Better Living through Operational Semantics: An Optimizing Compiler for Radio Protocols

GEOFFREY MAINLAND, Drexel University, USA

Software-defined radio (SDR) promises to bring the flexibility and rapid iterative workflow of software to radio protocol design. Many factors make achieving this promise challenging, not least of which are the high data rates and timing requirements of real-world radio protocols. The Ziria language and accompanying compiler demonstrated that a high-level language can compete in this demanding space [Stewart et al. 2015], but extracting reusable lessons from this success is difficult due to Ziria's lack of a formal semantics. Our first contribution is a core language, operational semantics, and type system for Ziria.

The insight we gained through developing this operational semantics led to our second contribution, consisting of two program transformations. The first is *fusion*, which can eliminate intermediate queues in Ziria programs. Fusion subsumes many one-off optimizations performed by the original Ziria compiler. The second transformation is *pipeline coalescing*, which reduces execution overhead by batching IO. Pipeline coalescing relies critically on fusion and provides a much simpler story for the original Ziria compiler's "vectorization" transformation. These developments serve as the basis of our third contribution, a new compiler for Ziria that produces significantly faster code than the original compiler. The new compiler leverages our intermediate language to help eliminate unnecessary memory traffic.

As well as providing a firm foundation for the Ziria language, our work on an operational semantics resulted in very real software engineering benefits. These benefits need not be limited to SDR—the core language and accompanying transformations we present are applicable to other domains that require processing streaming data at high speed.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Compilers*; • **Theory of computation →** *Operational semantics*;

Additional Key Words and Phrases: software-defined radio, domain-specific languages, compilers, operational semantics

ACM Reference Format:
Geoffrey Mainland. 2017. Better Living through Operational Semantics: An Optimizing Compiler for Radio Protocols. *Proc. ACM Program. Lang.* 1, ICFP, Article 19 (September 2017), 26 pages.
https://doi.org/10.1145/3110263

1 INTRODUCTION

There are many platforms for software-defined radio (SDR), including some, like BladeRF, that are well-within the price range of casual hobbyists. Collectively, these platforms fulfill one half of the SDR promise—a *reconfigurable* substrate on which wireless protocols can be implemented. However, the other half of the promise—that wireless system designers can program SDR applications as easily as they can program standard software—remains unfulfilled. There are three attributes we expect from software development systems: ease of use (programmability), ability to fully utilize the capabilities of the underlying system (performance), and portability. GNU Radio [Blossom

2475-1421/2017/9-ART19
https://doi.org/10.1145/3110263

2004] offers two P's: programmability and (limited) portability. SORA [Tan et al. 2009] provides performance; because SORA allows the programmer to write in C, it would seem as though it also offers programmability, but SORA's C implementation is so highly-tuned and sensitive to changes that this is not the case. Ziria [Stewart et al. 2015] is a high-level language for wireless PHY protocols that provides both programmability and SORA-level performance. This is achieved by several compiler optimizations—in effect, instead of an expert C programmer doing the work of translating a high-level specification of a wireless PHY protocol into efficient low-level C code, the Ziria compiler does the work.

Ziria offers a promising direction, but we want more. For example, to fully exploit many SDR platforms requires programming an FPGA. There are no existing tools for accomplishing this that are specialized to the SDR domain. Both WARP [Murphy et al. 2006] and SOFDM [Chacko et al. 2014] use MATLAB's Simulink environment to program the FPGAs that underlie their respective SDR platforms. However, these models are large, difficult to construct and reason about, and they are intimately tied to platform traits such as the FPGA clock rate and the bit width of the A/D converter in the radio front end. Ideally, we could use a language like Ziria to program these devices directly, but it is far from clear how to map Ziria onto hardware. More generally, it is difficult to transfer the successes of Ziria to other application areas or hardware platforms because its successes are embodied in a software artifact; the authors of Ziria do not provide a semantics for their language or a specification of the transformations that enable it to compete with SORA. How then can we leverage high-level languages like Ziria to move the SDR field forward?

While we have not solved the problem of compiling Ziria to hardware, this paper presents a significant step forward on the path towards building high-level languages that provide programmability, performance, *and* portability in the SDR domain. The work we report grew out of an effort to create a core language and operational semantics for Ziria. Thinking in terms of this core language and operational semantics led to insights regarding many of the optimizations that the creators of Ziria report as crucial to performance. For example, we describe a general program transformation, *fusion* (Section 4), that subsumes many of the ad hoc optimizations performed by the original Ziria compiler. After developing the fusion transformation, it became clear how to cleanly implement Ziria's "vectorization" transformation. In the original compiler, vectorization was a complex source-to-source transformation, but in our formulation it is a simple transformation that leverages fusion to do the real work. Our operational semantics also led to a new compilation strategy that eliminates the overhead of the original implementation's "tick/proc" execution model. Furthermore, because our core language makes a distinction between pure and impure code, we can perform many low-level optimizations that the original compiler could not perform (Section 5). These optimizations can significantly reduce memory traffic, leading to faster implementations (Section 6).

The overarching theme of this paper is that developing a core language and operational semantics for Ziria led to *generalized* versions of high-level optimizations based on source-to-source transformations found in the original Ziria compiler and enabled our compiler to perform *new* low-level optimizations that lead to faster code. Our core language and operational semantics are not tied to the software-defined radio domain; they describe a general language for expressing resource-constrained producer-consumer computations. We expect the lessons we present here to be applicable to a wide range of domains where such computations play an important role, such as machine vision and video encoding; we discuss possible applications in Section 8. Our work is embodied in a completely new compiler, kzc, available under an MIT-style license, which uses the core language we describe internally and implements transformations, like fusion, on this core language. Our kzc compiler passes all tests in the extensive test suite included with the

```
1  let (default_scrmbl_st: arr[7] bit) = {'1,'0,'1,'1,'1,'0,'1}
2
3  fun comp scrambler(init_scrmbl_st: arr[7] bit) {
4    var scrmbl_st : arr[7] bit := init_scrmbl_st;
5
6    repeat {
7      x ← take;
8
9      var tmp : bit;
10
11     tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
12     scrmbl_st[0:5] := scrmbl_st[1:6];
13     scrmbl_st[6] := tmp;
14
15     emit (x^tmp)
16   }
17 }
```

Listing 1. Ziria implementation of 802.11 scrambler.

original Ziria compiler—it is completely compatible with the original compiler. Concretely, our contributions are as follows:

- A core language and operational semantics for Ziria that provide a foundation for both high-level and low-level optimizations. The core language and semantics are implemented in PLT Redex [Felleisen et al. 2009].
- *Fusion*, a source-to-source transformation that fuses producers and consumers, eliminating communication coordination between the two. Unlike array fusion, fusion in Ziria operates on possibly infinite streams of data, like waveforms received by a radio. It must also fuse together long producer/consumer chains in which different processes may consume data at mismatched input and output rates. For example, it must handle cases like a producer that yields one bit at a time to a consumer that expects input to arrive in byte-sized chunks.
- *Rate analysis*, which approximates the shape of the streams consumed and produced by a computation, and *pipeline coalescing*, which improves performance by coalescing communication between producers and consumers.
- Low-level memory optimizations, enabled by the intermediate language, that reduce memory traffic. This includes optimizations that in many cases eliminate data copies between producers and consumers.
- The kzc compiler, an open source implementation of the core language and optimizations we describe, which is completely compatible with the original Ziria implementation.

