Ziria: A DSL for wireless systems programming

Mahanth Gowda[1]
UIUC
gowda2@illinois.edu

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

Geoffrey Mainland
Drexel University
mainland@cs.drexel.edu

Božidar Radunović
MSR Cambridge
bozidar@microsoft.com

Dimitrios Vytiniotis
MSR Cambridge
dimitris@microsoft.com

Cristina Luengo Agulló
Universitat Politècnica de Catalunya
cristinaluengoagullo@gmail.com

Abstract

Software-defined radio (SDR) brings the flexibility of software to wireless protocol design, promising an ideal platform for innovation and rapid protocol deployment. However, implementing modern wireless protocols on existing SDR platforms often requires careful hand-tuning of low-level code, which can undermine the advantages of software.

Ziria is a new domain-specific language (DSL) that offers programming abstractions suitable for wireless physical (PHY) layer tasks while emphasizing the pipeline reconfiguration aspects of PHY programming. The Ziria compiler implements a rich set of specialized optimizations, such as lookup table generation and pipeline fusion. We also offer a novel – due to pipeline reconfiguration – algorithm to optimize the data widths of computations in Ziria pipelines. We demonstrate the programming flexibility of Ziria and the performance of the generated code through a detailed evaluation of a line-rate Ziria WiFi 802.11a/g implementation that is on par and in many cases outperforms a hand-tuned state-of-the-art C++ implementation on commodity CPUs.

1. Introduction

The past few years have seen increased innovation in the design and implementation of wireless protocols, both in industry and academia (cf. [8, 28, 44]). Much of this work – especially at the physical (PHY) layer of the protocol stack, managing the translation between radio hardware signals and protocol packets – has been driven by the increased availability of *software-defined radio (SDR)* platforms. SDR platforms, unlike ASIC designs, enable wireless devices to be programmed by researchers and other end users. In industry, low-cost SDR platforms have paved the way for new business models to disrupt the traditionally conservative telecom sector [23, 34, 51].

In order to meet the processing requirements of wireless protocols, SDR platforms have traditionally been FPGA-based, and were therefore difficult to program and extend. Despite some recent efforts [35], these FPGA-based SDR platforms still haven't achieved wide-spread adoption. Recently, however, projects such as Sora [45] and USRP [50] have demonstrated that CPU-based platforms can also meet the processing demands of contemporary wireless standards. Sora, for example, was the first to achieve interoperability with commercial WiFi hardware while executing all signal processing on the CPU.

Writing realtime PHY code for a CPU, though easier than programming FPGAs, is still difficult. Wireless PHY algorithms have to process tens of millions of I/Q samples per second – equivalent to a rate of around one gigabit per second. To achieve such speed a programmer has to have a good understanding of the underlying hardware and the effects of design choices on performance. As evidence of the difficulty of the problem, the initial Sora WiFi implementation – the first to solve the problem in software – merited a best-paper award [45]; the first WiFi receiver for GNU Radio appeared only a year ago [11].

A PHY design typically consists of a pipeline of signal processing stages. So far, much of the effort has been devoted to generating efficient implementations (using advanced CPU features) of the digital signal processing (DSP) algorithms that sit within these pipelines – such as FFT, correlation, encoding, and decoding. Such implementations are developed by human experts [45] or are auto-generated from high-level algebraic specifications [40, 52].

[1] Co-primary Authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694368

However, wireless PHY programming is not only about generating fast DSP code and efficient matrix computations. It is also about expressing and correctly and efficiently compiling *pipeline reconfigurations* when system state (e.g. channel information, index of element in the LTE resource grid, coding and modulation rates) changes. Pipeline reconfigurations do appear in a PHY implementation and a programmer needs to maintain state that persists across the processing of multiple entities (header, resource blocks, packets, etc.), specify when a state changes, and how that change will affect the functionality of other blocks.

This pipeline reconfiguration aspect of PHY design is overlooked in previous work on SDR programming, yet it is of paramount importance. For example, the 4G/LTE standard – the state-of-the-art wireless cellular technology – contains over 400 pages of specifications defining the PHY layer alone [1–3]. The signal processing blocks used in LTE are mainly standard (e.g. FFT, turbo coding); most of the specification actually describes the control flow and state changes. The IEEE WiFi standard and typical textbooks give simplified block diagrams [25, Figure 118] that look like straightline dataflow composition. Unfortunately, these diagrams completely ignore pipeline reconfigurations. The full specification of state changes and pipeline reconfigurations is instead given in 90 pages of text [25] and is nontrivial to correctly and efficiently implement due to the delicate synchronization required between state changes and data processing.

Our key contribution is a new programming language, Ziria, specifically designed to expose high-level pipeline reconfiguration and control flow in PHY protocols without sacrificing the performance required for line-rate SDR. Ziria has a layered design. The main novelty is the higher-layer *computation* language for specifying and composing stream processing pipelines. Ziria also offers a lower-layer *expression* language for implementing DSP algorithms and data manipulation within pipeline components. This language is similar to those used by domain experts (C or Matlab), with a subset of features and low-level optimizations carefully selected to maintain expressiveness yet guarantee efficient compilation. The main benefits of a new domain-specific language over a library-based approach implemented in a general-purpose language are (a) domain-specific programming abstractions and a rich type system that allow programmers to express and confidently reason about state changes in their pipelines, and (b) support for aggressive optimizations on Ziria ASTs, which is crucial for compiling high-level programs to meet the tight performance constraints of modern PHYs on commodity CPUs. In this work we focus on CPUs because they are adopted by wide-spread SDR platforms [45, 50].

An important optimization in SDRs involves increasing the data widths of the components in a processing pipeline. For instance, although an encoder may naturally be defined to operate on single bit inputs, it will be much more efficient if implemented to operate on bytes. Wide data paths bring clear benefits: the ability to use SIMD instructions and wider lookup tables, efficient data manipulation, and more data locality. However, having to optimize data widths *manually* compromises ease of programming and flexibility. The acceptable data widths of a component may depend in complex ways on the rest of the pipeline (Section 3). As a consequence, a programmer needs to jointly optimize all the data widths in a pipeline, which is error-prone and tedious. Additionally, optimizing each component for the data widths of the specific pipeline in which it is used can lead to code that cannot easily be reused in other pipelines.

The Ziria compiler instead offers a *vectorization* transformation that automates this process. Thanks to this transformation, a component can be written once and reused – without any changes – in different pipelines or in different positions in the same PHY implementation with possibly different vectorizations. A well-typed pipeline modification does not require manual modifications in the constituent components. The main novelty of our vectorization transformation (compared to more traditional coarsening transformations of data paths [4, 48]) is that – for correctness – it must take into account pipeline reconfigurations (Section 3). In addition, our algorithm must jointly optimize the data path widths for *all* the components in a pipeline, which (we have empirically confirmed) can easily cause compilation times to blow up. To address this problem, we employ an idea from a distributed convex optimization framework [26] that, for a large class of utility functions, allows our vectorizer to make a local decision for each sub-computation in the pipeline without compromising global optimality. Local pruning dramatically reduces the number of vectorization candidates. As a consequence, full-featured WiFi transmit (TX) and receive (RX) PHY pipelines can compile (from Ziria to C) in 2-4 seconds.

In addition to vectorization, the Ziria compiler implements numerous other optimizations, including: automatic lookup table (LUT) creation, inlining, partial evaluation, memory management optimizations, fusion, a variant of static scheduling and others that are fundamental in achieving the desired processing speeds.

In summary, our contributions are:

- We present a new programming language, Ziria, with domain-specific abstractions well-suited for programming wireless SDRs with reconfigurable pipelines and a compilation scheme that supports limited coarse-grained pipeline parallelism and produces efficient executables for commodity CPUs (Section 2).
- We design and implement a novel *vectorization* algorithm to optimize the data widths of components in processing pipelines (Section 3). We measure the effects of this optimization and show that it also dramatically amplifies the effects of other optimizations (a combined $10\times-100\times$ speedup over baseline).
- To demonstrate the viability of Ziria for developing SDR applications, we implement a IEEE 802.11a/g PHY. Our

implementation meets the protocol *throughput* and *latency* constraints, its performance matches and often outperforms hand-tuned C++ code, and it is tested with live transmissions on Sora boards, delivering $< 2\%$ packet error rate on a nonrealtime OS (Section 5).

