Abstract
Software-defined radio (SDR) is a challenging domain for language designers. To be useful in the real world, radio protocol implementations must operate at high data rates with low latency, yet to be useful to implementers, a language should allow programmers to express algorithms at a high level of abstraction without having to worry about the very low-level details that are necessary for meeting performance requirements. Ziria [27] demonstrated that a high-level language for writing wireless physical layer (PHY) protocols could be competitive with hand-written C++, but only in a context where performance-critical computations, such as FFT and Viterbi, were still written in C++ and accessed via a foreign function interface.

We demonstrate that a new implementation of Ziria, embodied in the kzc compiler, allows even performance-critical blocks such as FFT and Viterbi to be written in a high-level language without sacrificing performance. Because kzc performs whole-program optimization, a radio protocol pipeline using an implementation of Viterbi written in Ziria can outperform an implementation that calls out to C++. The contributions of this paper fall into two categories. First, we describe two new optimizations in kzc, both of which are critical for wringing performance out of high-level code: an aggressive partial evaluator for Ziria programs, and an automatic lookup-table (LUT) generator. Second, we show how these optimizations allow the efficient implementation of three performance-critical blocks in Ziria: Viterbi decoding, the Fast Fourier Transform (FFT), and the inverse Fast Fourier Transform (IFFT).

CCS Concepts → Software and its engineering → Compilers; Domain specific languages;

Keywords software-defined radio, domain-specific languages, compilers, partial evaluation

1 Introduction
Real-world radio protocol implementations must operate at high data rates and with low latency. The promise of software-defined radio (SDR) is to provide a platform for radio protocol implementation and experimentation that can meet these performance constraints while providing a software-style compile-debug cycle. Hardware platforms for SDR, such as BladeRF [23] and the USRP series [10], meet SDR performance requirements by utilizing field-programmable gate arrays (FPGAs). FPGAs can be easily reprogrammed, but constructing the bitfiles used to configure an FPGA requires knowledge not only of a hardware description language, such as VHDL or Verilog, but a fair amount of experience with a vendor-specific tool chain. For these reasons, the tools often used to program SDR platforms, like GNU Radio [4] and Matlab, use a given SDR platform as a black box and make no attempt to program the underlying FPGA-hosted firmware. However, pushing all computation onto the CPU make it difficult to meet real-time protocol requirements. Sora [29] demonstrated that it is possible for a CPU-only physical layer (PHY) protocol implementation to run at acceptable speeds; however, the price of this performance was a highly optimized, hand-tuned C++ implementation that is difficult to modify while maintaining performance.

Ziria [27], a domain-specific language for wireless PHY protocols, showed that an 802.11 PHY protocol could be written in a high-level language without sacrificing (too much) efficiency. Many of the transformations performed manually in Sora are implemented as program transformations in the Ziria compiler, wplc. For example, Sora relies critically on lookup tables (LUTs), which appear simply as large array constants in Sora’s source code with no documentation of the method by which they were generated. The wplc compiler instead generates LUTs automatically from high-level code; we describe a much-enhanced version of this LUT transformation, which was originally implemented in wplc by the first author, in Section 4. Even though much of the 802.11 PHY pipeline was written in Ziria, performance-critical blocks, including FFT, IFFT, and Viterbi, were taken from Sora and called via a foreign function interface. In this paper, we answer the following question: how can we push a high-level language further, using it to implement even speed-critical signal processing algorithms?

We show that FFT, IFFT, and Viterbi can be implemented in Ziria without sacrificing performance—in fact, our evaluation (Section 6) demonstrates that writing these components in Ziria improves performance of the overall 802.11 transmit and receive pipelines. This is accomplished through aggressive whole-program optimizations, including partial evaluation, as described in Section 4. These optimizations are implemented in a new compiler for the Ziria language, kzc, which is publicly available and fully source compatible with...
1  let (default_scrmbl_st: arr[7] bit) = 
2      ("1",0,"1","1","0","1")
3
4  fun comp scrambler(init_scrmbl_st: arr[7] bit) { 
6
7        repeat { 
8            x ← take; 
9
10           var tmp : bit; 
11
12           tmp := (scrmbl_st[3] ^ scrmbl_st[0]); 
15
16            emit (x^tmp) 
17        } 
18    }
19

Listing 1. Ziria implementation of 802.11 scrambler.

the original Ziria compiler, wplc. Concretely, our contributions are as follows:

- We show that via whole-program optimizations, implementations of FFT, IFFT, and Viterbi written in Ziria can lead to performance improvements in an 802.11 PHY pipeline relative to an implementation that relies on hand-written C++ implementations of these signal processing blocks.