2 BACKGROUND

We first give a brief overview of Ziria (the surface language) to provide the context necessary for describing transformations and optimizations later in the paper. This section does not present novel work; the surface language we use is almost identical to the language Stewart et al. describe.

Ziria is a two-level language. At its core is a typical (functional) imperative language. Layered on top of this core is a language for producer-consumer computations that communicate via queues. Listing 1 shows the Ziria implementation of the scrambler as described in the 802.11 specification [IEEE 2012, §16.2.4]. The scrambler randomizes the bits being transmitted on a radio channel to avoid long strings of ones or zeros, which can impair detection at the receiver. The

Ziria implementation consumes bits one-by-one using **take** (line 7), convolves each bit with the scrambler's state (lines 9 through 13), and then produces an output bit using **emit** (line 15). Mutable storage is allocated with **var**, as shown on lines 4 and 9. Ziria also includes a **let** binding form for allocating immutable storage, as shown on line 1 (`default_scrmbl_st` is the initial scrambler state used by the transmitter). The **repeat** construct (lines 6 through 16) repeats a computation—the body of the **repeat**—forever. A computation that continually consumes input, transforms it, and produces output without terminating, like the scrambler, is a *stream transformer*.

2.1 Composition Along the Control Path and Data Path

Ziria makes a distinction between the control path and the data path. Composition along the control path is performed by sequencing code with a semicolon. The scrambler uses composition along the control path to read a value from its input stream, perform a computation with that value, and yield the result to its output stream. Once we have the scrambler, we need to compose it somehow with other producer-consumer components; in the context of an 802.11 pipeline, we will want to feed the bits we wish to transmit first to the scrambler and then hand the output of the scrambler to the producer-consumer component that encodes bits in a format that is appropriate for a radio front end. This kind of composition is composition along the *data path* and is performed using the *par* operator, ⋙.

```
repeat { x ← take; emit (x + 1); }
⋙
{
  var x : int := 0;
  for i in [0, 8] { y ← take; x := x + y; };
  return x;
}
```

Listing 2. Composition along the data path.

Listing 2 shows composition along both the control and data paths. The first component in the data path is a stream transformer that adds one to every element in the data stream. The second element in the data path reads 8 elements from the data stream and terminates, returning the sum of the elements it read. Because it terminates with a final value on the control path, it is a *stream computer*. The distinction between a *stream transformer* and a *stream computer* is apparent in their types. Terms in the producer-consumer level of the language have types of the form ST ω α β. ST is the *stream* monad, and it is indexed by three types. The stream's input type is α, and its output type is β. The type ω is either T for a *stream transformer* or C δ for a *stream computer*, where δ is the type of the final control value produced when the stream computer terminates. In Listing 2, the first argument to ⋙ has type ST T int int, and the second argument has type ST (C int) int α for any type α since it never emits a value to its output data stream. The entire computation also has type ST (C int) int α. We detail the type system in the following section.

2.2 Pipeline Parallelism and the Subtleties of Executing Producer-Consumer Pipelines

One possibility for executing par constructs is to run the producer and consumer in different threads, thereby obtaining pipeline parallelism and speeding up the overall computation. There are several subtleties involved in executing par, pipelined or otherwise, which are reflected in the semantics we give in the following section. The root of the problem lies with par constructs that execute in sequence with other computations. Consider the following code block:

```
{
  x ← t1 >>> c2;
  c3
}
```

In this block, t_1 is a stream transformer and c_2 and c_3 are stream computers. We would like to have the option of running t_1 "freely," allowing it to consume data as fast as possible and queue the transformed data for consumption by c_2. However, we know that c_2 will eventually terminate—what should happen to the data that t_1 has queued for consumption by c_2 in the mean time? The answer is that t_1 *never should have consumed* the input used to produce the contents left in the queue when c_2 terminates—that input data should instead be consumed by c_3! Execution of the par construct is therefore driven by the consumer; the upstream producer (t_1 here) is not allowed to request data from *its* producer until its downstream consumer (c_2 here) requests data. Concretely, t_1 may not execute **take** until c_2 has executed **take** *and the queue between* t_1 *and* c_2 *is empty*. This restriction only applies when a par construct is in sequence with another producer-consumer computation, i.e., when it is involved in composition along the *control* path. In this example, we only need to worry about t_1 "over-consuming" because it is part of a computation that will terminate and pass control to c_3. In contrast, we allow the producer in a top-level par to run freely. If the top-level pipeline is a computer rather than a transformer, this may cause the pipeline to consume more input than is strictly necessary to produce a final control value, but our choice allows top-level producer-consumer computations to be executed in parallel.

2.3 Efficient Execution of Ziria Programs

Listings 1 and 2 exhibit the "high-level" nature of Ziria; the scrambler is a direct, readable, translation of the 802.11 specification, and the par snippet demonstrates the pleasingly compositional nature in which producer-consumer computations can be constructed. But what is the cost? No real-world scrambler implementation would operate bitwise in this way because doing so is too slow. Like the original Ziria compiler, we implement a LUT transformation that converts code like the scrambler into lookup tables. For the scrambler, the Ziria computation is compiled into an implementation that operates on whole bytes using a single LUT lookup whose index is the current scrambler state and input byte and whose output is the new scrambler state and scrambler output. We do not discuss the LUT transformation further; it differs from the original compiler's LUT transformation primarily in that it is a source-to-source transformation on the core language rather than being baked in to the C back-end.

The overhead introduced by many small producer-consumer computations composed together to form a pipeline is more challenging to address. This overhead arises from the co-routine-style execution model for chaining together producers and consumers (Section 3.4) because switching execution between the producer and consumer requires either jumping through a pointer (if we can utilize first-class labels) or "jumping" via a trampoline. The fusion and pipeline coalescing transformations we describe in Section 4 eliminate this overhead. For the 802.11 receiver and transmitter pipelines, they eliminate *all* par constructs, producing a single **repeat** construct that consumes input and produces output only at the top level. In general, not all producers and consumers can be fused; for these cases, our runtime provides *queue-free* serial execution of producers and consumers. We describe this feature of our runtime in Section 5.