To the best of our knowledge, we are the first to present a *high-level* programming language that – in addition to being performant – can concisely capture and optimize for reconfigurations in PHY pipelines. Our compiler, testsuite, and WiFi implementation are publicly available [22].

2. Programming in Ziria

In this section we present the basic programming abstractions of Ziria and show how these abstractions help programmers compose complex pipelines such as those required for 802.11a/g. We then show how programmers familiar with imperative C-like languages can implement the basic processing blocks of those pipelines as imperative code in Ziria's expression language. A small number of signal processing concepts are introduced along the way.

2.1 Ziria stream programming abstractions

Ziria programs process *streams* of values. At a high level, a WiFi receiver is just one such computation that reads a stream of complex numbers (I/Q samples) from the A/D converter of a radio and outputs a stream of bytes corresponding to MAC-layer packets. Ziria programmers can create and compose together computations of two *types*.

Stream transformers are computations that resemble traditional stream processing "blocks" or "bricks" commonly found in existing SDR platforms like GnuRadio and Sora. They execute indefinitely by taking values from input streams and emitting values onto output streams, perhaps also maintaining some internal state. A *scrambler* block is a simple example: scramblers are used, e.g., in a WiFi transmitter, to XOR input data with a pseudorandom sequence in order to shape the transmitted signal.

Stream computers, on the other hand, are a novel concept in SDR programming. Like stream transformers, they take values from input streams and emit onto output streams. In contrast to stream transformers, they execute for a while, consuming input and producing output, but eventually *halt* and return an additional value, which we refer to as a *control value*. The WiFi packet header decoder is an example of a stream computer: it decodes the header of a packet in a streaming fashion but eventually halts, returning a *control value* containing coding and modulation parameters from the header, which can subsequently be used to decode the packet payload. The fact that computers continuously process and output data while computing the final return value allows downstream components to continuously use these outputs, reducing overall program latency.

The Ziria type system distinguishes between these two abstractions. It assigns type $(\mathsf{Zr}\ \mathsf{T}\ a\ b)$ to Ziria transformers that take inputs of type a and emit outputs of type b and type $(\mathsf{Zr}\ (\mathsf{C}\ c)\ a\ b)$ to Ziria computers that take inputs of type a and emit outputs of type b while eventually returning a control value of type c. Value types a, b, c are more conventional and include bit, integer and complex types of various widths, structures, arrays of statically known length, etc.

2.2 Composition on the control path

Control values returned by stream computers are *the only* Ziria mechanism for reconfiguring the processing pipeline. In our decoding example, the runtime – upon termination of the header decoder – will use the returned control value to initialize and configure the payload decoder and serve it the rest of the input stream. This is in sharp contrast to other SDR platforms [35, 46, 48], in which reconfiguration and initialization of different parts of the processing pipeline is achieved via shared global variables, asynchronous message passing, or has to be manually programmed in a low-level fashion with extra messages in data paths. These alternatives can hamper code maintainability and reusability.

In Ziria, this pattern of control flow is expressed using the `seq`-uence combinator (excerpt from our WiFi receiver):

```
seq { (h : HeaderInfo) ← DecodePLCP()
    ; Decode(h) }
```

This code runs the stream computer `DecodePLCP()` until it produces a control value `h`, of type `HeaderInfo`, and then switches to the stream computer `Decode(h)`. Because this sequencing expresses control flow in a program, we refer to the sequence operator as composition "on the control path." The Ziria type system ensures that in a sequence `seq { x ← c1; c2 }`, the component `c1` is indeed a stream computer, and not a transformer (`c2` may be either a stream computer or transformer). The (simplified) typing rule is:

$$\frac{\vdash \texttt{c1} : \mathsf{Zr}\ (\mathsf{C}\ c)\ a\ b \quad (\texttt{x}{:}c) \vdash \texttt{c2} : \mathsf{Zr}\ t\ a\ b}{\vdash \texttt{seq \{ x} \leftarrow \texttt{c1; c2 \}} : \mathsf{Zr}\ t\ a\ b}$$

where t is either T or $\mathsf{C}\ d$. Notice that both `c1` and `c2` take values of the same type (a) and emit values of the same type (b), albeit at different points in time.

Dynamic reconfiguration using `seq` directly reflects the control flow of many PHY-layer protocols. Another typical example is WiFi packet reception, in which the receiver first uses a sample preamble to estimate the characteristic of the communication channel and uses this estimation to invert the effects of the channel and decode the packet.

2.3 Composition on the data path

Whereas control path composition combines two components that take from the same input stream and emit to the same output stream (although at different points in time), Ziria also supports the more convential form of composition "on the data path": this is composition in which the stream output of one block becomes the stream input of another. For instance, in a WiFi transmitter the output stream of the CRC block is

piped to a scrambler, followed by an encoder. We express this composition with the >>> combinator. In Ziria code (again, an excerpt from our WiFi transmitter):

```
crc24(len) >>> scrambler() >>> encode12()
```

The general form of this combinator c1>>> c2 allows at most one of c1 and c2 to be a stream computer, in which case the whole c1>>> c2 becomes a computer, halting and returning back the return value of the computer component. If both components are transformers, then so is their composition. The (simplified) typing rule is:

$$\frac{t = t_1 \oplus t_2 \quad \vdash \texttt{c1} : \mathsf{Zr}\ t_1\ a\ b \quad \vdash \texttt{c2} : \mathsf{Zr}\ t_2\ b\ c}{\vdash \texttt{c1} >\!\!>\!\!> \texttt{c2} : \mathsf{Zr}\ t\ a\ c}$$

where $\mathsf{T} \oplus t = t \oplus \mathsf{T} = t$.[1] In addition, only one of c1 and c2 can have read-write access to shared variables, to guarantee race-freedom in a pipeline parallel execution of >>> composition.

2.4 Example: WiFi receiver pipeline

We now show how these abstractions fit together to form complex pipelines. Our example in this section is an implementation of a WiFi 802.11a/g receiver in Ziria, based on the Sora implementation [45].

```
1  let comp Decode(h : struct HeaderInfo) =
2    DemapLimit(0) >>>
3      if h.modulation == M_BPSK then
4        DemapBPSK() >>> DeinterleaveBPSK()
5      else if h.modulation == M_QPSK then
6        DemapQPSK() >>> DeinterleaveQPSK()
7      else ... -- QAM16, QAM64 cases
8    >>> Viterbi(h.coding, h.len*8 + 8)
9    >>> scrambler()
10 let comp detectSTS() = removeDC() >>> cca()
11 let comp receiveBits() =
12   seq { h ← DecodePLCP()
13       ; Decode(h) >>> check_crc(h.len)
14       }
15 let comp receiver() =
16   seq { det ← detectSTS()
17       ; params ← LTS(det.shift)
18       ; DataSymbol(det.shift) >>>
19           FFT() >>>
20           ChannelEqualization(params) >>>
21           PilotTrack() >>>
22           GetData()    >>>
23           receiveBits() }
```

Listing 1: Ziria WiFi 802.11a/g receiver pipeline

The main phases of the WiFi pipeline are: channel detection (line 16), channel estimation (line 17), and packet demodulation and decoding (lines 20 to 23). Lines 18 to 22 convert from the time into the frequency domain.

Channel detection determines if there is WiFi transmission by searching for a known preamble in the input stream. The first block, detectSTS, consists of a block that removes the signal DC component (removeDC), followed by the actual detection algorithm (clear channel assessment, cca). Detection returns fine-grained timing information, bound to det.

Channel estimation estimates physical effects (multipath fading) on the transmitted signal (block LTS). The channel estimate is returned as control value params.

OFDM demodulation converts the received data from the time to the frequency domain (lines 18 to 22), and the receiver removes the effects of the channel, as estimated in the previous phase, using ChannelEqualization followed by PilotTrack. Finally, the processed symbols are passed to receiveBits for demodulation and decoding.

Discussion. The WiFi example highlights key high-level features of the Ziria computation language. First, component composition is intuitive and code is well-structured, short, and guaranteed type-correct by the compiler. Second, configuration parameters are explicitly passed to and from components, and all reconfigurations – contrary to Sora – become explicit in the seq-structure of the programs.

