Demo: 802.11 a/g PHY implementation in Ziria, domain-specific language for wireless programming

Gordon Stewart1 Princeton
jsseven@cs.princeton.edu

Mahanth Gowda1 UIUC
gowda2@illinois.edu

Božidar Radunović Microsoft Research
bozidar@microsoft.com

Dimitrios Vytiniotis Microsoft Research
dimitris@microsoft.com

ABSTRACT
Software-defined radio (SDR) brings the flexibility of software to the domain of wireless protocol design, promising an ideal platform both for research and innovation and the rapid deployment of new protocols on existing hardware. However, existing SDR programming platforms require either careful hand-tuning of low-level code, negating many of the advantages of software, or are too slow to be useful in the real world.

In this demo we present Ziria, the first software-defined radio programming platform that is both easily programmable and performant. Ziria introduces a novel programming model tailored to wireless physical layer tasks and captures the inherent and important distinction between data and control paths in this domain. We show the capabilities of Ziria by demonstrating a real-time implementation of WiFi PHY running at 20 MHz.

1. INTRODUCTION
The past few years have witnessed tremendous innovation in the design and implementation of wireless protocols, both in industry and academia ([Sen et al.(2011)Sen, Choudhury, and Nelakuditi], [Li et al.(2011)], [Bansal et al.(2013)]). Much of the work has occurred at the physical (PHY) layer of the protocol stack, which manages the translation between radio hardware signals and protocol packets. The numerous new signal processing algorithms and novel coding schemes that have resulted—many of which were first implemented using software-defined radio (SDR) platforms—have greatly increased the efficiency of existing radio communication channels.

There are clear advantages to implementing novel protocol designs in software, such as ease of development, fast and cheap deployment, and a much shorter development cycle compared to hardware designs. For example, GnuRadio [Blossom(2004)], USR()—currently one of the most widely-used SDR platforms—is implemented in a combination of C++ and Python, making it easy to program and extend. Unfortunately, this extensibility and ease of use come at a cost: the initial versions of GnuRadio suffered from performance limitations compared to hardware PHYs (Schmid et al. Schmid et al.(2007) and Nychis et al., Nychis et al.(2009) reported delays of hundreds of µs in the GnuRadio Software chain). This limited GnuRadio’s utility for testing in environments where line-speed operation is critical.

This is not to say that no SDR architecture provides acceptable performance: Warp [Murphy et al.(2006)], Sabharwal, and Aazhang, Sora [Tan et al.(2009)], TI Keystone [TIS()] and USRP Series X [USR()] are all high-performance hardware–software SDR implementations. These platforms can meet tight timing deadlines, and thus provide real-time support for protocol designers wishing to test at line rate. However, they suffer from another problem, which has seriously limited their adoption: these platforms are difficult to program. FPGA-based platforms, such as WARP, require substantial digital design expertise. CPU and DSP-based platforms, such as Sora and TI Keystone, require the ability to write code that is highly tuned to the underlying processor’s architecture. For example, Sora relies heavily on externally created lookup tables for performance. Furthermore, different parts of the receiver are manually placed onto different cores in order to balance CPU load. Such optimizations are heavily hardware dependent and must be performed manually for each new architecture. Furthermore, both Sora and GnuRadio have limitations associated with state sharing and dynamic reconfiguration of the flow graph.

In this demo, we present a new language, Ziria, and corresponding programming model which close the gap between performance and flexibility. Ziria consists of three components: (1) a high-level domain-specific language for programming wireless physical layer protocols, (2) an execution model for CPU-based platforms, and (3) an optimizing compiler. The Ziria compiler is open-source and available for download from the following URL:

https://github.com/dimitriv/Ziria

We implement Ziria as a language, as opposed to a library in C or C++, in order to enable aggressive domain-specific optimization of high-level Ziria code. The optimizations currently performed by the Ziria compiler include automatic lookup table (LUT) generation—which generates lookup...
tables directly from high-level code—as well as automatic
vectorization and annotation-guided pipeline parallelization.

To show that Ziria can handle real signal processing code, we
demo a Ziria implementation of a WiFi PHY running on
Sora at 20 MHz. Ziria’s optimizing compiler produces code
that generously meets the timing limits imposed by the 20
MHz sampling rate. To the best of our knowledge, we are the
first to present a high-level programming platform that can
implement a real-time 20 MHz PHY on a general-purpose
CPU while meeting timing constraints.

2. ZIRIA BY EXAMPLE

In this section we give a high-level overview of the main
components of the Ziria language. We illustrate the main
components with code excerpts from our 802.11 PHY imple-
mentation.

2.1 Language Overview

Expressions.