2.4 Relationship to the Original Ziria Language

The original Ziria language makes a syntactic distinction between the two language levels using the **seq** keyword to sequence code in the imperative core and the **do** keyword to sequence code that

e, c	::=	k	(constant)	τ	::=	()	base types
	|	x	(variable)		|	bool	
	|	$\circ_1\ e$	(unary operator)		|	int	
	|	$e\ \circ_2\ e$	(binary operator)	ω	::=	C τ	
	|	if e then e else e			|	T	
	|	let $x = e$ in e		μ	::=	ST $\omega\ \tau\ \tau$	computation types
	|	letfun $f\ (\overline{x_i : \rho_i}) : \sigma = e$ in e		ρ	::=	τ	function arguments
	|	letref $x = e$ in e			|	ref τ	
	|	$f\ e_1\ \ldots\ e_n$		σ	::=	τ	function result
	|	$!x$	(dereference)		|	μ	
	|	$x := e$	(assignment)	ϕ	::=	ρ	types bound in Γ
	|	return e			|	$\rho_1 \ldots \rho_n \to \sigma$	
	|	$x \leftarrow c \fatsemi c$	(bind)	Γ	::=	$\cdot$	
	|	$c \fatsemi c$	(sequence)		|	$x : \phi, \Gamma$	
	|	take		θ	::=	τ	result of typing relation
	|	emit e			|	ref τ	
	|	repeat c			|	μ	
	|	$c \ggg c$	(par)		|	$\rho_1 \ldots \rho_n \to \sigma$	

Fig. 1. The core language.

performs producer-consumer IO. Our surface language captures the distinction between the two language levels purely in the types of terms. We use the type encoding described by Mainland [2017] to make this distinction explicit in the core language. However, unlike Mainland, we have modified the surface language to eliminate the awkward syntactic **seq**/**do** distinction while retaining full backwards compatibility with existing Ziria code. Although our new parser can parse all existing Ziria code as well as code written without the **seq**/**do** distinction (it parses a strict superset of the original Ziria language), we provide a second parser that accepts only "classic" Ziria—unfortunately, the grammars were different enough that we could not combine the two parsers. Both parsers use the same abstract syntax data type.

3 CORE LANGUAGE AND OPERATIONAL SEMANTICS

The core language and operational semantics we present in this section[1] are transliterated from a formalization encoded in PLT Redex [Felleisen et al. 2009]. The core language collapses all terms into a single syntactic category of expressions. The distinction between terms that perform input and output and those that do not is maintained in the type system. In our compiler, the first step after parsing is type inference and elaboration of the surface language directly into the core expression language. As well as originally requiring programmers to manually choose between sequencing imperative code and IO code with **seq** and **do**, Ziria includes multiple surface language binding forms that must distinguish between, e.g., functions that perform IO and those that do not. Our compiler's new parser and elaboration phase make it clear that as well as being unnecessary in the core language, these distinctions are unnecessary in the surface language. Eliminating them simplifies type checking as well as source-to-source transformations, leaving both to cope with fewer cases. Full compatibility with existing Ziria code is maintained by parsing a more general grammar that collapses previously distinct constructs into a single syntactic category of expressions.

[1] Available at https://github.com/mainland/kyllini-semantics

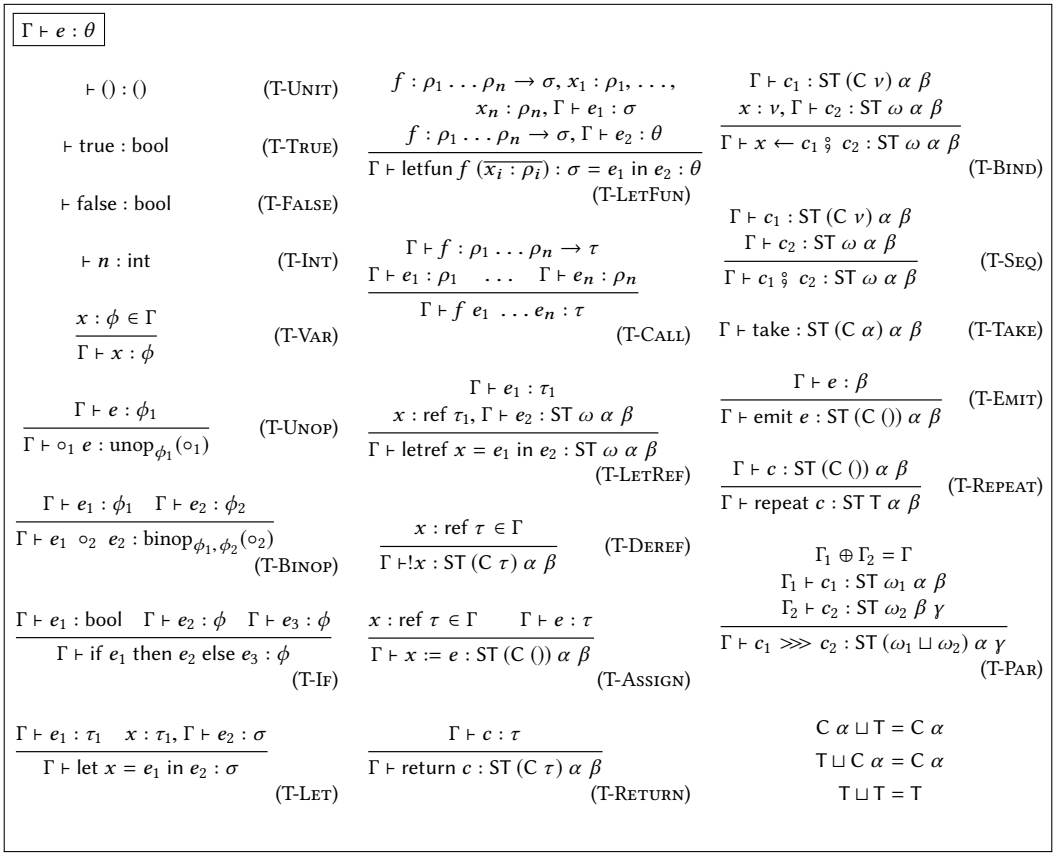


Fig. 2. Typing relation for core.

Our core language is given in Figure 1. It is a standard *pure* first-order functional language with a single monad, the ST monad, and it includes the standard monadic bind and sequence operations. It has two special binding forms: one for mutable references, letref, and one for functions, letfun. Dereferencing, $!x$, and assignment to a reference, $x := e$, are explicit. The novel language constructs are take, emit, repeat, and $\ggg$. The actual implementation uses a core language that supports additional features that are straightforward to add, which we describe briefly in Section 3.5.