2.5 Implementing processing blocks

Ziria is a two-layer language consisting of (i) a computation language and (ii) an imperative and by-and-large conventional language for computing with bits, integers, complex values, structures, arrays, etc. The most important syntactic forms of the language are given in Figure 1. We use c to denote computations and e, m to denote expressions from the imperative fragment. The details of the imperative fragment, which we omit, are unsurprising. As a notational convention, we use m for statements (e.g. y := y+1) and e for expressions (e.g. y+42). Formally, statements are just expressions that return unit.

Computations c include the seq and >>> combinators, conditionals, bindings for mutable variables, functions, computations, and calls.

In addition, Ziria provides a set of primitives:

- emit e. A stream *computer* that evaluates its argument and emits the value on the output stream. It is a computer, since it halts after emitting its value, returning a unit control value.
- take. A stream *computer* that takes one element from the input stream and returns it as a control value. For instance:

```
seq { (x : int) ← take
    ; emit (x+1) }
```

  will take one value x from the input stream and emit $x+1$.
- do m and return e. These primitives *lift* an m or e from the low-level imperative fragment to the computation language. They are both stream *computers* that execute their argument and return the final result as a control value. For instance, given a mutable variable y, the following code will take an input value, update y, and emit $y+1$:

```
seq { (x : int) ← take
    ; do { y := y+x+1; }
    ; emit (y+1) }
```

[1] Readers may recognize the combination of a monad [32] and arrow [24] structure in Ziria computation types, an idea that appears in several variations in the functional progrmaming literature [14, 38, 39].

$$
\begin{array}{lll}
c ::= & \mathsf{seq}\{x \leftarrow c;\ c\} & \text{Sequence} \\
\mid & c \gg\!\gg\!\gg\ c & \text{Data path composition} \\
\mid & \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 & \text{Conditionals} \\
\mid & \mathsf{var}\ \overline{x{:}\tau := v}\ \mathsf{in}\ c & \text{Mutable variables} \\
\mid & \mathsf{let}\ f(\overline{x{:}\tau}) = m\ \mathsf{in}\ c & \text{Function definition} \\
\mid & \mathsf{let\ comp}\ f(\overline{x{:}\tau}) = c_1\ \mathsf{in}\ c_2 & \text{Computation definition} \\
\mid & c(\overline{e}) & \text{Call computation function} \\
 & & \\
\mid & \mathsf{take} & \text{Take from input} \\
\mid & \mathsf{emit}\ e & \text{Emit on output} \\
\mid & \mathsf{do}\ m \mid \mathsf{return}\ e & \text{Lift expression} \\
\mid & \mathsf{repeat}\ c & \text{Repeat } c \text{ indefinitely} \\
 & & \\
\mid & \mathsf{while}\ e\ c & \text{Repeat } c \text{ while } e \\
\mid & \mathsf{times}\ e\ c & \text{Repeat } c\ e\text{-times} \\
\mid & \mathsf{map}\ f & \text{Map function over input} \\
 & & \\
\mid & \ldots & \\
e, m ::= & x \mid v \mid x := e \mid m_1; m_2 \mid f(\overline{e}) \mid \ldots & \text{Expressions} \\
v \quad ::= & \mathtt{unit} \mid i \mid \ldots & \text{Values}
\end{array}
$$

Figure 1: Simplified syntax of Ziria

The do and return primitives have identical semantics, but as a programmer convenience we use do for imperative code that returns `unit` (such as `y := x+y+1`) and return for expressions that will return a value, e.g. $\mathsf{return}(y+1)$.

- repeat c. A stream *transformer* that executes the stream *computer* c. When c halts, repeat re-initializes it and restarts, effectively implementing: $\mathsf{seq}\{c; c; \ldots\}$. Here is a component that filters out all 0-value elements of its input stream.

```
repeat { (x : int) ← take
       ; if x == 0 then return ()
         else emit x }
```

Ziria also provides a few more combinators for: repeating computers n times, mapping (expression) functions over input streams, and more. The typing rules for the most important primitives are summarized below:

$$
\begin{array}{rll}
\vdash \mathsf{take}: & \forall ab.\ \mathsf{Zr}\ (\mathsf{C}\ a)\ a\ b & \\
\vdash \mathsf{emit}\ e: & \forall a.\ \mathsf{Zr}\ (\mathsf{C\ unit})\ a\ \tau, & \text{if } \vdash e : \tau \\
\vdash \mathsf{return}\ e: & \forall ab.\ \mathsf{Zr}\ (\mathsf{C}\ \tau)\ a\ b, & \text{if } \vdash e : \tau \\
\vdash \mathsf{map}\ f: & \mathsf{Zr\ T}\ \tau\ \sigma, & \text{if } \vdash f : \tau \to \sigma \\
\vdash \mathsf{repeat}\ c: & \mathsf{Zr\ T}\ \tau\ \sigma, & \text{if } \vdash c : \mathsf{Zr\ (C\ unit)}\ \tau\ \sigma
\end{array}
$$

2.6 Executing Ziria pipelines

Ziria programs target commodity CPUs and large parts of program pipelines run on dedicated cores. The design of the Ziria *intrathread* execution model is motivated by performance, namely (a) processing with low latency, and (b) avoiding compiler-introduced buffering – even for programs that use $\gg\!\gg\!\gg$.

The fundamental insight is that there exist computations that can spontaneously produce output and push it downstream (the simplest being `emit e`), and computations that must first pull input in order to execute (the simplest being `take`). Many complex Ziria processing blocks will need to both push *and* pull data during their execution. Consequently, every Ziria computation compiles to a pair of code blocks, called `tick` and `proc`. The `tick` code block determines if the computation has a result readily available, and if so, immediately pushes it to the `proc` code block of the *downstream* component for processing. On the other hand, if input needs to be pulled, `tick` jumps to the `tick` code block of the *upstream* component. A `proc` code block consumes a pushed input and can push output to the downstream `proc` code block.

Two subtle points deserve attention:

- The `tick` or `proc` block can determine that the computation – if it is a stream computer – should halt. This will happen when `c1` halts in a `seq { x ←c1; c2}` computation. For this reason, `seq` compiles with a switchtable that selects which of the two `tick`/`proc` blocks is active at any point in time. Termination of `c1` activates the `tick`/`proc` blocks associated with `c2`.
- The `tick` and `proc` blocks for `c1` $\gg\!\gg\!\gg$ `c2` *are* `c2`'s `tick` and `c1`'s `proc`, respectively. This means pipelines are drained from the right, and thus there is no need for variable-sized queues between $\gg\!\gg\!\gg$. It also implies that values are pushed *as soon* as they become available, which is beneficial for latency.

On top of this basic compilation scheme, the Ziria compiler also makes static scheduling decisions, such as eliminating `tick` code blocks for components that can never spontaneously produce output but always require input (e.g. `map f` or `take`). This optimization reduces the administrative overhead of `tick`-ing through the data path.[2]

Pipeline parallelization. A `c1` $\gg\!\gg\!\gg$ ... $\gg\!\gg\!\gg$ `cn` in isolation can naturally be mapped onto multiple cores by introducing interthread queues and compiling each constitutent component according to the previous intrathread model. We generalize this observation to programs where the last computation in a `seq`-uence is a data-path composition:

```
seq { x ← c0
    ; c1 >>> ... >>> cn }
```

We allow programmers to replace any of the $\gg\!\gg\!\gg$ operators above with a variant $|\gg\!\gg\!\gg|$ to explicitly indicate partitioning onto multiple cores. Pipeline-parallelizing arbitrary uses of $\gg\!\gg\!\gg$ is more delicate and left as future work. The reason is, upon termination and reconfiguration of a data path, we need to synchronously notify all participating threads, making sure that no upstream thread in that path has erroneously consumed data not intended for this particular data path. The currently supported form of pipeline parallelization is sufficient for our workloads, however. Section 5 gives performance evaluation for WiFi TX/RX parallel pipelines.

3. Vectorization

A central optimization in the Ziria compiler is the *vectorization transformation*. It rewrites pipeline components that take values of type a and emit values of type b to take instead values of type $(\mathsf{array}[d_{\text{in}}]\ a)$ and emit values of type

[2] In Section 5 we evaluate execution model overhead with program size.

$(\mathsf{array}[d_{\text{out}}]\ b)$ for appropriate d_{in} and d_{out}, thus systematically transforming pipelines data paths to operate on vectors rather than scalars.

