of 10 short training symbols and two long training symbols. Following the preamble is the signal symbol encoding the length of the data portion of the frame and the data rate at which the data is encoded. The signal symbol is always modulated using the 6Mbit/sec rate.

2) A Software 802.11a Transmitter: It was shown in [15] that there are smart ways to implement a software 802.11a transmitter such that the performance of the system is much better than a straightforward implementation of the processing blocks. For example, the convolutional coding, puncturing and interleaving steps can be combined into one step to avoid unnecessary data copies between processing elements. Our implementation uses similar techniques. We also precompute as much information as possible so that operations such as scrambling and convolutional coding become fast table lookups. For the IFFT operation, we use FFTW [16], a highly optimized software FFT implementation. The output of the software transmitter is a stream of complex I and Q samples representing the preamble symbols, signal symbol and data symbols of the frame sampled at 20MHz.

parameter	**value**
Number of FFT Points	64
Number of Data Subcarriers	48
Number of Pilot Subcarriers	4
Sample Rate	20MHz
FFT Symbol Period	3.2 μs
Guard Interval Duration	0.8 μs
OFDM Symbol Duration	4.0 μs

TABLE II
FIXED IEEE 802.11A OFDM PARAMETERS

required rate	**software rate**
6	8.0
9	14.0
12	18.1
18	24.5
24	28.4
36	34.9
48	39.1
54	40.5

TABLE III
REQUIRED RATE VS. ACHIEVED RATE FOR A SOFTWARE IMPLEMENTATION OF A 802.11A MODULATOR (1500 BYTE PACKETS, 2.2GHZ AMD ATHLON 64 PROCESSOR)

Table III shows the performance of our software implementation on a modest desktop computer. The required rate column shows the 802.11a rate at which the data is being encoded and the software rate column shows the rate achieved by the software when processing 1500 byte packets.

In table IV, the performance of the software without the OFDM modulation step (i.e. subcarrier modulation, IFFT, guard insertion) is shown. This is the implementation used with our FPGA modulator design.

IV. REFLECTIONS AND EVALUATION OF DESIGN

Although the stream abstraction is well suited to a "layered" software approach [17], there are complications arising from the software abstractions presented by software radio systems such as GNU Radio. Chief among these is that the stream abstraction complicates

required rate	software rate
6	43.2
9	49.7
12	49.6
18	55.6
24	54.2
36	59.2
48	60.5
54	60.0

TABLE IV
REQUIRED RATE VS. ACHIEVED RATE FOR A SOFTWARE IMPLEMENTATION OF A 802.11A MODULATOR WITHOUT SUBCARRIER MODULATION, IFFT, OR GUARD INTERVAL INSERTION (1500 BYTE PACKETS, 2.2GHZ AMD ATHLON 64 PROCESSOR)

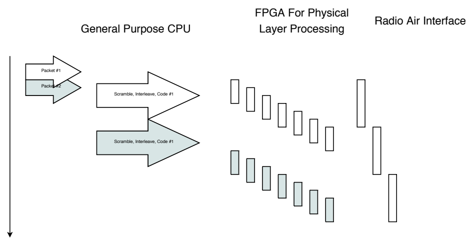
General Purpose CPU
FPGA For Physical Layer Processing
Radio Air Interface
Packet #1
Packet #2
Scramble, Interleave, Code #1
Scramble, Interleave, Code #1

Fig. 8. The lack of scheduling in a stream-based software radio frame complicates the synchronization of near simultaneous events. This diagram gives a schematic illustration of the process of encoding and transmitting two near-simultaneous packets. Since data transfer is fastest within the host CPU, the queue length and delays increases as the data moves from the host to the FPGA and then to the (much slower) air interface.

the synchronization of different layers in the protocol stack; this is particularly pressing in protocols that have tight "feedback loops". An example of this is any packet or frame based protocol (such as 802.11) that uses a "preamble" in a specific format to indicate how the remainder of the packet should be processed – such a stream of "samples" cannot be processed to an arbitrary point since their proper decoding depends on data at the beginning of the packet. Some number of samples at the start of the stream must be completely processed and decoded before the processing pipeline can begin decoding the rest of the stream. This dependence is an example of a "feedback loop" during the decoding process. There are similar dependencies when transmitting a packet - for example, the preamble of 802.11a/g packets must be transmitted using a well known modulation scheme and the packet contents can be transmitted using any of a number of modulation schemes; it is currently difficult to express such transformations in a purely stream-based software radio framework.

A more complex example occurs in OFDM waveforms when using different "bundles" of subcarriers to carry separate streams of information, which is one of the requirements we faced in our OFDM software radio. Software packages that focus on continuous streams of data make it difficult to coordinate timing. For example, assume we are processing "packets" that arrive on two interfaces almost simultaneously; Figure 8 shows a schematic timeline (along the vertical axis) for three components of an SDR - the host processor, and FPGA and the "air interface". Both packets are received almost simultaneously on their respective interfaces but only one can be selected and processed by the radio pipeline - that processed packet is expanded to a longer sequence of data that is then sent to the FPGA. The host processor is then free to process the second packet and then start transferring the data to the FPGA. However, the air interface is already engaged in modulating & transmitting the buffered data from the first packet. Thus, although both packets were intended to be transmitted almost simultaneously, the "stream abstraction" in the software precludes simultaneous transmission. The actual impact of these problems depends on the depth of buffering between the host and FPGA components, the number of streams and how scheduling is implemented in the software radio layer; none the less, the use of multiple output streams (or any PHY that depends on precise timing control) greatly complicates the traditional software architecture of radio pipelines.