3.1 Types and the Typing Relation

The type system for core, shown in Figure 2, is a typical type system augmented with support for the producer-consumer operations take, emit, repeat, and $\ggg$. Base types, τ, are as expected. Computation types, represented by μ, are of the form ST $\omega\ \alpha\ \beta$. Function arguments, ρ, may have type τ or ref τ. Function results, σ, may be of type τ or μ; that is, functions may return computations. The typing context may contain values of the form ρ (base types and refs) or $\rho_1 \ldots \rho_n \to \sigma$ (functions). The typing judgment assigns most pure expressions a type of the form θ, which may be a base type, ref type, computation, or function. The exceptions are the rule T-Var (a variable may be bound to a function) and T-LetRef.

The type system is careful to bound the scope of references: T-LetRef disallows escaping references, and T-LetFun prevents a function from returning a reference. The type system not

only prevents references from escaping from an inner scope, but also prevents aliasing since T-LetRef only allows a reference to be initialized with a value. The only way aliasing between references can be introduced is by passing the same reference multiple times as an argument to a function. Eliminating this form of aliasing is impractical as the WiFi pipeline often needs to operate simultaneously on different portions of the same mutable array, which may in general overlap. Nonetheless, the fact that references cannot escape greatly simplifies both the compilation strategy and memory management since we do not need a garbage collector.

3.2 Typing ≫

The typing rule T-Par deserves explanation. First, note that unlike every other rule, the subterms c_1 and c_2 may have different input and output types. This is because par is the only construct that allows composition on the data path; unlike all other rules for composing computations, c_1 and c_2 do not both **take** (resp. **emit**) from the same data source (resp. sink), but c_2 **take**'s what c_1 **emit**'s. This rule also makes use of the join operator, $\cdot \sqcup \cdot$, to determine the ω type index of the result of the par. It guarantees that two stream transformers may be composed on the data path, as may a stream computer and a stream transformer, but it prevents two stream computers from being composed along the data path. We could imagine adding a fourth case to the join operator, $\mathsf{C}\ \alpha \sqcup \mathsf{C}\ \alpha = \mathsf{C}\ \alpha$, but this complicates the semantics as it requires additional synchronization on the final computed result. This change would also complicate the implementation; with the current semantics, we are guaranteed that at most one side of the par will ever terminate and call the par's continuation.

The second new operator in the T-Par rule is the context splitting operation, $\oplus$. The operational semantics runs the two sides of a par in separate threads; how then can we prevent race conditions? The answer is that the context splitting operation $\oplus$ splits the portion of the context that contains variables that have type ref τ, leaving the rest of the context as-is. In T-Par, this ensures that Γ_1 and Γ_2 contain completely distinct sets of references, i.e., no reference occurs in both environments, so there can be no race condition[2]. Note that the operational semantics does not perform a store splitting operation analogous to the context splitting operation because it only need guarantee that *well-typed* terms don't race.

3.3 Operational Semantics

At a high level, the goal of the operational semantics is to define how Ziria programs can run in parallel while preventing the overconsumption issue raised in Section 2.2. This leads us to a demand-driven execution model, where computation in upstream producers is driven by demand in downstream consumers. When the queue between a producer and consumer is empty and the consumer requests an element using **take**, the producer may execute long enough to produce an element. Because producers are "gated" by consumer demand, parallelism is necessarily limited. We decouple threads from queues via *actions*, which mediate threads' interactions with the outside world via dataflow. The reduction relation is formulated as a CESK-style abstract machine [Felleisen 1987] whose evaluation contexts are shown in Figure 3. The environment, E, maps variables to values or locations, and the store, Σ, maps locations, ℓ, to values or closures. The only values are constants. Closures, denoted clo, contain a list of function parameters, a function body, and an environment.

The δ actions in Figure 3 allow producers and consumers to synchronize on queues. There are four possible such actions: tick, which indicates that a non-IO action was performed, wait, which indicates that a thread is waiting for input, cons v, which indicates that the value v is available

[2]The implementation inserts a memory barrier before spawning a thread to guarantee consistent reads from shared state in both sides of the par.

$$
\begin{array}{lcl}
\kappa & ::= & \text{halt} \\
& | & \text{popk } E\ \kappa \\
& | & \text{unopk } \circ_1\ \kappa \\
& | & \text{binop1k } \circ_2\ e\ \kappa \\
& | & \text{binop2k } \circ_2\ v\ \kappa \\
& | & \text{ifk } e\ e\ \kappa \\
& | & \text{letk } x\ e\ \kappa \\
& | & \text{argk } f\ (v_1^{\text{env}}, \ldots, v_{i-1}^{\text{env}})\ (e_{i+1}, \ldots e_n)\ \kappa \\
& | & \text{letrefk } x\ e\ \kappa \\
& | & \text{setk } x\ \kappa \\
& | & \text{returnk } \kappa \\
& | & \text{bindk } x\ e\ \kappa \\
& | & \text{seqk } e\ \kappa \\
& | & \text{emitk} \\
v & ::= & k \\
\text{clo} & ::= & \langle x_1, \ldots, x_n; e; E \rangle \\
v^{\text{env}} & ::= & k \mid \ell \\
v^{\text{sto}} & ::= & k \mid \text{clo}
\end{array}
$$

$$
\begin{array}{lcl}
E & ::= & \emptyset \\
& | & x = v^{\text{env}}, E \\
\Sigma & ::= & \emptyset \\
& | & \ell = v^{\text{sto}}, \Sigma \\
\delta & ::= & \text{tick} \\
& | & \text{wait} \\
& | & \text{cons } v \\
& | & \text{yield } v \\
t & ::= & \langle E; \Sigma; e; \kappa; \delta \rangle \\
Q & ::= & \text{queue } v \ldots \\
p & ::= & \ll \langle E; \Sigma; e; \kappa; \delta \rangle ; q_i; q_o \gg \\
\Phi & ::= & \emptyset \\
& | & q = Q, \Phi \\
X & ::= & \emptyset \\
& | & q, X
\end{array}
$$

Fig. 3. Evaluation contexts.

for the thread to consume, and yield v, which indicates that the thread has produced the value v. Formulating thread reduction using δ actions allows us to decouple it from the reduction relation that specifies how collections of threads—machines—step. Although we do not do so here, we could give a single-threaded machine reduction relation without having to rewrite the thread reduction relation.

The full relation is given in two parts: a thread reduction relation given in Figure 4, grouped roughly by syntactic form, and a machine reduction relation given in Figure 5. Both figures directly reflect the PLT Redex model. The reduction relation for an individual thread is a standard CESK-style abstract machine except for δ actions. Threads step according to the relation

$$\langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle \longrightarrow \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle$$

Threads t consist of an environment, a store, an expression under evaluation, a continuation κ, and an *action* δ. The simplest forms, involving variables, unary and binary operators, and if-then-else expressions, are shown in Figure 4a. When subexpressions move into evaluation position, an appropriate continuation is constructed so that the result of evaluation is properly dealt with.