The benefits of compiler-based vectorization were outlined in the introduction. The alternative – manually adjusting the implementations of processing blocks for a specific placement in a pipeline – not only departs from intuitive protocol specifications and hampers reusability, but as the next paragraph shows, is also challenging to get right.

The challenge of vectorization. Let us examine a simple transformer `t` in isolation:

```
let comp t = repeat { (x:int) ← take
                    ; emit f(x); }
```

We could vectorize `t` to take arrays instead of singleton values of type `int`:[3]

```
let comp t_vect =
  repeat { (xa:arr[8] int) ← take
         ; for i in 0..7 { emit f(xa[i]); }}
```

Consider now placing `t` inside a pipeline that can be *reconfigured* with a `seq` block:

```
seq { x ← (t >>> c1); c2 }
```

Suppose that `c1` returns after consuming 4 values (one by one). The original `t` transformer can produce those 4 values by consuming 4 values from the input stream. On the other hand, `t_vect` can produce 4 values only after consuming a full array of 8 values. Hence, the moment the pipeline reconfigures to `c2`, `t_vect` has erroneously consumed 4 extra values, originally destined for `c2`. We conclude that the `t_vect` vectorization is incorrect for this pipeline.

The example illustrates the subtlety of vectorization and the challenge we address. The set of *feasible* vectorizations of a component depends on the placement of the component inside the pipeline, as well as on the data widths of adjacent blocks (as we have to – at the very least – produce well-typed pipelines to avoid buffer overruns and segmentation faults).

Overview of our algorithm. Guided by this intuition, we design a decentralized vectorization algorithm that:

1. In a top-down fashion, identifies sets of feasible vectorizations for transformers and computers based on the number of values they take and emit (Section 3.1) and their pipeline placement (Section 3.2), then
2. In a bottom-up fashion, composes feasible vectorization candidates together to re-assemble vectorization candidates for the original pipeline (Section 3.3).

To avoid search space explosion, the second step uses ideas from distributed optimization to locally prune the set of candidates and finally return a single candidate that maximizes a (parameter) utility function.

We implement vectorization as an AST transformation (versus a code-generation-time optimization) to allow for further waves of optimizations on the vectorized ASTs, as we illustrate in Section 4.

[3] Though valid, this may not be the most efficient vectorization since the output is not vectorized.

3.1 Cardinality analysis

As the examples above demonstrate, central to vectorization is a static analysis that we refer to as "cardinality analysis." Cardinality analysis infers for each computer `c` the number α_{in} of values the computer will take from its input and the number α_{out} of values the component will emit on its output before returning. All transformers in our WiFi workload use `repeat`-ed computers of statically deducible cardinalities, but a few computers take or emit a dynamic number of values. For these few cases Ziria provides several forms of annotations to force vectorizations, e.g., when we know that some multiple of a fixed number of values will be emitted, such as in the CRC implementation in our repository.

3.2 Feasible vectorization sets

In a top-down fashion, our algorithm determines feasible vectorizations, distinguishing the following three cases.

Computer vectorizations. For correctness, vectorization of a computer `c` needs to ensure that the vectorized computer takes (one or more) arrays of size d_{in} and emits (one or more) arrays of size d_{out} for some *divisors* d_{in} and d_{out} of cardinalities α_{in} and α_{out} respectively. We call this a *down-vectorization* of the computer because the final array sizes are less or equal to the cardinality parameters.

Transformer-before-computer vectorization. Consider now the example from the beginning of this Section,

```
seq { x ← (t >>> c1); c2 }
```

and assume `t` is of the form `repeat c` for some computer `c` with input cardinality α_{in} and output cardinality α_{out}.

Suppose `t` vectorizes so that in each iteration it takes one array of size $2\alpha_{\text{in}}$ and emits two arrays of size α_{out} each. The downstream computer `c1` may also be vectorized to take an array of α_{out} input values, so the types match. This vectorization candidate is nevertheless incorrect because `c1` could stop after reading the first α_{in} inputs from `t`'s output, leaving `t` with extra α_{in} values which should have been processed by `c2`.

The solution is to never increase the output rate per input in a "transformer-before-computer" vectorization. A transformer in a data path with a computer to the right can only safely *up-vectorize* to take arrays of size $d \cdot \alpha_{\text{in}}$ and emit arrays of size $d \cdot k \cdot \alpha_{\text{out}}$ for some d and k. We refer to this mode as up-vectorization because it takes and emits arrays with larger width than the cardinality parameters.

Finally, we can also *down-vectorize* to take arrays d_{in} and emit arrays d_{out} where d_{in} and d_{out} are divisors of α_{in} and α_{out} taken and emitted per iteration.

Transformer-after-computer vectorization. The symmetric situation is also interesting. Consider a transformer `t` that consumes the output of computer `c1`:

```
seq { x ← (c1 >>> t); c2 }
```

Although transformers do not themselves return control values, in this case `t` will process a *finite* amount of data because the component upstream is a computer.

Again, consider an example of a transformer `t` in the form `repeat c` for some computer `c` that has input cardinality α_{in} and output cardinality α_{out}. Suppose `t` vectorizes so that in each iteration it takes one array of α_{in} values and in every second iteration emits an array of $2\alpha_{\text{out}}$ values, leading to an input–output rate of 2-to-1. If the computer `c1` emits an array of size α_{in} (so the types match) and then immediately returns, the expected α_{out} output values will never be emitted since the output granularity is $2\alpha_{\text{out}}$.

Unsurprisingly, the solution is to never decrease the output rate per input in a "transformer-after-computer" vectorization. A transformer in a data path with a computer to the left can only safely *up-vectorize* to take arrays of size $d \cdot k \cdot \alpha_{\text{in}}$ and emit arrays of size $d \cdot \alpha_{\text{out}}$ for some d and k. Finally, as before, down-vectorizations are also acceptable.

3.3 Assembling vectorization candidates

Having collected constraints describing all feasible vectorizations for simple components, our algorithm proceeds to compose these sets to form well-typed vectorized pipelines. Let $\mathcal{D}(\texttt{c})$ denote the feasible set for computation `c` as a set of triples $(\texttt{cv}, d_{\text{in}}, d_{\text{out}})$, where `cv` takes and emits arrays $(\mathsf{array}[d_{\text{in}}]\ a)$ and $(\mathsf{array}[d_{\text{out}}]\ b)$ respectively. The following rules describe how feasible sets compose over $>\!\!>\!\!>$ and `seq`:

$$\mathcal{D}(\texttt{c1} >\!\!>\!\!> \texttt{c2}) =$$
$$\{\ (\texttt{cv1} >\!\!>\!\!> \texttt{cv2}, d_{\text{in}}, d_{\text{out}}) \text{ where}$$
$$\exists d.\ (\texttt{cv1}, d_{\text{in}}, d) \in \mathcal{D}(\texttt{c1}) \text{ and } (\texttt{cv2}, d, d_{\text{out}}) \in \mathcal{D}(\texttt{c2})\ \}$$
$$\mathcal{D}(\texttt{seq x}\leftarrow\texttt{c1;c2}) =$$
$$\{\ (\texttt{seq x}\leftarrow\texttt{cv1;cv2}, d_{\text{in}}, d_{\text{out}}) \text{ where}$$
$$(\texttt{cv1}, d_{\text{in}}, d_{\text{out}}) \in \mathcal{D}(\texttt{c1}) \text{ and } (\texttt{cv2}, d_{\text{in}}, d_{\text{out}}) \in \mathcal{D}(\texttt{c2})\ \}$$

The sets of vectorizations that the aforementioned strategy introduces can grow large very quickly (tens or hundreds of thousands, for WiFi-scale pipelines), even if we impose limits on the maximum size of arrays.

To avoid search space explosion, we perform local pruning of candidates. Consider the example of `c1` $>\!\!>\!\!>$ `c2`. There will be multiple feasible input (d_{in}) and output (d_{out}) data widths that this component can vectorize into. Moreover, for each feasible pair of d_{in} and d_{out}, there may be several possible internal vectorizations arising from the set of intermediate widths d in the first rule above.

Locally pruning the candidates for `c1` $>\!\!>\!\!>$ `c2` for a given d_{in} and d_{out} amounts to deciding, for a choice of two intermediate widths d and d' (and consequently two vectorization candidates), which of the two we should prefer. A natural choice, guided by the need for "fat" pipelines, is to choose the one with the highest intermediate width.