An alternative approach is to used a frame based software model. In this model, all data, from the packet received from the MAC layer to the time domain samples about to be sent to the radio, is represented in software as a collection of data and associated metadata. The metadata gives configuration information to processing blocks as data flows through the processing pipeline. Any given function in the data processing path can reconfigure itself or even ignore portions of the input data based on the metadata. We use the frame based approach in this work. The GNU Radio framework is also evolving in this direction [18]. Our expectation is that we will be able to adapt our OFDM layer to use the new Gnu Radio "mblocks" mechanism that essentially combines a frame based approach.

V. RELATED WORK

In this paper, we build upon a broad range of prior work in the area of FPGA-based software defined radios. Work by Cummings in 1999 *et al* [19] provides some initial guidance on how FPGAs might be applied to the production of software defined radios. Lin *et al* [20] present a special purpose hardware design for software radio called SODA. One of the example protocols implemented is 802.11a. Meeuwsen *et al* [21] claim to be the first full-rate software implementation of a 802.11a modulator. The hardware platform is an Asynchronous Array of simple Processors (AsAP). They argue against the use of ASICs, however base their design on special purpose hardware. Tang *et al* [15] implement a full-rate software 802.11a transmitter on a DSP. They point out some 802.11a specific tricks that can be used to parallelize and concatenate some functional blocks in the modulator. Recent work by Di Stenfano *et al* [22] explores on the development of an IEEE 802.11 and ZigBee SDR receiver. Coulton and Carline [23], [24] present an 802.11a implementation on an FPGA that is partitioned to take advantage of partial reconfiguration of the FPGA. Instead of including all the different coding and modulation configurations on the hardware, parts of the design are reconfigured at runtime when requirements change. Harada *et al* [25] and Shono *et al* [26] both describe SDR implementations of 802.11 on FPGA.

Tarid *et al* [27] implements a fixed 24Mbit/s OFDM modem on a DSP system consisting of a PC plus PCI DSP card. Gifford, Kleider and Chuprun [28] studies the performance of a 16-bit fixed point implementation of an OFDM system written in ANSI C. The software is used in a simulation environment to determine the quantization error introduced in a fixed point system and its impact on achievable BER performance. Work by Dick *et al* [7] and later by Jamieson *et al* [8] demonstrate that an OFDM transceiver can be implemented in a FPGA. Both of these groups used Simulink to design an implementation of a 802.11a transceiver.

Reves *et al* [29], [30] consider some of the performance implications that arise when FPGAs are used to implement SDRs. They propose a middleware and abstraction layer to offer flexibility and simplicity for such design. What differentiates our work from these previous efforts is the focus on architectural issues relating to the design of an FPGA-based SDR.

VI. Conclusions

Our original goal was to develop a flexible OFDM waveform that allow spectrum agility, dynamic spectrum management and multiple use of a single radio for multiple point-to-point streams. The challenge of building such software has highlighted some of the issues in current software radio frameworks. Our resulting design places most of the "intelligence" in the host-based software; the FPGA device primarily serves as an accelerator for the FFT processing and as a means to reduce the I/O demands for a wide band signal.

Our performance comparisons indicate that current generations processors are on the cusp of providing sufficient bandwidth for a wide-band signal; however, data movement between the host and the A/D interfaces will likely preclude a "pure software" wideband radio for the next 2-3 years. The next generation of commodity processors, which are likely to contain vector-processing extensions (originally intended as graphics processors), will allow more flexible choices in software architecture of radio processing pipelines because concerns about "overhead" will largely vanish for most practical wide-band waveforms.

Interested parties can order the hardware components from Xilinx and Fidelity Comtech, Inc. That hardware can be used as a simple "fully software" SDR and can later be updated to use the OFDM waveform described in this paper. We are in the process of preparing the full software suite for release in Spring 2007.

Acknowledgements

We gratefully acknowledge the time and patience of Eric Blossom from the GNU Radio project in visiting the University of Colorado to explain the up-coming Gnu Radio framework and how it may impact our project.

This project is partially funded by NSF NeTS Project #0435297 - "NeTS:ProWiN: Programmable Radio Platforms for Highly Dynamic Networks" and "NeTS-FIND: Radio Wormholes for Wireless Label Switched Mesh Networks". Resources for this project are provided by NSF Project #0454404 - "CRI: Wireless Internet Building Blocks for Research, Policy, and Education" and NSF Project #0435452 - "NeTS - ProWiN: A Programmable Wireless Platform For Spectral, Temporal and Spatial Spectrum Management".

References

[1] V. Bose. *Design and Implementation of Software Radios Using a General Purpose Processor*. PhD thesis, MIT, 1999.