The reduction relation produces a value in one of two forms: a pure value v, or a monadic value having the form return v—the semantics differentiates between pure values and monadic values. These forms show up in the E-PopK and E-PopReturnK rules, respectively, in Figure 4b. The popk $E'\ \kappa$ continuation represents "popping" a saved environment and calling the continuation κ. It is used whenever a new scope is entered: in the body of a let construct, in the body of a bind construct, and when evaluating the body of a function. The environment is then popped when the current expression evaluates to a value (E-PopK) or when it evaluates to a monadic return (E-PopReturnK).

The rules for references, in Figure 4c, manipulate the store, Σ, as one would expect. The rules for sequencing, in Figure 4d, are also straightforward. The rules for function calls (Figure 4e) are complicated by the need to pass references. Note that the only expression that can evaluate to a reference is x, where x is bound by a letref. Two rules, E-ArgRefK and E-CallRefK, handle the

$\boxed{\langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle \to \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle}$

$$\langle E; \Sigma; x; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; v; \kappa; \mathsf{tick} \rangle \quad \text{where } v = E[x] \qquad \text{(E-Var)}$$

$$\langle E; \Sigma; \circ_1\ e; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e; \mathsf{unopk}\ \circ_1\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Unop)}$$

$$\langle E; \Sigma; v; \mathsf{unopk}\ \circ_1\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; [\![\circ_1 v]\!]; \kappa; \mathsf{tick} \rangle \qquad \text{(E-UnopK)}$$

$$\langle E; \Sigma; e_1\ \circ_2\ e_2; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_1; \mathsf{binopk1}\ \circ_2\ e_2\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Binop)}$$

$$\langle E; \Sigma; v_1; \mathsf{binop1k}\ \circ_2\ e_2\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_2; \mathsf{binop2k}\ \circ_2\ v_1\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Binop1K)}$$

$$\langle E; \Sigma; v_2; \mathsf{binop2k}\ \circ_2\ v_1\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; [\![v_1\ \circ_2\ v_2]\!]; \kappa; \mathsf{tick} \rangle \qquad \text{(E-Binop2K)}$$

$$\langle E; \Sigma; \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_1; \mathsf{ifk}\ e_2\ e_3\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-If)}$$

$$\langle E; \Sigma; \mathsf{true}; \mathsf{ifk}\ e_1\ e_2\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_1; \kappa; \mathsf{tick} \rangle \qquad \text{(E-IfTrueK)}$$

$$\langle E; \Sigma; \mathsf{false}; \mathsf{ifk}\ e_1\ e_2\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_2; \kappa; \mathsf{tick} \rangle \qquad \text{(E-IfFalseK)}$$

(a) Base forms

$$\langle E; \Sigma; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_1; \mathsf{letk}\ x\ e_2\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Let)}$$

$$\langle E_1; \Sigma; v; \mathsf{letk}\ x\ e_2\ \kappa; \mathsf{tick} \rangle \to \langle E_2; \Sigma; e_2; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \quad \text{where } E_2 = E_1[x \mapsto v] \qquad \text{(E-LetK)}$$

$$\langle E; \Sigma; x \leftarrow e\ ;;\ c; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e; \mathsf{bindk}\ x\ c\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Bind)}$$

$$\langle E_1; \Sigma; \mathsf{return}\ v; \mathsf{bindk}\ x\ c\ \kappa; \mathsf{tick} \rangle \to \langle E_2; \Sigma; c; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \quad \text{where } E_2 = E_1[x \mapsto v] \qquad \text{(E-BindK)}$$

$$\langle E; \Sigma; v; \mathsf{popk}\ E'\ \kappa; \mathsf{tick} \rangle \to \langle E'; \Sigma; v; \kappa; \mathsf{tick} \rangle \qquad \text{(E-PopK)}$$

$$\langle E; \Sigma; \mathsf{return}\ v; \mathsf{popk}\ E'\ \kappa; \mathsf{tick} \rangle \to \langle E'; \Sigma'; \mathsf{return}\ v; \kappa; \mathsf{tick} \rangle \qquad \text{(E-PopReturnK)}$$

$$\langle E_1; \Sigma_1; \mathsf{letfun}\ f\ (x_1 : \rho_1, \ldots, x_n : \rho_n) : \sigma = e_1\ \mathsf{in}\ e_2; \kappa; \mathsf{tick} \rangle \to \langle E_2; \Sigma_2; e_2; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \quad \text{where } \ell \text{ fresh}, E_2 = E_1[f \mapsto \ell], \Sigma_2 = \Sigma_1[\ell \mapsto \langle x_1, \ldots, x_n; e_1; E_2 \rangle] \qquad \text{(E-LetFun)}$$

(b) Binding forms

$$\langle E; \Sigma; \mathsf{letref}\ x = e_1\ \mathsf{in}\ e_2; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e_1; \mathsf{letrefk}\ x\ e_2\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-LetRef)}$$

$$\langle E_1; \Sigma_1; v; \mathsf{letrefk}\ x\ e_2\ \kappa; \mathsf{tick} \rangle \to \langle E_2; \Sigma_2; e_2; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \quad \text{where } \ell \text{ fresh}, E_2 = E_1[x \mapsto \ell], \Sigma_2 = \Sigma_1[\ell \mapsto v] \qquad \text{(E-LetRefK)}$$

$$\langle E; \Sigma; !x; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; \mathsf{return}\ v; \kappa; \mathsf{tick} \rangle \quad \text{where } \ell = E[x], v = \Sigma[\ell] \qquad \text{(E-Deref)}$$

$$\langle E; \Sigma; x := e; \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma; e; \mathsf{setk}\ x\ \kappa; \mathsf{tick} \rangle \qquad \text{(E-Set)}$$

$$\langle E; \Sigma_1; v; \mathsf{setk}\ x\ \kappa; \mathsf{tick} \rangle \to \langle E; \Sigma_2; \mathsf{return}\ (); \kappa; \mathsf{tick} \rangle \quad \text{where } \ell = E[x], \Sigma_2 = \Sigma_1[\ell \mapsto v] \qquad \text{(E-SetK)}$$

(c) References

Fig. 4. Thread reduction relation.

case of reference arguments specially, ensuring that when a function is called, the appropriate parameter is bound to the reference's location in the store.