However, at top-level we must choose a *single vectorization* among a set of vectorizations, irrespectively of input and output widths. A simple generalization of the aforementioned local choice is to choose the vectorization with the highest sum of all input, intermediate, and output widths in the pipeline. However, that choice is not always desirable. Consider two pipelines vectorized to input, intermediate, and output widths of 256-4-256 and 128-64-128, respectively,[4] where the unvectorized constituent components emit one value for each value they take. Choosing the pipeline with the highest sum of all widths means the first candidate ($256 + 4 + 256 > 128 + 64 + 128$) is selected, but the intermediate narrow width, 4, in the first pipeline is a clear bottleneck.

$$\mathcal{D}(\texttt{c1} >\!\!>\!\!> \texttt{c2}) =$$
$$\{\ (\texttt{cv1} >\!\!>\!\!> \texttt{cv2}, d_{\text{in}}, d_{\text{out}}, u) \text{ where}$$
$$(\texttt{cv1}, d_{\text{in}}, d, u_1) \in \mathcal{D}(\texttt{c1}) \text{ and } (\texttt{cv2}, d, d_{\text{out}}, u_2) \in \mathcal{D}(\texttt{c2})$$
$$\text{and } u = u_1 + f(d) + u_2 \text{ is maximal }\ \}$$

$$\mathcal{D}(\texttt{seq x}\leftarrow\texttt{c1;c2}) =$$
$$\{\ (\texttt{seq x}\leftarrow\texttt{cv1;cv2}, d_{\text{in}}, d_{\text{out}}, u_1 + u_2) \text{ where}$$
$$(\texttt{cv1}, d_{\text{in}}, d_{\text{out}}, u_1) \in \mathcal{D}(\texttt{c1}) \text{ and}$$
$$(\texttt{cv2}, d_{\text{in}}, d_{\text{out}}, u_2) \in \mathcal{D}(\texttt{c2})\ \}$$

Figure 2: Vectorization composition and utility calculation

An alternative is to pick the candidate with the highest minimal width – this strategy would happily select the second candidate above. But highest-minimal-width is not always good either, since it will prefer 8-8-8-8 over 256-256-256-4 even though the second candidate is better in this case: candidate 256-256-256-4 sends 256 elements directly through the first two blocks and only pays the cost of breaking the result into arrays of size 4 at the end, causing 1+1+64 = 66 `tick`/`proc` executions. By contrast the 8-8-8-8 candidate induces $4 \cdot 32 = 128$ `procs`.

From the above discussion, it is evident that the two extreme metrics, maximizing the sum of all block widths and maximizing the smallest of all widths, are conflicting. A similar issue is often encountered in computer network design. The seminal work of Kelly et al. [26] presents a utility framework parameterized over a set of concave (utility) functions that is amenable to decentralized optimization. The trade-off between the two extremes can be balanced through the choice of the utility function. Let f be such a utility function. Choosing $f(d) = d$ results in maximizing the sum of widths, whereas choosing $f(d) = 1/d^a$ for large a is known to be equivalent to maximizing the minimum width.

Here we adapt this framework. Specifically, we choose $f(d) = \log(d)$, a function that is known to strike a good balance between the two extremes [26]. We extend the earlier composition rules with local pruning and decentralized utility calculation, as given in Figure 2. We evaluate the effects of the resulting vectorizations in Section 5.

[4] In light of the previous discussion on rates, this may occur if the pipeline is composed just of transformers.

4. Other Optimizations

The Ziria compiler implements a series of type- and semantics-preserving optimizations that eliminate computations over expressions, reduce memory copying, and fuse multiple computations for more efficient execution. These optimizations are mainly based on standard techniques but are instrumental in producing performant code. We review the most important in this section and illustrate their synergy.

Static scheduling optimizations. One set of optimizations focuses on converting sequences of computations that include `seq` and >>> into imperative code wherever possible. One particular form of this optimization is called *auto-mapping* and is essential for producing performant code, as the overhead measurements in Section 5.2 (Figure 4) show. Figure 3 illustrates auto-mapping: the vectorized scrambler kernel is pulled out in a separate local function, and the `repeat` block is replaced with a call to `map`. A whole set of optimizations, such as inlining, pushing around `let`-bound definitions and conditionals, and replacing computation-level loops with expression-level loops, are specifically designed to enable auto-mapping. Auto-mapping is a form of static scheduling, since the code generator is aware that `map` cannot proceed without consuming input (cf. Section 2.6) – for instance a long sequence of >>> compositions of `map` will only push values downstream without using `tick` blocks.

Lookup table generation. Many PHY operations are defined to operate at a bit-level granularity, but direct implementation of such code is inefficient on CPUs with wide data buses. A common optimization is to identify such operations and speed them up by memoizing the results as lookup tables (LUTs) (manually created LUTs are used pervasively in Sora and GNU Radio for performance, for example). However, writing functions that use LUTs is tedious and results in code that is hard to read and modify.

The Ziria compiler solves this problem by automatically detecting portions of a program that are amenable to a LUT implementation (e.g., expressions complex enough to be worth converting to a LUT but whose LUT sizes are not too large) and automatically converting these expressions to LUTs. For instance, the auto-mapped version of the scrambler in Figure 3 is compiled to use a LUT with an index consisting of the current input (8 bits) and the scrambler state (7 bits), a total of 2^{15} entries. Packing input values to LUT indices and unpacking results required delicate performance tuning in the Ziria compiler; Section 5 evaluates the results of this optimization in our WiFi pipeline.

5. Evaluation

In this section we seek to answer the following questions about the performance of Ziria: (1) What is the overhead of Ziria's execution engine? (2) What is the speedup of various compiler optimizations? (3) Can we implement WiFi PHY in Ziria to meet the protocol specifications and how does this implementation compare with a state-of-the-art, manually optimized implementation on the same platform?

5.1 Methodology

We evaluate the performance of our framework on various DSP algorithms that are part of a standard WiFi transceiver. We choose Sora's WiFi implementation [45] as our reference implementation (since it is the most stable available) and port it to Ziria. Our port consists of $\approx$ 3k lines of Ziria code in total and it is verified against Sora's implementation. As part of Ziria, we provide a basic signal processing library with a high-level interface for SIMD instructions and efficient implementations (borrowed from Sora) of FFT, IFFT and Viterbi decoding.[5] The rest is written entirely in Ziria and available online [22].

We evaluate data throughput of a number of the processing blocks and compare the performance of our Ziria implementation with the manually optimized Sora implementation [45] and the WiFi requirements. We compile both on the Sora-supported C compiler [31], and we adapt our runtime to use Sora's runtime libraries. In particular, we use Sora's user-mode threading library, which allows us to run threads at the highest priority and pinned to a specific core, effectively preventing the OS from preempting their execution. Our evaluation machine is a 2-year-old Dell T3600 PC with an Intel Xeon E5-1620 CPU[6] at 3.6 GHz running Windows 8.1 and the WinDDK compiler [31], together with the SoraMIMO SDR platform. When we measure the maximum rates of our software components (which are faster than the line speed of WiFi), we feed the input samples directly from memory and discard the output. We also evaluate the performance of the full SDR system using live wireless transmissions.

5.2 Overheads of nonoptimized execution model

Control-flow composition (`seq`). We measure the overhead of `seq` by measuring the runtime of a program containing n computation components in sequence, with each component consisting of a single call to `sin()`. As a performance baseline, we use the runtime of the equivalent program in which all n `sin` operations are executed in the same block. The results are depicted in Figure 4, top. The dashed line gives runtime of the baseline program for n `sin` computations. The solid line gives runtimes for n `sin` components bound in sequence. Both are average over 200 million inputs and we report average execution time per data item (confidence intervals are very small). We verify that the runtime data fit a linear model as a function of n, indicating that the cost of `seq` grows linearly with the number of components. The cost of a single `seq` operation on our system, given by the difference of the slopes of the two lines, is around 3ns.

[5] These blocks are standard and are reused across all modern physical layers (WiFi, WiMax, LTE). Efficient implementations are already available across a large range of SDR platforms.

[6] We observe very similar results on a laptop with an Intel i7.