[2] V. Bose, M. Ismert, M. Welborn, and J. Guttag. Virtual radios. *IEEE/JSAC Special Issue on Software Radios*, April 1999.

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

[4] O. Edfors, M. Sandell, J. van de Beek, D. Landstrom, and F. Sjoberg. An introduction to orthogonal frequency-division multiplexing. Technical report, Lulea University of Technology, Sept 1996.

[5] T. Keller and L. Hanzo. Adaptive multicarrier modulation: A convenient framewrok for time-frequency processing in wireless communications, 2000.

[6] The MathWorks. Mathworks simulink product website. HTTP: http://www.mathworks.com/products/simulink/, Nov 2006.

[7] C. Dick and F. Harris. Fpga implementation of an ofdm phy. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 1, pages 905–909Vol.1, 9-12 Nov. 2003.

[8] C. Jamieson, S. Melvin, and J. Ilow. Rapid prototyping hardware platforms for the development and testing of OFDM based communication systems. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference, 2005*, pages Vol., pp. 57– 62, 2005.

[9] J. Chapin and V. Bose. The vanu software radio system. In *2002 Software Defined Radio Technical Conference*, Nov 2002.

[10] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, Jan 1979.

[11] R. Ramanathan. Challenges: A radically new architeture for next generation mobile ad hoc networks. In *MOBICOM*, 2005. Notes From Keynote Address.

[12] H. Yaghoobi. Scalabel ofdma physical layer in 802.16 wirelessman. *Intel Technology Journal*, 8(3):201–212, 2004.

[13] IEEE 802.11a 1999. Wireless lan medium access control (mac) and physical layer specifications: High speed physical layer in the 5 ghz band, 1999.

[14] IEEE 802.11b 1999. Wireless lan medium access control (mac) and physical layer specifications: Further high speed physical layer extensions in the 2.4 ghz band, 2003.

[15] Y. Tang, L. Qian, and Y. Wang. Optimized software implementation of a full-rate ieee 802.11a compliant digital baseband transmitter on a digital signal processor. In *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, volume 4, page 5pp., 28 Nov.-2 Dec. 2005.

[16] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[17] J. Chapin and K. Allain. Waveforms for dynamic wireless networks: Is layering a good idea. In *2004 Software Defined Radio Technical Conference*, Nov 2004.

[18] G.D. Troxel et al. Adaptive dynamic radio open-source intelligent team (adroit): Cognitively-controlled collaboration among sdr nodes. In *First IEEE Workshop on Networking Technologies for Software Defined Radio (SDR) Networks*, September 2006.

[19] M. Cummings and S. Haruyama. FPGA in the software radio . In *IEEE Communications Magazine*, pages Vol. 37, pp.108–112, 1999.

[20] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. *isca*, 0:89–101, 2006.

[21] M.J. Meeuwsen, O. Sattari, and B.M. Baas. A full-rate software implementation of an ieee 802.11a compliant digital baseband transmitter. In *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 124–129, 2004.

[22] A. Di Stefano, G. Fiscelli, and C.G. Giaconia. An FPGA-Based Software Defined Radio Platform for the 2.4GHz ISM Band. In *Research in Microelectronics and Electronics 2006*, pages Vol., pp. 73– 76, 2006.

[23] P. Coulton and D. Carline. An sdr inspired design for the fpga implementation of 802.11a baseband system. In *Consumer Electronics, 2004 IEEE International Symposium on*, pages 470–475, Sept. 1-3, 2004.

[24] P. Coulton and D. Carline. Enabling fine grained hardware platforms for software defined radio. In *International Conference onInformation and Communication Technologies: From Theory to Applications*, pages Vol., pp. 313– 314, 2004.

[25] H. Harada. Software defined radio prototype for W-CDMA and IEEE802.11a wireless LAN . In *IEEE Vehicular Technology Conference*, pages Vol. 6, pp. 3919– 3924, 2004.

[26] Shono, Shirato, Shiba, Uehara, Araki, and Umehira. IEEE 802.11 wireless LAN implemented on software defined radio with hybrid programmable architecture. In *IEEE Transactions on Wireless Communications*, pages Vol.4, pp. 2299– 2308, 2005.

[27] M.F. Tariq, Y. Baltaci, T. Horseman, M. Butler, and A. Nix. Development of an ofdm based high speed wireless lan platform using the ti c6x dsp. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 1, pages 522–526, 28 April-2 May 2002.

[28] S. Gifford, J.E. Kleider, and S. Chuprun. Broadband ofdm using 16-bit precision on a sdr platform. In *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE*, volume 1, pages 180–184vol.1, 28-31 Oct. 2001.

[29] X. Reves, V. Marojevic, R. Ferrus, and A. Gelonch. The cost of an abstraction layer on FPGA devices for software radio applications. In *15th IEEE International Symposium onPersonal, Indoor and Mobile Radio Communications, 2004*, pages Vol. 3, pp. 1942– 1946, 2004.

[30] X. Reves, V. Marojevic, R. Ferrus, and A. Gelonch. FPGA's middleware for software defined radio applications. In *International Conference on Field Programmable Logic and Applications*, pages Vol., pp. 598– 601, 2005.