- We describe a partial evaluator for the core language used by the kzc compiler. This core language has features that restrict its expressivity but in exchange allow aggressive partial evaluation. These restrictions are necessary in any case to produce efficient PHY implementations, so we lose nothing by enforcing them in the core language.

- We describe the LUT transformation for kzc’s core language. Minimizing the state required for a LUT is complicated by other transformations performed by the compiler; we show how introducing a small core language construct, views, cleanly solves the problem of bad interactions between the LUT optimization pass and other transformations.

2 Background

This section provides necessary background for the remainder of the paper by describing the Ziria surface language and some of its features. The language we describe is essentially identical to the language described by Stewart et al. [27]; this section does not represent novel work. As a language designed for modular descriptions of wireless protocols, Ziria provides imperative features for manipulating bits, bytes, etc. as well as high-level combinators for expressing communication between producer-consumer computations. We show a small fragment of the 802.11 PHY transmission pipeline—the scrambler [17, §16.2.4]—in Listing 1. Long strings of identical bits make it difficult for the receiver to distinguish individual bits; the scrambler ensures an even distribution of bits in its output by convolving its input bit stream with a known 127-bit sequence.

Ziria makes a distinction between mutable and immutable storage. In line 5, mutable storage for the scrambler’s state is allocated and initialized to the (immutable) value of the function parameter init_scrmbl_st. Line 10 allocates mutable temporary storage for a single bit. The semantics of the language dictate that if a mutable variable is not explicitly initialized at declaration, the bytes it occupies will be initialized to all zeros; the kzc compiler performs an analysis to determine when this default initialization is unnecessary, as is the case here. Immutable bindings are declared with let instead of var, as shown in line 1, which defines the default initial scrambler state.

After initializing its state, the scrambler enters an infinite loop by using the repeat construct in line 7. This loop continuously reads a bit from its input using take (line 8), computes the current bit to XOR with its input, updating its internal state in the process, and then emits the XOR’d input bit using emit in line 16. The take and emit constructs are Ziria primitives used to construct pipelines of single-producer/single-consumer data processing components. Dereferencing is implicit in Ziria, as can be seen in line 14, where tmp is implicitly dereferenced. The pure monadic core language described in Section 3 makes both reference manipulation and input/output via take and emit explicit. The kzc type checker elaborates Ziria expressions into this core language, allowing the programmer to write typical imperative programs.

Semicolon-separated blocks of code and bind (←) compose computations sequentially along the control path. The syntax for this sort of composition is deliberately reminiscent of Haskell’s do syntax, as seen in the binding form in line 8. Composition along the data path, expressed using the par operator ∘, allows producers and consumers to be chained together.

Listing 2, the decoder component from the 802.11 receiver pipeline, demonstrates the clear separation between data flow and control flow. The decoder first reads a packet header, which is always encoded at the lowest possible rate, to determine the coding rate for the rest of the packet (line 8). This information is then used from line 9 onwards to decode the remainder of the message. One of the goals of the work we describe is to enable cross-component optimizations, such as fusing the Viterbi and descramble components in lines 12 and 13, by rewriting components like Viterbi directly in Ziria, thereby exposing them to the compiler.

2.1 Language Constraints

There are notable constraints on two Ziria language constructs: references and arrays. PHY layer protocols must not only execute as quickly as possible, but they must also operate in resource-constrained environments. Ziria therefore makes every effort to avoid dynamic allocation, and garbage collection is not available. Mutable bindings declared with var scope as one would expect;
The type language includes a single, built-in monad, ST. This monad is indexed by three types: a type that indicates whether a term computes forever or terminates with a value, a type for data read by the computation, and a type for data written by the computation. We explain this construct in Section 3.1.

Functions may abstract over computations in the ST monad. We explain this feature in Section 3.2.

References are prevented from escaping their binders, as described in Section 3.3.
3.4 Typing Composition Along the Data Path

Whereas composition along the control path is reminiscent of monads [34], composition along the data path using \( \Rightarrow \) is reminiscent of arrows [16]. The rule for parallel composition, T-PAR, uses the join operator, \( \cdot \cup \cdot \), to determine the type index \( \omega \) of the result of the parallel composition. The join operator allows parallel composition of two ST computations as long as at most one of them is a terminating computer, that is, at least one of the two computations must not terminate. One could imagine allowing parallel composition of two terminating computations if they both terminated and produced a value of the same type, but this "first to finish" semantics would complicate the execution model and introduce extra synchronization. The T-PAR rule also requires that its typing context be split in order to type the two subterms. The context-splitting operator splits the reference heap; it guarantees that the two contexts \( \Gamma_1 \) and \( \Gamma_2 \) do not contain overlapping references so that the two subterms can be executed concurrently without racing on mutable reference updates—the two sides of the par can synchronize only on the data stream that connects them. See Mainland [19] for a more complete discussion of the join and context-splitting operations.