Almost all transitions in the thread reduction relation are internal: they are only allowed to run when the δ action is tick, and they do not change the δ action. Only when evaluating **`take`** and **`emit`** using the rules in Figure 4f does the action component of the thread state change. In E-TakeWait, a thread in the tick action state transitions to the wait action state, but the current expression under evaluation remains **`take`**. The transition to the wait action state allows the machine reduction relation to then either consume an element from this thread's upstream queue or run this thread's upstream producer if the upstream queue is empty. When an element v becomes available in the thread's upstream queue, the machine reduction relation will change the action from wait to cons v. The thread can then resume computation by *consuming* the value v in E-TakeConsume. Emitting a value v occurs via E-EmitK, which transitions the thread to the yield v action. The thread is not allowed to resume computation until the machine reduction relation places the value v on the thread's output queue and changes the thread's action back to tick, which will happen only

$\boxed{\langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle \rightarrow \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle}$

$$\langle E; \Sigma; c_1 \fatsemi c_2; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; c_1; \mathsf{seqk}\ c_2\ \kappa; \mathsf{tick} \rangle \quad \text{(E-Seq)}$$

$$\langle E; \Sigma; \mathsf{return}\ v; \mathsf{seqk}\ c\ \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; c; \kappa; \mathsf{tick} \rangle \quad \text{(E-SeqK)}$$

$$\langle E; \Sigma; \mathsf{return}\ e; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; e; \mathsf{returnk}\ \kappa; \mathsf{tick} \rangle \quad \text{where } e \text{ is not a value} \quad \text{(E-Return)}$$

$$\langle E; \Sigma; v; \mathsf{returnk}\ \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; \mathsf{return}\ v; \kappa; \mathsf{tick} \rangle \quad \text{(E-ReturnK)}$$

$$\langle E; \Sigma; \mathsf{repeat}\ c; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; c \fatsemi \mathsf{repeat}\ c; \kappa; \mathsf{tick} \rangle \quad \text{(E-Repeat)}$$

(d) Sequencing

$$\langle E; \Sigma; f\ e_1 \ldots e_n; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; e_1; \mathsf{argk}\ f\ ()\ (e_2, \ldots, e_n)\ \kappa; \mathsf{tick} \rangle \quad \text{(E-Call)}$$

$$\begin{array}{c} \langle E; \Sigma; v_i; \mathsf{argk}\ f\ (v_1^{\mathsf{env}}, \ldots, v_{i-1}^{\mathsf{env}})\ (e_{i+1}, \ldots e_n)\ \kappa; \mathsf{tick} \rangle \rightarrow \\ \langle E; \Sigma; e_{i+1}; \mathsf{argk}\ f\ (v_1^{\mathsf{env}}, \ldots, v_{i-1}^{\mathsf{env}}, v_i)\ (e_{i+2}, \ldots e_n)\ \kappa; \mathsf{tick} \rangle \end{array} \quad \text{(E-ArgK)}$$

$$\begin{array}{c} \langle E; \Sigma; x; \mathsf{argk}\ f\ (v_1^{\mathsf{env}}, \ldots, v_{i-1}^{\mathsf{env}})\ (e_{i+1}, \ldots e_n)\ \kappa; \mathsf{tick} \rangle \rightarrow \\ \langle E; \Sigma; e_{i+1}; \mathsf{argk}\ f\ (v_1^{\mathsf{env}}, \ldots, v_{i-1}^{\mathsf{env}}, \ell)\ (e_{i+2}, \ldots e_n)\ \kappa; \mathsf{tick} \rangle \\ \text{where } \ell = E[x] \end{array} \quad \text{(E-ArgRefK)}$$

$$\begin{array}{c} \langle E_1; \Sigma; v_n; \mathsf{argk}\ f\ (v_1, \ldots, v_{n-1})\ ()\ \kappa; \mathsf{tick} \rangle \rightarrow \langle E_2; \Sigma; e_f; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \\ \text{where } \ell_f = E_1[f] \\ \langle x_1, \ldots, x_n; e_f; E_f \rangle = \Sigma[\ell_f] \\ E_2 = E_f[\overline{x_i \mapsto v_i}] \end{array} \quad \text{(E-CallK)}$$

$$\begin{array}{c} \langle E_1; \Sigma; x; \mathsf{argk}\ f\ (v_1, \ldots, v_{n-1})\ ()\ \kappa; \mathsf{tick} \rangle \rightarrow \langle E_2; \Sigma; e_f; \mathsf{popk}\ E_1\ \kappa; \mathsf{tick} \rangle \\ \text{where } \ell = E_1[x] \\ \text{where } \ell_f = E_1[f] \\ \langle x_1, \ldots, x_n; e_f; E_f \rangle = \Sigma[\ell_f] \\ E_2 = E_f[x_1 \mapsto v_1, \ldots, x_{n-1} \mapsto v_{n-1}, x_n \mapsto \ell] \end{array} \quad \text{(E-CallRefK)}$$

(e) Function calls

$$\langle E; \Sigma; \mathsf{take}; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; \mathsf{take}; \kappa; \mathsf{wait} \rangle \quad \text{(E-TakeWait)}$$

$$\langle E; \Sigma; \mathsf{take}; \kappa; \mathsf{cons}\ v \rangle \rightarrow \langle E; \Sigma; \mathsf{return}\ v; \kappa; \mathsf{tick} \rangle \quad \text{(E-TakeConsume)}$$

$$\langle E; \Sigma; \mathsf{emit}\ e; \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; e; \mathsf{emitk}\ \kappa; \mathsf{tick} \rangle \quad \text{(E-Emit)}$$

$$\langle E; \Sigma; v; \mathsf{emitk}\ \kappa; \mathsf{tick} \rangle \rightarrow \langle E; \Sigma; \mathsf{return}\ (); \kappa; \mathsf{yield}\ v \rangle \quad \text{(E-EmitK)}$$

(f) Input/output

Fig. 4. Thread reduction relation (cont'd).

when the thread's downstream queue is empty and its downstream consumer demands a value. The machine reduction relation uses threads' δ actions to coordinate IO.

Machines step according to the relation

$$\| \Phi_1; X_1; \ldots, \ll \langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle ; q_i; q_o \gg, \ldots \| \Rightarrow \| \Phi_2; X_2; \ldots, \ll \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle ; q_i; q_o \gg, \ldots \|$$

A machine consists of a *queue store*, Φ, that maps queue locations q to queues, a *wait set* X, the set of queues waiting for input, and a collection of processes. Queues can contain zero or more values. A process, p, consists of a thread, an input queue, and an output queue. P-Thread allows any process to take a step as long as its thread can step. This both allows internal thread transitions and enables thread interaction with queues via the E-TakeWait, E-TakeConsume, and E-EmitK thread transition rules in Figure 4f.

