Ziria: Language for Rapid Prototyping of Wireless PHY

Mahanth Gowda[1] UIUC
gowda2@illinois.edu

Gordon Stewart[1] Princeton
jsseven@cs.princeton.edu

Geoffrey Mainland[2] Drexel
mainland@cs.drexel.edu

Božidar Radunović
Microsoft Research
bozidar@microsoft.com

Dimitrios Vytiniotis
Microsoft Research
dimitris@microsoft.com

Doug Patterson
Microsoft
dougpatt@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 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 practically.

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. Ziria provides the capability of implementing a real-time WiFi PHY running at 20 MHz.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special Purpose and Application Based Systems—*Signal processing systems*

Keywords

Languages; Signal Processing; Wireless; Compilers

1. INTRODUCTION

The past few years have witnessed tremendous innovation in the design and implementation of wireless protocols, both in industry and academia ([1, 2]). 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 radio communication channels.

However, SDRs have not yet reached the mainstream of wireless research. The main impediment is implementation complexity. Wireless PHY algorithms have to be very fast and efficient to meet line rate requirements. Software has to process tens of millions of complex samples per second, which is equivalent to a processing rate of close to a gigabit per second. To cope with these speeds a programmer has to have a good understanding of the underlying SDR hardware architecture and the effects of her design choices on performance.

In order to design wireless algorithms, an SDR programmer has to have a profound understanding of signal processing and wireless networking, and in order to efficiently implement them she has to be an expert in computer architecture and system design. An average student finds it quite hard to master all these areas. Only very few research groups have found enough resources to pursue this line of research.

To illustrate this complexity we consider two popular SDR designs. The first example is FPGA-based platforms, such as Warp [3] and Lyrtech [4]. A typical state-of-the-art SDR programming tool for FPGA platforms is based on Matlab, Simulink and an FPGA compiler. One of the main benefits of this environment is a rich set of DSP building blocks (such as basic fixed-point arithmetics, FFT, encoders, decoders, etc.). Further, being integrated in Matlab, this environment provides a programmer with powerful tools for debugging input and output signals, and for simulating different real-world effects using readily available libraries. However, the programming model is essentially the same as hardware design in VHDL or Verilog. A programmer has to have a fundamental understanding of how FPGAs operate. She has to be aware of different propagation delays and their effects on the system performance. One misplaced delay element can easily lead to a week of debugging!

Recently, several SDR platforms, such as GnuRadio [5] and Sora [6], allow programmers to design SDR code on general-purpose CPU processors. The hardware part of the platform performs analog-to-digital and digital-to-analog conversion of the baseband signals and supplies these signals to a CPU, where a programmer can further process it as desired.

Signal processing on a CPU provides great flexibility and permits rapid experimentation, but often at the price of decreased performance. Writing fast, real-time PHY code on a CPU can be equally, if not more complex than programming FPGAs. In fact, most of the recent academic works on SDR do not even attempt to process signals in real-time. They

[1]This work was done during an internship with Microsoft Research.
[2]Part of the work was done while the author was with Microsoft Research.

MobiCom'14, Sep 07-11 2014, Maui, HI, USA.
ACM 978-1-4503-2783-1/14/09
http://dx.doi.org/10.1145/2639108.2642893.

only use SDR platforms to capture signals. Post-processing is later done offline at much smaller processing rates than required by PHY. This approach clearly precludes any real-world networking experiments on SDR platforms.

One of the first CPU-based SDR platforms that has managed to process wireless baseband signals at a line rate is Sora [6]. It has demonstrated interoperability with commodity WiFi cards. But this performance comes with increased complexity. The code for Sora has to be written in C/C++ and a programmer needs to manually optimize the code to achieve the required speeds. Sora has put substantial effort into designing a C++ template library that eases the programming pain. However, use of templates imposes several limitations, in particular on state sharing and dynamic reconfigurability of the dataflow graphs. It is worth noting that GnuRadio has a similar architecture based on C++ templates and similar limitations. We next overview the architecture of Ziria and explain how it addresses these issues.

2. ZIRIA

We give a high-level overview of main components of Ziria.

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 terminate, 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 outputs, 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 *dynamically 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

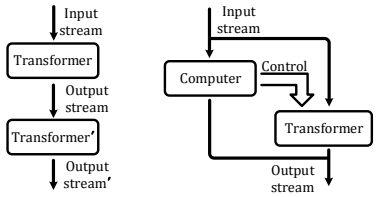

Figure 1: Transformer-transformer composition (left); computer-transformer composition (right).

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.

2.2 Optimizing Compiler

The Ziria optimizing compiler consists of two parts. It has a front-end which parses a Ziria program and stores it in an abstract representation, and then performs several optimization passes on it. It also has a back-end which compiles the optimized abstract representation into a low-level execution model optimized for a target architecture. Our current back-end targets a low-level execution model in C for the Sora SDR platform, but it can easily be extended to other platforms, such as GnuRadio.

There are several benefits of this approach when compared to the existing approaches:

Dynamic staging of the control graph: Ziria language for staging data flows allows for a dynamic reconfiguration of the control graph. This reconfiguration is very flexible, so for example a downstream component can reconfigure an upstream component or any other part of the graphs (for example using parameter passing and the repeat construct). This is in contrast with Sora and GnuRadio C++ templates that allow limited downstream reconfigurability using the multiplexing block.