3.5 Re-Differentiating IO-Performing Computations

For the programmer, it is convenient to have a surface language that does not make a syntactic distinction between code that uses references and code that performs input/output using `take` and `emit`. However, for the compiler writer, it is useful to be able to write optimization passes over a term language that separates IO actions from all other actions. For example, the partial evaluator only evaluates expressions that do not perform any IO, and fusion operates specifically on computations that do perform IO. Because the original Ziria compiler, wpcl, uses the surface language as the compiler’s internal language instead of elaborating to a separate core language, the compiler writer’s convenience trumped programmer convenience, so the original Ziria language did enforce a syntactic distinction between reference-manipulating code and IO-performing code. In kzc, after several passes over the expression language shown in Figure 1, the compiler performs a type-directed transformation from the expression core language to a stratified computation core language, shown in Figure 3.

Programs in the stratified language have been lambda-lifted, so all function declarations occur at top-level; a program in this language consists of a series of function bindings followed by a single computation. Both expressions, e, and computations, c, are monadic—expressions may contain dereferencing and assignment, whereas all IO is performed by computations. The type system...
for this language is analogous to that previously given for the expression language. Expressions that use references, which have an ST type, are lifted to the computation level using lift. Pure expressions, which do not have an ST type, are lifted to the computation level using return.

4 Whole-Program Optimizations in kzc

In this section, we detail two optimizations performed by kzc: aggressive partial evaluation and LUT generation. These optimizations operate on expressions in the stratified core language shown in Figure 3. Mainland [19] describes in detail the fusion and pipeline coalescing, two optimizations that operate on the computation portion of the stratified language. We first give a brief overview of fusion and pipeline coalescing because they interact with partial evaluation.

4.1 Fusion and Pipeline Coalescing

The intuition behind fusion, which is a partial operation, is to fuse together producers and consumers, eliminating any communication via data streams between the two computations in a par ( ) construct; a fully fused core program will not contain par. Pipeline coalescing inserts “rate matching” transformers between producers and consumers so that the producer emits blocks of data at the same rate that the consumer takes data, allowing fusion to occur. For the 802.11 transmit and receive pipelines, fusion can eliminate all parallel composition. Fusion eliminates communication and merges modular computations into single, large computations, revealing many new opportunities for code specialization. Once pipeline coalescing and fusion have taken place, the compiler performs partial evaluation and LUT generation.

4.2 Partial Evaluation

Although the original Ziria compiler performed some partial evaluation, it was limited in what it could do by the structure of the language it was manipulating. In particular, since Ziria is impure, the value of a variable in a Ziria program may change during execution, making symbolic evaluation of a Ziria program very difficult. Because kzc operates on a pure core language instead of directly on Ziria’s surface syntax, it can and does perform symbolic partial evaluation by exploiting the invariant that any variable will always have the same value even if that value is unknown because the variable is free in the expression being partially evaluated.

The second feature of the core language that aids partial evaluation is the lack of escaping references (Section 3.3). This simplifies the problem of tracking heap effects because a reference is only live for as long as it is in scope. This makes it relatively easy to push heap effects back towards their bindings sites. A separate, small compiler pass is run after partial evaluation that can determine that a reference is only read, never written, and if this occurs, it will convert that reference into a constant. This works well because of the lack of aliasing—a reference with declared name x is only modified through x or through a function call that takes x as an argument, making for an easy analysis.

A concrete example is the BPSK interleaver, shown in Listing 3. The function interleaver_bpsk constructs two permutations by calling intlv_perm1 and intlv_perm2, and it then composes the permutations using fuse_perm. After computing the permutations, it repeatedly takes 48 bits, interleaves them using the constructed permutation, and emits the result. Although the permutation could be expressed directly as a single array constant, the executable specification given here nicely maps to the description in the 802.11 standard—it is much easier to see by inspection that the executable implementation is correct. In this case, we would appear that we don’t pay much of a performance price for using the executable specification because the permutation is calculated once outside the repeat loop. In fact, we pay no performance penalty if we invoke the partial evaluator, because it can completely evaluate all three calls to the permutation-generating functions so that the permutation provided as an argument to perm in line 34 ends up being reduced to a constant array. The original Ziria compiler is unable to perform this optimization.