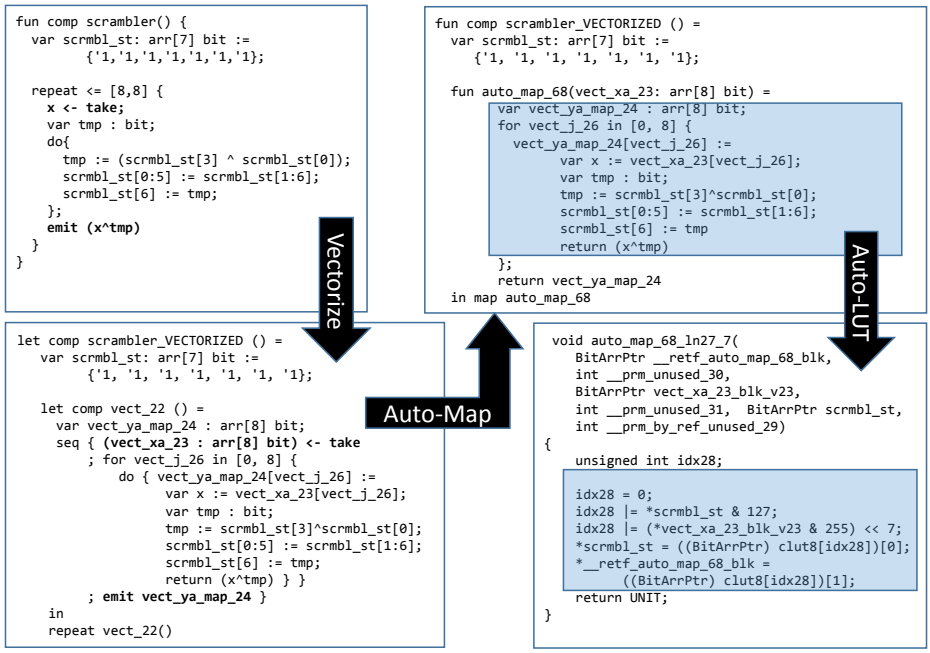


Figure 3: Optimization synergy: original scrambler, auto-vectorized, auto-mapped, and generated C code with a LUT.

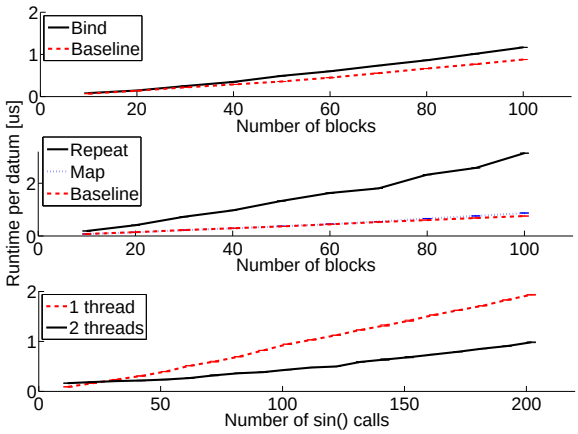
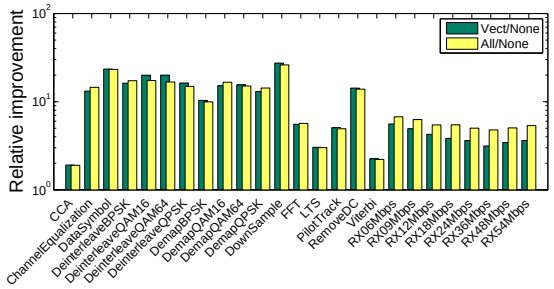


Figure 4: Overheads of various Ziria components: `seq` (top), >>> (middle), |>>>| (bottom)

Dataflow composition >>>**.** To evaluate the overhead of >>>, we measured the runtime of a program containing n >>>-composed code blocks, with each block executing the computation `repeat {x←take; emit(sin(x))}`. We also measured the runtime of a version of this program optimized to use `map`. The baseline is the runtime of a program in which all n `sin` calls are merged into a single block. We average over 200 million inputs and report average execution time per data item (confidence intervals are very small). The results are depicted in Figure 4, middle. The cost of a single >>> with repeat on our system, given by the difference of the slopes of Repeat and Baseline, is ≈ 24ns, while the cost of a single >>> with map is ≈ 1ns. The difference stems from the fact that the `repeat` block executes several `ticks` and `procs` in each round, whereas `map` execution is completely streamlined. As shown in Section 5.3, the overhead of >>> is usually significantly reduced by auto-mapping (Section 4), which converts certain `repeat` blocks into `maps`. The cost is further amortized by vectorization (Section 3), except in a few cases in which automapping and vectorization cannot be applied because not all sizes are statically known. Examples where this occurs include CRC, for which the input size depends on packet length and is therefore not known at compile time, and Viterbi, whose output frequency and size may depend on the amount of noise present at the received signal.

Pipelined dataflow composition |>>>|**.** To gauge the overhead of pipelining Ziria programs onto multiple cores using |>>>|, we measured the runtime of n `sin` calls both on a single core and divided evenly onto two cores. Figure 4, bottom, shows the results of this experiment. We average over 200 million inputs and report average execution time per data item (confidence intervals are very small). The red dashed line and solid black line give the execution time when running on a single core and on two cores, respectively. The point at which these two lines intersect, approximately 30 computations per datum, is the point at which we break even, i.e., when pipelining gives speedup rather than slowdown. The speedup is approximately 1.7x at 60 calls and 2x at 90 calls.

(a) Vect. **WiFi RX** blocks (green) vs. all (yellow), relative to no opts.



(b) Vect. **WiFi TX** blocks (green) vs. all (yellow), relative to no opts.

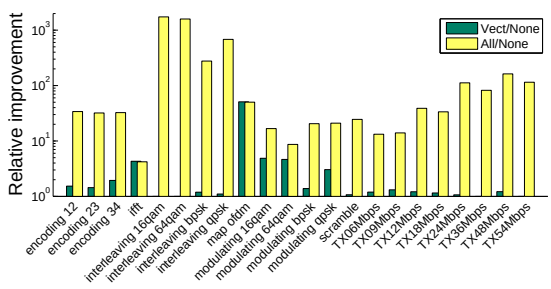
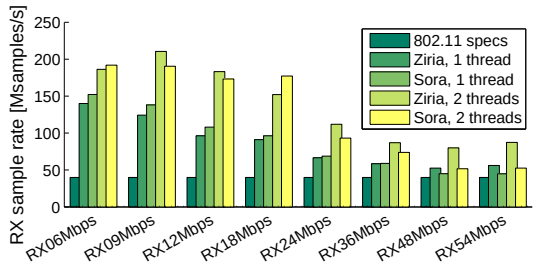


Figure 5: Benefits of optimizations

(a) WiFi receiver throughputs at various data rates



(b) WiFi transmitter throughputs at various data rates

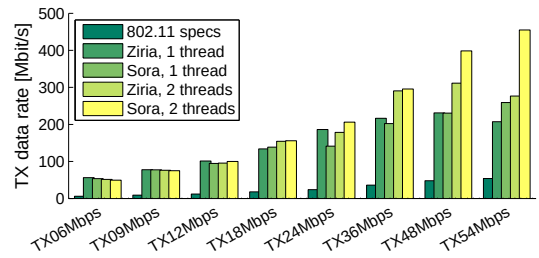


Figure 6: WiFi throughput

5.3 Ziria optimizations

Vectorization is instrumental in achieving good performance, both on its own and in combination with LUTs. In plot 5a and 5b, we give the relative improvement of vectorization (green) and vectorization+LUT (yellow) over no optimization for receiver (RX) and transmitter (TX) blocks. Vectorization brings large benefits to the receiver, with order-of-magnitude speedups. Vectorization alone is not as effective on the transmitter side, since most TX blocks perform bit-level operations that dominate performance. However, vectorization enables LUT generation, and the combined speedup of both optimizations is up to $1000\times$ (in the full TX pipeline at 54 Mbps the compiler automatically identifies 40 LUT opportunities). Note that wherever we measure the performance of vectorization, we also include other control flow optimizations mentioned in Section 4.

5.4 WiFi performance

Finally, we compare our WiFi implementation in Ziria with the IEEE WiFi standard requirements and the Sora implementation in terms of compile time, throughput, and latency. We also evaluate its real world performance.