The Ziria language consists of two main parts. The first
is an expression language used to encode basic imperative
calculations. This language is a mix of Matlab and C designed
to make the right trade-off between programmability and
efficiency. It is a strongly typed language, which allows us
to simplify memory management. It also supports array
operations, like Matlab, which allows for efficient mapping
to SSE vector instructions. An example of a code in the
expression language is given inside the execute keyword in
Listing 1.

Computations.

Ziria’s computation language is designed to capture the
control flow of a PHY program and map it to an efficient
execution model. It consists of stream transformers, which
map values from input to output streams and which never ter-
minate, and stream computers, which can halt with a return
value. Composition of stream transformers and computers
is depicted in Figure 1. Many of the standard wireless PHY
blocks are stream transformers. For example, a scrambler
reads an input bit, XORs it with internal state, writes the
result to the output, and updates its internal state. FFT
is also a stream transformer. In the case of WiFi, it reads
64 complex numbers, performs FFT on them and writes 64
complex numbers to the output stream.

Stream computers also map stream inputs to stream out-
puts, but additionally may halt with a control value. In the
right-hand side of Figure 1, this control value is depicted
as the fat “Control” arrow connecting the computer to the
transformer. In Ziria, the control value is used to dynamical-
ly reconfigure the rest of the processing pipeline; in the figure,
this corresponds to switching from the “Computer” to the
“Transformer” in the lower right corner. After reconfiguration,
the inputs that originally flowed to the computer are routed
to the next component in the pipeline—in this case, the
“Transformer.” Both components output to the same stream.
This dynamic reconfiguration is called binding and it directly
reflects the control flow of many PHY-layer protocols, which
read a preamble or packet header from the input stream and
then reconfigure computation of the rest of the stream based
on data in the preamble.

Figure 1: Transformer-transformer composition (left); computer-transformer composition (right).

2.2 Examples: TX scrambler and RX pipeline

In order to illustrate the versatility of Ziria, this section
walks through two examples of real Ziria code. The first, a
signal scrambler, is typical of the kind of imperative code
that lives within the blocks of a Ziria pipeline. The second
example is a design of the pipeline of a 802.11a/g receiver.

Scrambler.

In signal processing domains, the purpose of a scrambler
is to XOR input data with a pseudorandom sequence. This
reduces the probability of sending data sequences that have
undesirable signal properties, such as all 1’s or all 0’s, over
the air (cf. Section 17.3.5.4 of [IEEE(1999)].

```
1 let comp scrambler() =
2   var scrmbl_st: arr[7] bit := {1,1,1,1,1,1,1};
3   tmp: bit; y: bit;
4   repeat seq{
5     x := take;
6     do {
7       tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
8       scrmbl_st[0:5] := scrmbl_st[1:6];
10      y := x ^ tmp;
11     emit y}
```

Listing 1: Scrambler function of WiFi 802.11a/g transmitter
in Ziria

The scrambler above, a let-bound Ziria computation taking
no arguments, is an example of a feedback shift register. The
scrambler’s body declares three local variables in lines 2
through 3, scrmbl_st, an array of 7 bits that gives the current
state of the shift register, and two one-bit references: tmp and
y. The scrambler takes a value from the input stream (line 7),
then performs an imperative computation that assigns tmp
the XOR of taps 3 and 0 in the shift register, shifts the
register state left by one, feeds tmp into position 6 of the
register, and finally returns the XOR of tmp and the input
bit x.

There are two interesting aspects to this code. First, be-
because both the WiFi standard and the code in the listing
operate on bit arrays, one can easily verify by inspection
that the listing code matches the definitions found in the
WiFi standard. This would be more difficult if we imple-
mented the scrambler directly in a more efficient fashion,
say by using an integer rather than a bitvector to store the
scrambler state, and by shifting instead of indexing into the
array. Second, we remark that, when compiled with the
Ziria compiler, the scrambler in Listing 1 compiles to quite
efficient code. In the context of our WiFi pipeline, the Ziria
compiler first automatically generates a lookup table for the

The first block, “Detect Carrier,” determines whether a WiFi transmitter is operating on the radio channel by looking for a known constant preamble sequence. Once carrier detection observes the preamble of a valid packet transmission, the pipeline enters a channel estimation phase, operating over the subsequent 160 input samples, in which it attempts to quantify physical effects such as multipath fading on the transmitted signal. The channel information is then used in the channel inversion block of the steady state of the pipeline for WiFi 802.11a/g receiver.

The lookup table implementation may be efficient, but it gives the reader no insight into the computation being performed. Nor is it easy to verify by inspection that this implementation meets the scrambler specification given in the WiFi standard.