Our evaluation relation has the following form:

\[ \Sigma; H; e \rightarrow H'; v \]

Here \( \Sigma \) is the environment mapping bindings to values, \( H \) is a heap mapping locations \( \ell \) to values, \( e \) is an expression, and \( v \) is a value. The value language is given in Figure 4. Values may take several forms; as well as constants \( k \), values may be top (T, the value assigned to a free variable), a reference to a heap location, or a residual expression \( e \). Values may also take the form of a

```plaintext
1 fun fuse_perm(p1 : arr int, p2 : arr[1:length(p1)] int) {
2   var p3 = arr[length(p1)] int;
3   for i in [0, length(p1)]
4     p3[i] := p2[p1[i]];
5   return p3;
6 }
7 
8 fun intlv_perm1(mod : int) {
9   var perm = arr[288] int;
10   for k in [0, 288]
11     perm[k] := nCBPS(mod)/16 * (k%16) + k/16;
12   return perm
13 }
14 
15 fun intlv_perm2(mod : int) {
16   var perm = arr[288] int;
17   let s = if (mod == M_BPSK)
18       then 1 else nBPSK(mod)/2
19   for k in [0, 288]
20     perm[k] := s * (k/s) +
21     (k+nBPSK(mod) - (16*k/nBPSK(mod))) % s;
22   return perm
23 }
24 
25 fun comp interleaver_bpsk() {
26   let nCBPSMod = 48 in
27   let ids1 = intlv_perm1(M_BPSK) in
28   let ids2 = intlv_perm2(M_BPSK) in
29   let ids3 = fuse_perm(ids1, ids2) in
30   repeat {
31     (x : arr[48] bit) ← takes 48;
32     emit perm(ids3[0, 48], x);
33   }
34 }
35 
36 
37 Listing 3. Ziria BPSK interleaver. Note that the function perm on line 34 applies a permutation, and the notation ids3[0, 48] specifies an array slice consisting of the first 48 elements of ids3.
```
command, cmd e, consisting of a residual expression representing a computation that has only heap effects. A computer, comp c, is an IO-performing residual computation (e and c are taken from Figure 3). In both cases, the heap \( H' \) in the partial evaluation judgment captures heap effects performed during partial evaluation that are not captured by the residual terms cmd e or comp c.

The fragment of the partial evaluations rules for evaluating expressions that involve references is given in Figure 5. The rule for assignment is straightforward. There are two cases for dereferencing: either the heap location maps to a non-\( \top \) value, in which case we return the value (P-Deref-Known), or it maps to \( \top \), in which case we return the dereferencing expression directly as a residual term (P-Deref-Unknown). Similarly, there are two cases for letref. If the body of the letref evaluates to a constant, we return the constant. Otherwise, we need to incorporate the heap modifications produced by evaluating the body of the letref, \( e_2 \), into the letref binding. The term \([H''[f']]\) in rule P-LetRef-Unknown interprets the value \( H''[f'] \) as an expression, thereby pushing any heap effects incurred by partially evaluating the body of the letref, \( e_2 \), back into the binding for \( x \). In cases such as the interleaver, partially evaluation can fully evaluate a letref binding. In other cases, partial evaluation fully evaluates all assignments to a reference, but there are still dereferences of the binder left over in the residual body of the letref. When this happens, the separate optimization pass mentioned earlier will recognize that the reference is never assigned, convert it into a let binding, and finally convert all dereferences of the original reference binding into (now pure) binding occurrences.

### 4.3 LUT Generation

The first author wrote the original LUT generator for Ziria, which was incorporated into wplc. The primary deficiency of this LUT generator was that it was not a source-to-source transformation; instead, it was hacked into the C back end and generated C code to initialize LUTs on start-up. The LUT generator in kzc is a source-to-source transformation that converts expressions marked for LUTting to generator expressions, which are closed terms that look like Haskell list comprehensions. Generator expressions can be evaluated at compile time using the partial evaluator to produce large constant arrays, or the C back-end can compile them to C code that initializes LUTs on start-up. Although the programmer can manually mark expressions as "LUTtable," we do not use this feature in the 802.11 PHY implementation—instead, a separate compiler pass, the auto-LUTter, traverses the code using a heuristic to look for expressions that are both small enough to LUT and that would benefit from the transformation.