Compile time. We begin by measuring the compile times in Sora (C++ with templates to executable code) and Ziria (Ziria high-level code to executable code) using the same WinDDK C compiler [31]. Both Sora and Ziria take 8s to compile the transmitter at 54 Mbps to an executable file. Ziria is faster when compiling the receiver at 54 Mbps: it takes 15s compared to the 26s taken by Sora.

Throughput. We next inspect the receiver's performance, plot 6a. The specification mandates a sampling (input) rate of 40 million samples/second (dark green bars). We measure the maximum throughputs of the Sora implementations of receivers at various data rates and compare with the equivalent implementations in Ziria, both with 1 and 2 threads. We see that all Ziria implementations meet the WiFi specifications at all data rates. We also see that we are at most 15% slower than the corresponding manually optimized Sora implementations and even faster than Sora in several of the most demanding cases.

In particular, our 2-threaded implementation of the receiver at 54 Mbps is more than 60% faster than Sora's. This is because our vectorizer finds the optimal width of the Viterbi decoder for this particular pipeline (288 samples), whereas Sora uses the default width (48 samples). Since Viterbi is placed on a different core from the rest of the pipeline, we reduce the number of synchronization messages 6-fold.

In plot 6b we give the results for the same experiment applied instead to the WiFi transmitter. The inputs to the TX pipeline are data bits, so the required input rates correspond to the WiFi data rates (dark green bars). Ziria again meets the WiFi requirements and in most cases is faster than Sora.

The main exceptions are the transmitters at 48 and 54 Mbps. These data rates use 64QAM modulation and require packing chunks of 6 bits of data per symbol; our current bit operations are not very efficient for nonaligned data.

Latency. We also evaluate the latency of our single thread WiFi implementation. We measure two types of latencies: the read latency at input and the write latency at output. To measure read latency, we randomly sample 10000 latencies between two consecutive read operations, and we do the same for write latency. Since read and write operations are vectorized, we plot an average latency per sample for each measured vectorized read/write. We normalize all latencies with respect to the maximum average latency per datum for each data rate ($1/$data rate for TX reads and $1/$sampling rate $= 1/40$MHz for TX writes and RX reads).

We plot the CDFs of normalized latencies in Figure 7. Plot 7a shows that latencies at transmitter reads are highly nonuniform. This is because the time it takes to process one input datum may vary widely depending on the order in which we execute various blocks. For example, it will be small while we are reading data to fill an FFT buffer. But the FFT will get executed once the buffer is filled, which will introduce high latency before the subsequent read. However, since the FFT is the transmitter block with the largest vectorization and also the last component in the TX pipeline, the latencies at write are much more uniform (Plot 7b). The read latencies for the receiver are shown in Plot 7c.

For the performance of WiFi, it is important that the write latency of the transmitter is below the maximum latency dictated by the line rate of WiFi ($1/40$MHz). This is because a sudden excessive write delay can cause an interruption in transmission. Similarly, if the read latency of the receiver is above the maximum latency dictated by the line rate, a buffer overflow can occur causing dropped RX samples. WiFi is not designed to cope with dropped samples or interruptions, and such events will yield immediate packet losses.

However, occasional large delays are not catastrophic; all SDR hardware uses buffers to amortize such delays. The sizes of these buffers are dimensioned according to the protocol specs. For example, WiFi requires hardware to finish processing packet reception and start transmitting in 10μs (SIFS, "short interframe space" time), so any buffering and processing delays smaller than SIFS are acceptable.

We see that only 0.2% of all observed write latencies at the TX and read latencies at the RX are above the maximum allowed by the device line speed, and the highest of all observed latencies is $5\times$ the maximum, which is still more than $100\times$ smaller than the SIFS time.

Testbed evaluation. In order to further verify that our implementation meets the system requirements in practice and that no excessive delay between consecutive samples, caused by our nonrealtime OS scheduler, will cause packet transmission or reception errors, we set up a realtime wireless transmission using Ziria WiFi code and Sora SDR hardware. High data rates (24-54 Mbps) involve amplitude modulation (16QAM and 64QAM) and require a more accurate implementation of an automatic gain control algorithm, which is currently provided neither in Sora nor in Ziria. We therefore experiment with the four lowest data rates (6-18 Mbps). For each of these rates we send a sequence of packets encoded at the selected rate at a constant pace. Each packet has a unique ID so we can detect losses accurately. We count how many packets are correctly received and calculate the packet error rate. In our experiment, we lose around 2% of 10,000 packets transmitted [18]. This is on par with loss observed with commercial WiFi cards (cf. [41]). As discussed before, WiFi cannot cope with sample losses, and the low loss rate we observe here implies that we meet WiFi timing requirements.

6. Related work

Software-defined radio. SDR platforms have traditionally been FPGA- [29, 33, 35] and DSP-based [43, 49]. More recently, CPU-based platforms [6, 23, 34, 45, 50] have gained popularity, in particular due to widespread use of USRP radios and GNU Radio [12, 50]. Similarly, there are numerous programming platforms and approaches for SDRs [7, 12, 13, 19, 30, 35, 36, 42, 48, 52].

As we motivated in the introduction, however, the existing SDR platforms typically fail to provide convenient abstractions for programming pipeline reconfiguration, an important aspect of PHY design. For example, the Sora [13] transmitter consists of a single data pipeline with global shared state for (re)configuration. GNU Radio [37] encodes control flow in the data stream (e.g., using a stream of 0's and 1's to convey the presence or absence of a preamble). StreamIt [48] uses asynchronous teleport messages for pipeline reconfiguration, which can obscure control flow. Spiral [52] does not offer an easy way to mix control and data flow.

Further, many of the software platforms require tedious manual optimization, which can limit code reuse and lead to subtle errors. Examples include Sora and GNU Radio components that are programmed with predefined, manually adjusted widths and therefore may not be compatible with arbitrary pipelines. In Sora, the programmer must also manually identify and implement lookup tables, which can lead to verbose code (for example, the scrambler implementation is 90 lines of C++ code in Sora but only 20 lines in Ziria).

Dataflow languages. In addition to work on SDR, Ziria builds on a significant body of programming languages research. Synchronous dataflow (SDF) languages [9, 15, 16] have been used in embedded and reactive systems for modeling and verification but – to our knowledge – never to implement line-rate software PHY designs. Parameterized SDF [10], like Ziria, supports dynamic reconfiguration of the dataflow graph but does not develop language abstractions for reconfiguration like our `seq` combinator. StreamIt [48], also based on synchronous dataflow, was one of the early works to target DSP applications, including software WiFi and 3GPP

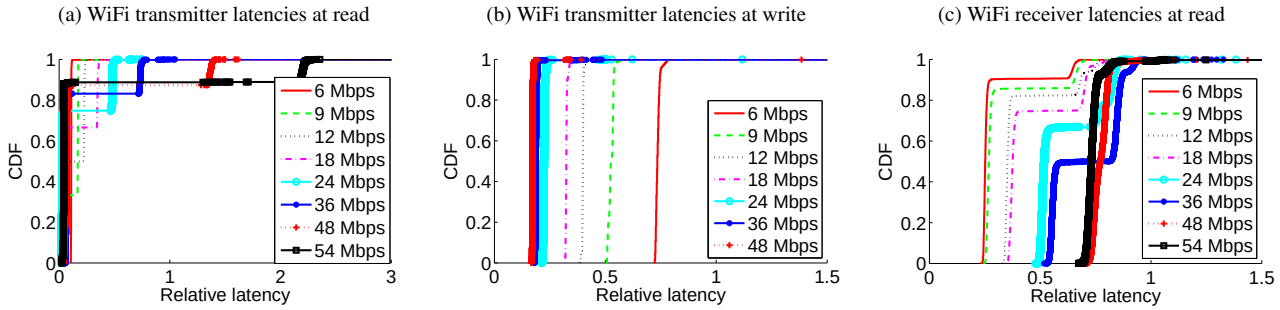


Figure 7: WiFi latency at various data rates for single-threaded WiFi transmitter and receiver

PHY, though it is not clear whether these implementations can operate at line rate. More recently, Liquid Metal [4] showed how to compile a high-level Java-like language with streaming primitives to a combination of Java bytecode, C code, and FPGA bitfiles, with runtime support for dynamically selecting from among the implementations. Both StreamIt and Liquid Metal support forms of cardinality-aware vectorization, although simpler in the absence of pipeline reconfigurations.