No need for shared state: Ziria dynamic reconfigurability and parameter passing between blocks eliminate the need for share states. Instead, a state is passed explicitly between components and all the dependencies are checked at the compile time. This is in contrast with Sora and GnuRadio which heavily rely on shared states. One example of a negative consequence is that an unitialized state cannot be detected by the compiler and causes program misbehaviour.

Code optimization: Ziria compiler's optimization phase can optimize jointly the imperative part of the language and the data-flow graph. It can merge data-flow blocks, change input and output widths, inline expressions and unroll loops. One example of a useful code optimization are look-up tables. Ziria compiler can automatically identify blocks of code that operate on small bit inputs (e.g. scrambler and encoder) and convert them into lookup tables, which execute much faster on a CPU than bit operations. All this is in contrast to Sora and GnuRadio C++ template libraries where C compiler does not have an explicit understanding of the control flow graph and can perform far fewer optimizations. Lookup tables have to be implemented and tuned manually.

Overall, Ziria code is much more concise. For example, an implementation of a WiFi scrambler in Sora takes 90 lines of C++ code. The same code can be implemented in Ziria in 13 lines. It is easier to understand and has very similar performance to the hand-tuned Sora code.

3. EXAMPLES: TX SCRAMBLER

This section walks through an example code of a signal processing block (Scrambler) in Ziria.

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 [7]).

```
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];
9        scrmbl_st[6] := tmp;
10       y := x ^ tmp};
11     emit (y)}
```

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

Scrambler is a let-bound Ziria computation taking no arguments, is an example of a feedback shift register. The scrambler's body declares three local variables (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`. Scrambler **take**s a value from the input stream (line 5), 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. First, 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 implemented 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 scrambler and then vectorizes the code to operate on multiple inputs at a time. This gives a $14.8x$ speedup over the same code compiled without optimizations. As comparison, in the current Sora implementation, the same scrambler is defined as a hand-written lookup table, an excerpt of which is shown in Listing 1.

```
1  const unsigned char
2  SCRAMBLE_11A_LUT[SCRAMBLE_11A_LUT_SIZE] = {
3      0x00, 0x91, 0x22, 0xb3, 0x44, 0xd5, 0x66, 0xf7,
4      0x19, 0x88, 0x3b, 0xaa, 0x5d, 0xcc, 0x7f, 0xee,
5      //... 13 lines elided ...
6      0x87, 0x16, 0xa5, 0x34, 0xc3, 0x52, 0xe1, 0x70,};
```

Listing 2: Sora lookup table corresponding to the Ziria scrambler implementation in Listing 1

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.

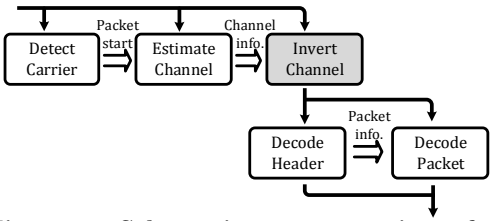

Figure 2: Schematic representation of a WiFi receiver. The shaded box is a transformer; the white boxes are computers.

4. WIFI IMPLEMENTATION

In the previous section, we saw an example of the design of a signal processing block in Ziria. Using many such building blocks, Figure 2 illustrates the high level block diagram of an 802.11a/g receiver in Ziria.

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 (in gray) to nullify these effects. This block feeds input data first to a block that decodes the packet header, then to a block that decodes the packet payload. Information from the header decoding phase such as the data rate is used to configure the packet decoding stage.

Both the transmitter and the receiver of a 20MHz WiFi are fully implemented in Ziria, apart from specialized implementations of IFFT, FFT, and the Viterbi decoder. Currently, the code compiles to Sora SDR platforms. The transmitter/receiver are fully integrated in the Windows networking stack, by exposing an Ethernet interface. We are able to run a few real-world network applications on the top of our PHY. Our compiler optimizations (such as auto-vectorization, LUTing, etc) achieve over $5x$ improvement over a naive compilation strategy. The entire Ziria compiler toolchain has been open-sourced [8], with the hope that the entire MOBICOM community may benefit from it.

5. REFERENCES

[1] S. Sen *et al.*, "No time to countdown: Migrating backoff to the frequency domain," in *MOBICOM*, 2011.

[2] T. Li *et al.*, "CRMA: Collision-resistant multiple access," in *MOBICOM*, 2011.

[3] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: a wireless open-access research platform," in *ESPC*, 2006.

[4] Nutaq, "Zeptosdr." `http://nutaq.com/en/products/zeptosdr`.

[5] E. Blossom, "GNURadio: tools for exploring the radio frequency spectrum," *Linux Journal*, vol. 2004, no. 122, p. 4, 2004.

[6] K. Tan *et al.*, "Sora: High performance software radio using general purpose multi-core processors," 2009.

[7] IEEE, "Part 11: Wireless LAN MAC and PHY specifications high-speed physical layer in the 5 GHz band," 1999.

[8] "Ziria." `http://research.microsoft.com/en-us/projects/ziria/`.