The obvious way to judge whether an expression can be LUTted is to look at its free variables, determine which are read and which are written, calculate the size of the LUT that would be needed, and make a decision based on this calculation. The obvious solution turned out not to work for the 802.11 pipeline for several reasons. First, expressions that could be LUTted often operate on arrays, but they only touch part of the array. Consider, as an example, an expression in the body of a loop iterating over an entire array slice-by-slice; there are many such loops in the 802.11 pipeline. To handle such cases, the auto-LUTter relies on a range analysis, which conservatively bounds the array indices used by an expression. Originally, this analysis provided enough leverage for the auto-LUTter to operate effectively.

Introducing the fusion transformation caused the auto-LUTter to suddenly fail to mark many expressions as LUTtable. Upon inspection, we found that many previously-LUTted expressions of the form \( x[i1] \) suddenly had the form \( x[i1*k + j] \), where \( k \) was a constant, \( i \) was a variable that was free in the previously-LUTtable expression, and \( j \) was a loop index variable. The problem was that fusion introduced more complex indexing expressions in exchange for eliminating intermediate data streams, and our range analysis was not sophisticated enough to handle complex indexing expressions. Losing LUTs destroyed performance, so ignoring the problem was not an option. The natural solution seemed to be to extend the range analysis to handle more complex indexing schemes. This significantly increased the complexity of the range analysis, and we never managed to get the revised range analysis working. Instead, we introduced a small new language construct, views, that allow us to reuse the existing range analysis as-is while making it robust to any indexing transformations introduced by other compiler passes.

A view is an alias for a slice of an array, introduced by the letview construct:

\[
  e \ldots \letview v = e_1[e_{\text{init}}, e_{\text{len}}] \in e_2
\]

Here the variable \( v \) is bound to the portion of the array \( e_1 \) starting at position \( e_{\text{init}} \) and optionally having length \( e_{\text{len}} \). A new compiler pass, the view floating pass, introduces views when it sees expressions of the form \( x[i1*k + j] \), turning them into expressions of the from \( x'[j] \), where \( x' \) is bound to the slice \( x[i1*k] \). The range analysis handles view expressions without needing any changes, and a second pass lowers views back to standard arrays, eliminating views entirely. Introducing views allows the auto-LUTter to mark as many expressions as LUTtable as it did before fusion. Our ability to introduce views relies on the fact that the core language is pure. For example, we could not bind \( x' \) to the slice \( x[i1*k] \) if \( i \) or \( k \) could change in the body of the letview.

### 4.4 Optimization Contributions

The overarching moral is that choosing the right language features can enable new opportunities for optimization. In our setting, we can draw a direct connection between properties of Ziria/the core language and the optimizations they enabled:

- **Pure core expression language.** The immediate benefit of having a pure expression language is that we can perform symbolic evaluation, since the value of a variable never changes.
• Lack of escaping references. Large portions of the 802.11 pipeline rely on values that can be pre-computed. However, mapping the high-level specification contained in the standard to pre-computed tables must often be done manually. Our partial evaluator can turn large swaths of table-generating code into constants because mutable references cannot escape their binders. We note that the lack of escaping references is a property of Ziria, not just the core language. Its original purpose was to avoid any need for garbage collection.

• Pure core expression language and views. Because the core expression language is pure, we were able to add views to the core language and produce a simple, modular LUT that is insensitive to re-indexing transformations that are performed by other compiler passes.

While other partial evaluators can perform symbolic evaluation and propagate updates through the heap, the language restrictions we impose on Ziria and our core language are particularly effective at completely evaluating many large table generating computations at compile time. The lesson for language designers is that constraining the use of references can potentially enable excellent partial evaluation. The lesson for compiler writers is that choosing a pure intermediate language can make many optimizations easier because they need not worry about effects. Our partial evaluator is non-trivial, but it is self-contained and does not require any heavyweight machinery, such as an SMT solver.

Partial evaluation and LUT generation, in concert with fusion and pipeline coalescing, allow us to implement FFT, IFFT, and Viterbi in Ziria and see a performance increase relative to the pipeline using C++ versions of these signal processing blocks. In other words, aggressive whole-program optimization can more than compensate for the loss of low-level control that results from moving to a high-level language. An added benefit is that our WiFi implementations can more accurately reflect that language of the 802.11 specification, making it easier not only to write code, but also easier to understand. In general, we hope that highly effective optimizations will allow us to write future versions of Ziria that can serve as an efficient executable specification language.