Functional programming and FRP. The design of Ziria and its key combinators draws from monads [32] and arrows [24]. A flavor of our `seq` combinator (called "switch") is used in Yampa [17], a popular functional reactive programming (FRP) framework that lacks an efficient execution model. The Haskell Pipes [39] library includes monad transformers that provide high-level functionality similar to Ziria's stream combinators, but it does not guarantee implementation in constant space. Ziria's tick and process semantics resemble the push–pull model of streaming popular in FRP [20, 21, 47]. Feldspar [5], a language for DSP embedded in Haskell, uses a novel vector representation that supports vector fusion, which could be profitably incorporated into the Ziria compiler. Spiral [40] generates efficient implementations of a wide variety of linear transforms (e.g., DFT) from high-level algebraic specifications and compares favorably to Sora in benchmarks of a Spiral-generated WiFi PHY [53]. In principle, Spiral could be used to generate architecture-tuned versions of some of the basic stream-processing blocks we currently use in Ziria. Bitvector program synthesis using, e.g., sketching [27] may be another way to generate optimized low-level processing blocks.

7. Conclusion

We presented Ziria, the first high-level SDR platform with a performant execution model. To validate our design, we built a compiler that performs optimizations done manually in existing CPU-based SDR platforms: vectorization, lookup table generation, and annotation-guided pipelining. To demonstrate Ziria's viability, we used Ziria to build a rate-compliant PHY layer for WiFi 802.11a/g.

References

[1] 3GPP 36.211. Evolved universal terrestrial radio access (e-utra) - physical channels and modulation. URL `http://www.3gpp.org/DynaReport/36211.htm`.

[2] 3GPP 36.212. Evolved universal terrestrial radio access (e-utra) - multiplexing and channel coding. URL `http://www.3gpp.org/DynaReport/36212.htm`.

[3] 3GPP 36.213. Evolved universal terrestrial radio access (e-utra) - physical layer procedures. URL `http://www.3gpp.org/DynaReport/36213.htm`.

[4] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, 2010. ISBN 978-1-4503-0203-6.

[5] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Anders Persson, Mary Sheeran, Josef Svenningsson, András Vajda, et al. Feldspar: A domain specific language for digital signal processing algorithms. In *FMMC*, 2010.

[6] H. V. Balan, M. Segura, S. Deora, A. Michaloliakos, R. Rogalin, K. Psounis, and G. Caire. USC SDR, an easy-to-program, high data rate, real time software radio platform. In *Software Radio Implementation Forum*, 2013.

[7] Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. OpenRadio: a programmable wireless dataplane. In *HotSDN*, 2012.

[8] T. Bansal, B. Chen, P. Sinha, and K. Srinivasan. Symphony: Cooperative packet recovery over the wired backbone in enterprise WLANs. In *MOBICOM*, 2013.

[9] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *SCP*, 19(2), 1992.

[10] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Signal Processing, IEEE Transactions on*, 49(10):2408–2421, 2001.

[11] B. Bloessl, M. Segata, C. Sommer, and F. Dressler. An IEEE 802.11a/g/p OFDM receiver for GNU radio. In *SRIF*, 2013.

[12] Eric Blossom. GNURadio: tools for exploring the radio frequency spectrum. *Linux Journal*, 2004(122):4, 2004.

[13] Bricks. Microsoft Research, Brick specification, 2011. URL http://research.microsoft.com/apps/pubs/default.aspx?id=160800.

[14] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*. Doktorsavhandlingar vid Chalmers tekniska högskola. Ny serie, no: 1366. Institutionen för datavetenskap, Chalmers tekniska högskola, 1998. ISBN 91-7197-611-6.

[15] Paul Caspi. Lucid Synchrone. In *Actes du colloque INRIA OPOPAC, Lacanau*. HERMES, November 1993.

[16] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Univ. Pierre et Marie Curie, Lab. LIP6, 2000.

[17] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[18] Demo. Ziria WiFi demo. URL http://research.microsoft.com/apps/video/default.aspx?id=228361.

[19] A. Dutta, D. Saha, D. Grunwald, and D. Sicker. CODIPHY: Composing on-demand intelligent physical layers. In *SRIF*, 2013.

[20] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[21] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[22] GitHub. Ziria github repository. URL https://github.com/dimitriv/Ziria.

[23] GSM. Fairwaves GSM base-station. URL https://fairwaves.co/wp/.

[24] John Hughes. Generalising monads to arrows. *SCP*, 37(1-3), 2000.

[25] IEEE. Part 11: Wireless LAN MAC and PHY specifications high-speed physical layer in the 5 GHz band, 1999. URL http://standards.ieee.org/getieee802/download/802.11a-1999.pdf.

[26] F. P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[27] A Solar Lezama. *Program synthesis by sketching*. PhD thesis, Citeseer, 2008.

[28] T. Li, M. K. Han, A. Bhartia, L. Qiu, E. Rozner, Y. Zhang, and B. Zarikoff. CRMA: Collision-resistant multiple access. In *MOBICOM*, 2011.

[29] Lyrtech. Lyrtech. URL http://intrinsic.lyrtech.com/.

[30] Mathworks. Simulink. URL http://www.mathworks.co.uk/products/simulink/.

[31] Microsoft. Windows driver kit version 7. URL http://www.microsoft.com/en-us/download/details.aspx?id=11800.

[32] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991.

[33] Patrick Murphy, Ashu Sabharwal, and Behnaam Aazhang. Design of WARP: a wireless open-access research platform. In *ESPC*, 2006.

[34] Myriad. MyriadRF open source wireless hardware. URL http://myriadrf.org/.

[35] Man Cheuk Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. Airblue: a system for cross-layer wireless protocol development. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, page 4:1–4:11, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0379-8.

[36] NI. National instruments, LabVIEW. URL http://www.ni.com/labview/.

[37] OFDM. GNU Radio OFDM implementation. URL http://gnuradio.org/redmine/projects/gnuradio/repository/changes/gr-digital/python/digital/ofdm_receiver.py?rev=master.

[38] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 91–105, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65527-1. URL http://dl.acm.org/citation.cfm?id=645769.667757.

[39] Pipes. Haskell Pipes Tutorial. URL http://hackage.haskell.org/package/pipes-4.0.0/docs/Pipes-Tutorial.html.

[40] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[41] S. Rayanchu, A. Mishra, D. Agrawal, S. Saha, and Suman Banerjee. Diagnosing wireless packet losses in 802.11: Separating collision from weak signal. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, April 2008.

[42] T. Rondeau, N. McCarthy, and T. O'Shea. SIMD programming in GNURadio: Maintainable and user-friendly algorithm optimization with VOLK. In *SDR WinnComm*, 2013.

[43] Saankhya. Saankhya labs, multi-standard modems. URL http://www.saankhyalabs.com.

[44] S. Sen, R. R. Choudhury, and S. Nelakuditi. No time to countdown: Migrating backoff to the frequency domain. In *MOBICOM*, 2011.

[45] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. Voelker. Sora: High performance software radio using general purpose multi-core processors, 2009.

[46] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX symposium on {N}etworked {S}ystems {D}esign and {I}mplementation*, NSDI'09, pages 75—90, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558983.

[47] Colin J Taylor. *Formalising and reasoning about Fudgets*. PhD thesis, University of Nottingham, 1998.

[48] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *CC*, 2002.

[49] TMS320. Texas Instruments TMS320TCI6616 Communications Infrastructure KeyStone SoC. URL `http://www.ti.com/lit/ds/sprs624d/sprs624d.pdf`.

[50] USRP. Ettus Research, USRP: Universal Software Radio Peripheral. URL `http://www.ettus.com/`.

[51] Volo. Volo Wireless, low-cost last mile broadband. URL `http://www.volowireless.com/`.

[52] Y. Voronenko, V. Arbatov, C. Berger, R. Peng, M. Puschel, and F. Franchetti. Computer generation of platform-adapted physical layer software. In *Proc. Software Defined Radio (SDR)*, 2010.

[53] Y Voronenko, V Arbatov, CR Berger, R Peng, M Pueschel, and F Franchetti. Computer generation of platform-adapted physical layer software. *Proceedings Software Defined Radio (SDR)*, 2010.