The transmitter/receiver are fully integrated in the Windows networking stack, by exposing an Ethernet interface. In the demo, we will run a few real-world network applications on the top of our PHY. Also, we will present the Ziria code that implements the transmitter and the receiver and demonstrate how our compiler optimizations (such as auto-vectorization, LUT-ing, etc.) achieve over 5× improvement over a naive compilation strategy. The entire Ziria compiler toolchain has been open-sourced, with the hope that the entire SRIF community may benefit from it.

3. DESCRIPTION OF THE DEMO

In the demo we will show a WiFi wireless transmitter and receiver running at 20 MHz. Both the transmitter and the receiver are fully implemented in Ziria, apart from specialized implementations of IFFT, FFT, and the Viterbi decoder. We run both the transmitter and receiver on Sora SDR platforms. The transmitter/receiver are fully integrated in the Windows networking stack, by exposing an Ethernet interface. In the demo, we will run a few real-world network applications on the top of our PHY. Also, we will present the Ziria code that implements the transmitter and the receiver and demonstrate how our compiler optimizations (such as auto-vectorization, LUT-ing, etc.) achieve over 5× improvement over a naive compilation strategy. The entire Ziria compiler toolchain has been open-sourced, with the hope that the entire SRIF community may benefit from it.

4. REFERENCES


Listing 3: Top-level Ziria pipeline for WiFi 802.11a/g receiver

The Ziria code corresponding to the pipeline of Figure 2 is given in Listing 3. The structure of this code follows that of the block diagram quite closely. After reading from the input stream and downsampling (line 5), the first operation performed is carrier detection (line 8). In Ziria, the >> syntax denotes stream transformer composition, as in Figure 1. The computer detectCarrier() is an alias introduced with a let-binding (line 1) for another Ziria function, CCA() or clear channel assessment. Carrier detection is sequenced with the block operator with channelEstimation (line 9), a Ziria stream computer that returns a channel information value cInfo. The channel information cInfo is passed as an argument to invertChannel, a Ziria function defined at line 3. Finally, we stream the output of channel inversion to a Ziria computer that first decodes the packet header (t11aDecodePLCP(), line 11), and then decodes the packet payload (t11aDecode(pInfo), line 12).

3. DESCRIPTION OF THE DEMO

In the demo we will show a WiFi wireless transmitter and receiver running at 20 MHz. Both the transmitter and the receiver are fully implemented in Ziria, apart from specialized implementations of IFFT, FFT, and the Viterbi decoder. We run both the transmitter and receiver on Sora SDR platforms. The transmitter/receiver are fully integrated in the Windows networking stack, by exposing an Ethernet interface. In the demo, we will run a few real-world network applications on the top of our PHY. Also, we will present the Ziria code that implements the transmitter and the receiver and demonstrate how our compiler optimizations (such as auto-vectorization, LUT-ing, etc.) achieve over 5× improvement over a naive compilation strategy. The entire Ziria compiler toolchain has been open-sourced, with the hope that the entire SRIF community may benefit from it.

4. REFERENCES

[TIS(2002)] Texas Instruments TMS320TCI6616 Communications Infrastructure KeyStone SoC.

Listing 3: Top-level Ziria pipeline for WiFi 802.11a/g receiver

The Ziria code corresponding to the pipeline of Figure 2 is given in Listing 3. The structure of this code follows that of the block diagram quite closely. After reading from the input stream and downsampling (line 5), the first operation performed is carrier detection (line 8). In Ziria, the >> syntax denotes stream transformer composition, as in Figure 1. The computer detectCarrier() is an alias introduced with a let-binding (line 1) for another Ziria function, CCA() or clear channel assessment. Carrier detection is sequenced with the block operator with channelEstimation (line 9), a Ziria stream computer that returns a channel information value cInfo. The channel information cInfo is passed as an argument to invertChannel, a Ziria function defined at line 3. Finally, we stream the output of channel inversion to a Ziria computer that first decodes the packet header (t11aDecodePLCP(), line 11), and then decodes the packet payload (t11aDecode(pInfo), line 12).

3. DESCRIPTION OF THE DEMO

In the demo we will show a WiFi wireless transmitter and receiver running at 20 MHz. Both the transmitter and the receiver are fully implemented in Ziria, apart from specialized implementations of IFFT, FFT, and the Viterbi decoder. We run both the transmitter and receiver on Sora SDR platforms. The transmitter/receiver are fully integrated in the Windows networking stack, by exposing an Ethernet interface. In the demo, we will run a few real-world network applications on the top of our PHY. Also, we will present the Ziria code that implements the transmitter and the receiver and demonstrate how our compiler optimizations (such as auto-vectorization, LUT-ing, etc.) achieve over 5× improvement over a naive compilation strategy. The entire Ziria compiler toolchain has been open-sourced, with the hope that the entire SRIF community may benefit from it.

4. REFERENCES

[TIS(2002)] Texas Instruments TMS320TCI6616 Communications Infrastructure KeyStone SoC.