5 Case Studies

To test the effectiveness of our optimizations, we reimplemented three digital signal processing blocks—FFT, inverse FFT (IFFT), and Viterbi— in Ziria. These three components were the only significant external C++ blocks in the original Ziria implementation of the 802.11 protocol. Rewriting these blocks in Ziria was more than an engineering exercise; it also provided several new insights into implementing the WiFi pipeline in a high-level language. In this section, we focus on our reimplementations of these components and share these insights.

5.1 Viterbi

The Viterbi algorithm finds the most likely sequence of hidden states in a Hidden Markov Model—called the Viterbi path—that corresponds to a sequence of observed events. The WiFi receiver uses this algorithm to find the most likely decoding of a transmitted signal that was encoded to provide redundancy. Since the future states of the Markov Model are hidden, we are forced to encode our traversal as a data-dependent feedback loop. This nonlinear recursion is the only bottleneck for a high-speed implementation [11], and it is popularly called the ACS bottleneck. Most parallelization techniques fail to speed up the ACS butterfly in a CPU implementation. Thread-based parallelism, for example, is expensive and impractical, since data dependencies limit the amount of work each thread can do independently. An alternative approach takes advantage of the algebraic properties of the ACS computation to simplify the nonlinear recursion into a linear one [11]; this transformation is referred to as the lookahead transformation. While this approach is theoretically elegant, the space usage and number of transition branches at each step makes this technique applicable primarily to hardware implementations.

The performance constraints imposed by radio protocols make implementing Viterbi an interesting challenge in a high-level language. We originally hypothesized that although we were unlikely to be able to write an implementation of a Viterbi decoder in Ziria which performed as well as Sora’s hand-written implementation, we might more than make up the difference through whole-program optimizations that would kick in once the entire pipeline was implemented in Ziria. As we began our implementation, we noticed that we could leverage partial evaluation to specialize our implementation so as to outperform Sora, even in isolation. As an example, line 12 in Listing 2 passes in the code rate to the Viterbi decoder; kzc can specialize the Viterbi decoder for individual rates, while the Sora implementation is generic over all rates. The automatic generation of these specialized functions keeps our implementation fairly small: under 300 lines of code in Ziria, compared to about 1100 lines of templated C++ in Sora. These specialized functions are crucial for performance, and they keep our performance results competitive with the hand-tuned Sora implementation.
The only manual optimization that we perform is in the representation of traceback bits, a technique present in the Sora implementation that we translated to Ziria. The traceback bits are stored using the last bit of the path metric in the trellis. Essentially, this combines the trellis and traceback array into a single unit. This requires slightly more work when computing branch metrics, requires normalizing the trellis more frequently, and adds to traceback complexity, but it makes the ACS loop—our bottleneck—faster. Other than this manual optimization, we rely on kzc’s optimizer and gcc’s vectorizer for performance.

While our Viterbi block is competitive with Sora’s Viterbi block in isolation, the performance results in our full pipeline improve significantly once Viterbi is rewritten in Ziria. This result can be explained by the fact that while Sora signal processing blocks are written to be general purpose, Ziria implementations are specialized to the specific parameters and settings in which the blocks are used. Moreover, whole-program optimizations allows optimizing components across multiple data processing blocks. For example, the final stage of the Viterbi block writes to a first-in-last-out (FILO) buffer, and the results of this FILO buffer are not available to the downstream component (the scrambler in this case, see Listing 2) until the buffer is filled. The compiler can figure this out and decides to fuse the scrambler with the Viterbi block, thus allowing the scrambler to act directly on the FILO buffer as it is filled. This optimization was not possible with the C++ implementation because the buffer used by the Sora implementation was not visible to kzc.

5.2 FFT
Sora provides hand-tuned C++ implementations of FFT and IFFT that use SSE compiler intrinsics to yield fast code. We found that in this domain we could not rely on gcc’s vectorizer to produce competitive implementations directly from high-level Ziria code. However, by expressing much of the FFT logic in Ziria and relying on external code in a few key places, our Ziria FFT and IFFT implementations expose more internal structure to kzc’s optimization passes, which leads to WiFi transmit and receive pipelines that are faster overall. Furthermore, in two cases described below, the partial evaluator could completely eliminate an inverse FFT computation because the IFFT was written purely in Ziria.

We use intrinsics in three places: a 16-point butterfly for the FFT, and 32-point and 8-point butterflies in the IFFT. In these cases, the existing Sora implementations are made available to Ziria, which has a standard interface for importing foreign code. This hand-written code makes extensive use of vector operations like shuffles, and our attempts at pure Ziria implementations defeated (or were defeated by!) gcc’s vectorizer. There are two cases in the transmitter where known, constant signals are used: the short training sequence (STS) and long training sequence (LTS). The STS and LTS signals allow a receiver to recognize valid signals and compensate for channel effects. In the original Ziria WiFi stack, these signals were recomputed on every transmission. By using a pure Ziria IFFT, our modified WiFi stack leverages the partial evaluator to completely eliminate this redundant computation.