The P-Wait rule demonstrates what happens when a thread attempts to **take** a value and its input queue is empty, causing the thread to transition via E-TakeWait to the δ action wait. When the thread enters the wait state, the machine can transition via P-Wait so that the input queue of the process to which the thread belongs is added to the wait set X. This allows its upstream producer to begin consuming input. When the input queue is not empty, P-Consume allows a thread

$$\| \Phi_1; X_1; \ldots, \ll \langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle ; q_i; q_o \gg, \ldots \| \Rightarrow \| \Phi_2; X_2; \ldots, \ll \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle ; q_i; q_o \gg, \ldots \|$$

$$\frac{\langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle \rightarrow \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle}{\| \Phi; X; \ldots \ll \langle E_1; \Sigma_1; e_1; \kappa_1; \delta_1 \rangle ; q_i; q_o \gg \ldots \| \Rightarrow \| \Phi; X; \ldots \ll \langle E_2; \Sigma_2; e_2; \kappa_2; \delta_2 \rangle ; q_i; q_o \gg \ldots \|} \quad \text{(P-Thread)}$$

$$\| \Phi; X; \ldots \ll \langle E; \Sigma; e; \kappa; \text{wait} \rangle ; q_i; q_o \gg \ldots \| \Rightarrow \| \Phi; q_i, X; \ldots \ll \langle E; \Sigma; e; \kappa; \text{wait} \rangle ; q_i; q_o \gg \ldots \| \quad \text{(P-Wait)}$$
where q_i empty, $q_o \in X$ or q_o is final output queue, $q_i \notin X$

$$\| \Phi_1; X; \ldots \ll \langle E; \Sigma; e; \kappa; \text{wait} \rangle ; q_i; q_o \gg \ldots \| \Rightarrow \| \Phi_2; X; \ldots \ll \langle E; \Sigma; e; \kappa; \text{cons } v \rangle ; q_i; q_o \gg \ldots \| \quad \text{(P-Consume)}$$
where $q_o \in X$ or q_o is final output queue, $(\Phi_2; v) = \text{dequeue}(\Phi_1; q_i)$

$$\| \Phi; X; \ldots \ll \langle E; \Sigma; e; \kappa; \text{yield } v \rangle ; q_i; q_o \gg \ldots \| \Rightarrow \| \text{enqueue}(\Phi; q_o; v); X \setminus q_o; \ldots \ll \langle E; \Sigma; e; \kappa; \text{tick} \rangle ; q_i; q_o \gg \ldots \|$$
where $q_o \in X$ or q_o is final output queue

(P-Yield)

$$\| \Phi; X; \ldots \ll \langle E; \Sigma; e_1 \ggg e_2; \kappa; \text{tick} \rangle ; q_i; q_o \gg \ldots \| \Rightarrow$$
$$\| q = \text{queue}, \Phi; X; \ldots \ll \langle E; \Sigma; e_1; \kappa; \text{tick} \rangle ; q_i; q \gg, \ll \langle E; \Sigma; e_2; \kappa; \text{tick} \rangle ; q; q_o \gg \ldots \| \quad \text{(P-Spawn)}$$
where q fresh

Fig. 5. Machine reduction relation.

to consume as long as its *output* queue is in the wait set X, i.e., if its downstream consumer requires input. These rules together prevent overconsumption (Section 2.2). The P-Yield rule enqueues a value on a process' output queue and removes the queue from the wait set. The associated thread is placed in the tick action state and may run freely until it attempts to perform IO again. Finally, P-Spawn splits a par computation by creating a new thread and a new intermediate queue to connect the left and right sides of the par.

Our semantics limits overconsumption to ensure correctness. If we allow the entire machine to consume more input than is strictly necessary to produce as much output as it can—for example, if there are 129 bits available to the machine and it needs only 128 to produce a "packet", but we allow the machine to read all 129—then we can increase parallelism by allowing "top-level" processes to run freely. Our implementation does this, but we have not modeled this in the semantics. It would also be possible to design a semantics that allowed aggressive consumption in the interest of increasing parallelism but then "rolled back" previously consumed values if it turned out that overconsumption occurred. We leave exploration of such a mechanism to future work. We suspect it will be difficult to design a system where increased parallelism more than pays for the additional cost of the bookkeeping necessary to implement rollback; this amortization may be easier to accomplish on hardware platforms.

3.4 Implementing the Operational Semantics

The operational semantics give Ziria programs a well-defined interpretation, but implementing the semantics directly would lead to extremely inefficient programs—we would end up with a large number of threads, few of which do enough work to amortize the cost of inter-thread coordination. Instead, we implement both the threaded semantics given here and a sequential semantics, which relates strictly fewer machines than the threaded semantics. It is a small step from the continuation-based CESK formulation to a compiler that generates single-threaded C code based on co-routines. Instead of maintaining queues, the single-threaded runtime exchanges a pointer between the producer and consumer, avoiding data copies (with one caveat we describe further

in Section 5). Our code generator can make use of first-class labels, available in gcc, clang, and Intel's icc, or compile to a large `switch` statement. Continuations κ in the semantics map directly to labels, although we avoid labels except when necessary for producer-consumer coordination. As a result, we also avoid the overhead inherent in the tick/proc execution model used by Stewart et al. [2015] and instead generate straight-line C for Ziria code that does not perform IO.

In the single-threaded case, when a producer yields a value, the runtime passes a pointer to the yielded value to the downstream consumer, avoiding a data copy. The compiler performs a conservative analysis to determine whether the consumer may use this value after consuming an additional value via a second invocation of **take**, in which case it forces the value to be copied after all. Most code uses a consumed value immediately, so many copies are avoided. Our compiler can also generate multi-threaded code. In this case, data copies are required since the producer may continue running and reuse its internal buffers, possibly overwriting the yielded data before the consumer has a chance to read it. We use standard single-producer/single-consumer fixed-size FIFO queues, allowing for low-overhead thread synchronization and data transfer. The operational semantics guarantees that although a queue may have multiple consumers over the lifetime of a program, at any given time during execution there is only one consumer that could possibly be reading from the queue.

3.5 Language Extensions

The compiler's internal language extends the core language we describe with support for structs and arrays. There are also two additional IO primitives for consuming and producing entire arrays, which are important for pipeline coalescing; we describe them further in Section 4.4. The surface language also supports passing IO computations as arguments to other IO computations. The compiler inlines all IO computations, so these "first class" IO computation arguments are a programmer convenience and do not complicate the core language.

4 HIGH-LEVEL OPTIMIZATIONS

In single-threaded Ziria code, reading from a queue requires saving the consumer's current continuation and jumping to the upstream producer's continuation. When the producer emits an element, it must then jump to the consumer's previously saved continuation. Although the overhead of this co-routine execution model is lower than the overhead of the original tick/proc model for Ziria, eliminating all jumps would be preferable. The goal of the fusion transformation is to eliminate unnecessary synchronization by completely fusing producers and consumers. We illustrate fusion with a simple example:

```
repeat { x ← take; emit f(x) } ⋙ repeat { x ← take; emit g(x) }
```

This computation can be fused into the following par-free from:

```
repeat { x ← take; emit g(f(x)) }
```

This kind of fusion is analogous to map-map fusion, which is well-known in the functional programming literature. The original Ziria compiler goes to great lengths to recognize code in this form and has a special internal representation for it; Stewart et al. term the compiler pass that recognizes this pattern "auto-mapping." The fusion transformation can fuse the example above and many more as well, subsuming auto-mapping.