5.3 Case Study Lessons
The original Ziria paper [27] claimed to use the Sora implementation of Viterbi because the authors did not wish to compete with highly optimized signal processing components. Our results indicate that the best performance does not come from gluing together the fastest possible signal processing blocks; rather, it comes from aggressive whole-program optimizations. Even for FFT and IFFT, where we could not maintain performance in pure Ziria, exposing more of the structure of the computation to the compiler enables cross-component optimizations and leads to faster overall pipelines as shown in the next section.

Our experience does not indicate that Ziria is somehow inherently slower than lower level languages like C; a 32-point FFT written in C would also be much slower than the Sora implementation! The key to Sora’s speed is SSE intrinsics. We plan to add high-level vector operations to Ziria, which should allow the efficient implementation of FFT butterfly computations directly in Ziria. We are also working on a prototype re-implementation of the Spiral [35] code generator for signal transforms that will be integrated with kzc.

6 Evaluation

![Figure 6. Performance of C++ and native Ziria blocks. Sora blocks are written in C++ and use SSE intrinsics. The “unoptimized” Ziria implementation has auto-LUT and partial evaluation disabled; the “optimized” Ziria implementation was compiled with both auto-LUT and partial evaluation. Error bars show one standard deviation above and below the mean.](image)

We evaluate kzc using the 802.11 WiFi implementation taken from the publicly available Ziria release [28] as well as our reimplementations of FFT, IFFT, and Viterbi written directly in Ziria. All data was collected on an i7–4770 CPU running at 3.40GHz under Ubuntu 16.04 (x64), generated C code was compiled with GCC 5.4\(^2\), all runs were repeated 100 times, and we assume run times are normally distributed. Error bars show one standard deviation above and below the mean.

Figure 6 shows the performance of the Ziria implementations of the FFT, IFFT, and Viterbi signal processing blocks. For each block, we compare the performance of the version written in Ziria, both with and without partial evaluation enabled, to the Sora

\(^2\)-march=native -mtune=native -Ofast.
implementation. Both Viterbi and IFFT perform better than the Sora implementation. Our FFT implementation is slower than Sora’s, but its performance is still reasonable. While initially surprising, our results can be partially explained by the fact that the Sora signal processing blocks are written to be general-purpose, whereas the partial evaluator is able to specialize the Ziria implementations to the specific parameters and setting in which they are used. Partial evaluation does not make a substantial difference for these blocks in isolation—there simply aren’t many opportunities for partial evaluation. Sora relies on SSE intrinsics; other than the three butterfly foreign functions mentioned in Section 5.2, the Ziria implementations rely completely on gcc’s vectorizer to exploit SIMD instructions.

The transmitter and receiver pipelines are shown in Figure 7a and Figure 7b, respectively. Each pipeline was measured at 8 data rates. In every case, we measured the performance of the original Ziria compiler, wplc, using Sora’s signal processing blocks, the performance of our new compiler, kzc, using Sora’s signal processing blocks, and the performance of our compiler using native Ziria signal processing blocks. The receiver pipeline uses the FFT and Viterbi signal processing blocks and shows the greatest speedup. The transmitter pipeline, which uses the IFFT block, still shows a performance increase. Overall data rate falls significantly in the receiver as the protocol data rate increases. This is expected and results from the different encoding/decoding algorithms used at different 802.11 data rates; higher data rates require more sophisticated—and therefore more computationally expensive—decoding algorithms. The same effect is present, though less pronounced, in the transmitter.

Comparing Figure 6 and Figure 7, it is clear that the performance increase of the Viterbi and IFFT signal blocks seen in Figure 6 cannot compensate for the slower FFT block or for the large overall increase in the performance of the entire 802.11 pipeline. The additional speedup is the result of whole-program optimizations that cut across individual data processing blocks. For example, the FILO buffer holding decoded bits in the Ziria Viterbi implementation is fused with the downstream consumer; this optimization is not possible when using the C++ implementation of Viterbi because the buffer used by the C++ code is not visible to the optimizer.

7 Related Work

7.1 SDR

Our work is based on the original Ziria compiler [27]. Although we do not reuse any code from the Ziria compiler, we evaluate our implementation using Ziria’s WiFi implementation, including its standard library routines, portions of which are borrowed from Sora. Mainland [20] describes how the Ziria surface language is elaborated to the pure, monadic core language given in Section 3. The core language and it semantics are described in detail by Mainland [19].

Most SDR platforms are based on FPGAs [21, 22], including those that support development of SDR applications on commodity CPUs [1, 10, 23, 29]. There are also numerous approaches to programming SDR applications [2, 9, 25, 26, 32], although the most commonly used is GNU Radio [4]. None of these platforms simultaneously provide both the performance and powerful abstractions needed for the SDR domain. Instead, one must choose either programmer convenience or performance. The optimizations we describe in this paper are a large step towards a language that can provide programmer convenience without sacrificing performance.

7.2 Signal Processing

de Mesmay [8] partially implemented the Viterbi algorithm using the SPIRAL framework [24]. However, they implemented only the ACS butterfly, which is similar to the FFT butterfly already supported by SPIRAL, and relied on an existing Viterbi implementation for the traceback stage. We have implemented the entire Viterbi algorithm—including the traceback stage—in Ziria. The benefit of implementing the entire Viterbi algorithm in Ziria is that the compiler can perform inter-function optimizations, like fusion, that would not be possible with a C implementation, whether or not it was machine-generated.

FFTW [13] is a specialized framework for generating only FFT implementations. The original motivation behind SPIRAL was also to generate FFTs [12, 36], although it is based on a more general computational framework [31] for expressing signal transformation. SPIRAL performs a search, which can be time consuming, over a space of implementations using algebraic rules that specify how to decompose signal transformations into smaller transforms. Our compiler is more traditional; it does not perform search, but instead
uses static analyses to guide the optimizer. We are working on incorporating SPIRAL-like techniques in our compiler.

7.3 Dataflow Languages
Languages for specifying reactive systems, particularly in hardware, include Esterel [3], LUSTRE [15], SIGNAL [14], Lucid [33], and Lucid Synchronize [5]. Streamlt [30] has been used to implement the transmitter portion of the 802.11a PHY. A key innovation of Ziria over these languages is the clean separation between control and data paths.

7.4 Fusion
The fusion transformation is described in detail by Mainland [19]. It is most closely related to the work of Coutts et al. [7], which fuses list computations by rewriting standard functional operations in terms of stepwise computations similar to take and emit. Unlike their work, fusion in kzc must handle parallel composition of producers and consumers that are not rate matched, i.e., the producer may output chunks in groups of 5, whereas the consumer consumes blocks of 8 elements at a time.

7.5 Partial Evaluation
There is a substantial literature on partial evaluation [18]. We are unaware of work that specifically addresses how to propagate heap changes backwards through a pure, monadic functional language as we do in our partial evaluator for Ziria. This is likely because many partial evaluators target existing languages instead of custom domain-specific languages with restrictions similar to those imposed by Ziria.

8 Conclusions
There is a fundamental tension between performance and control; by yielding control, a high-level language would also seem to be simultaneously giving up performance. We show that for some signal processing blocks written in Ziria, not only is it the case that one need not necessarily give up performance of the blocks in isolation, but that writing these blocks in a high-level language leads to better overall program performance by allowing the compiler to optimize across more components of a program. We have described two optimizations, a partial evaluator and LUT generator, both of which are critical to the performance of the 802.11 transmitter and receiver pipelines. Our work is fully implemented in the kzc compiler.

Future Work
Our near-term goal is to add a HDL back end to Ziria and begin moving portions of the 802.11 pipeline into hardware. While rewriting pipeline components currently implemented in C++ is necessary for such a move, it is far from clear that all of Ziria will map cleanly to hardware. However, language restrictions imposed for the benefit of resource-constrained CPU execution environments should also benefit any Ziria HDL back end. For hardware targets, Ziria’s process model maps naturally to the underlying execution model, so fusion may not be beneficial. However, it seems clear that optimizations like partial evaluation will be directly applicable to any target platform, including hardware. We also plan to write a DSP back end for TI’s TMS320C66x family, where we expect fusion will increase performance significantly.

Longer term, we hope to re-implement all of SOFDM [6] in Ziria and use that experience to make Ziria a viable language for more general hardware development. We also plan to broaden the applicability Ziria to domains such as video encoding and decoding. Finally, we plan to formalize our semantics and optimizations.

References
From High-Level Radio Protocol Specifications to Efficient Low-Level Implementations via Partial Evaluation

FHPC’17, September 7, 2017, Oxford, UK