4.1 The Fusion Transformation

Fusion operates as an abstract interpretation of the operational semantics; in effect, it partially evaluates the producer and consumer in a par at compile time to produce a single, fused, par-free computation. It is convenient to express fusion in a language where expressions, which include

$$
\begin{array}{rcl}
e & ::= & k \\
& | & x \\
& | & \circ_1\ e \\
& | & e\ \circ_2\ e \\
& | & \text{if } e \text{ then } e \text{ else } e \\
& | & \text{let } x = e \text{ in } e \\
& | & \text{letref } x = e \text{ in } e \\
& | & f\ e_1\ \ldots\ e_n \\
& | & !x \\
& | & x := e \\
& | & \text{return } e \\
& | & x \leftarrow e \fatsemi e \\
& | & e \fatsemi e
\end{array}
\qquad
\begin{array}{rcl}
c & ::= & \text{if } e \text{ then } \bar{c} \text{ else } \bar{c} \\
& | & \text{let } x = e \\
& | & \text{letref } x = e \\
& | & \text{return } e \\
& | & \text{lift } e \\
& | & x \leftarrow c \\
& | & \text{take} \\
& | & \text{emit } e \\
& | & \text{repeat } \bar{c} \\
& | & c \ggg c
\end{array}
$$

$$
\begin{array}{rcl}
c' & ::= & c \\
& | & \text{for } i \text{ in } [x, len]\ \bar{c} \\
& | & \text{while } (e)\ \bar{c}
\end{array}
$$

$$
\bar{c} \quad ::= \quad c'_1 \fatsemi c'_2 \fatsemi \ldots
$$

Fig. 6. The computation core language.

reference manipulation but *not* IO, and IO-performing computations are syntactically distinct. In fact, many of the transformations our compiler performs are easier to express as transformations of computational terms modulo the expression language, and our compiler translates the expression core language of Figure 1 into a form that distinguishes the expression and computational terms in an early phase. We show this language, the *computation* core language, in Figure 6. Its operational semantics is completely analogous to the operational semantics given in Section 3; we gave a typing relation and operational semantics in terms of the uniform expression language because the presentation is cleaner. There is one new construct in the language, lift, which lifts a reference-using expression into the computation language. In contrast, return injects a truly pure expression into the computation language. The core computation language also makes the following simplifying assumptions, each of which is also true of the implementation:

(1) All calls to functions that perform IO have been fully inlined.
(2) Sequencing of computations is represented explicitly by the syntactic category $\bar{c}$.
(3) All nested occurrences of par have been eliminated before attempting to fuse two computations. This is accomplished by individually fusing the branches of a par before attempting to fuse the par.
(4) Each computational step has a unique label. We write $[c]^\ell$ for a labeled statement and $\bar{c}^\ell$ for a sequence of statements whose first statement has the label ℓ.
(5) The two terms being fused have disjoint sets of bound variables. This can be enforced by alpha-renaming one of the two terms. The two terms may have free variables in common, but for well-typed terms, none of these shared free variables will be references, a restriction enforced by the context-splitting constraint in the typing relation.

We first give the fusion transformation for computations of the form c and then discuss extending the transformation to terms of the form c', which may include loops. The fusion transformation takes the computations on the left and right-hand sides of a par, each of which consists of a sequence of statements c with unique labels $\ell \in \mathcal{L}$, and produces a fused computation consisting of a sequence of statements labeled by *pairs* of labels $\ell_1 \times \ell_2$. Fusion produces par-free code.

Figure 7 shows the fusion transformation, which operates on sequences of computations. The fusion state $\langle \bar{c}; \bar{c}_l; \bar{c}_r \rangle$ consists of three parts: the currently-fused computation ($\bar{c}$), the remaining

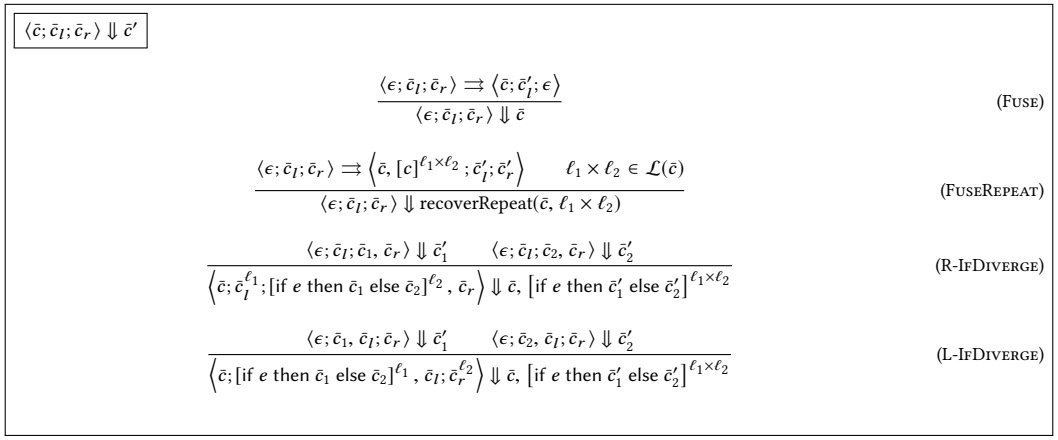


Fig. 7. The fusion transformation.

computation on the left side of the par ($\bar{c}_l$), and the remaining computation on the right side of the par ($\bar{c}_r$). Intuitively, the $\Downarrow$ relation runs the computation to the right of the par using the $\rightrightarrows$ relation given in Figure 8. The right side of the par runs, producing fused steps, until encountering a take. It then runs the left-hand side of the par using the $\leftrightarrows$ relation given in Figure 9 until encountering an emit. When an emit and take match up, they are fused and execution continues. Along the way, a sequence of (fused) statements accumulates.

There are two notable subtleties. First, we wish to fuse repeat statements. To avoid an infinite loop, we test the label of each fused statement; if we have seen its label before, we end fusion and recover the repeat construct. Recovering the repeat construct is simple because we know the label of the first statement in the repeat, and we know the body of the repeat must be a suffix of the accumulated fused statements. This operation is performed by the FuseRepeat rule.

The second subtlety involves if. To avoid code explosion, we attempt to detect when the branches of an if computation *converge*, e.g., when the branches of an if in the consumer both consume the same number of upstream items. When this happens, control flow also converges in the fused computation. If the two branches instead *diverge*, e.g., the branches of an if in the consumer each consume *different* numbers of items, control flow also diverges in the fused computation and code must be duplicated. The 802.11 pipeline does not exhibit this kind of control flow divergence.

4.2 Fusing Non-repeat Loops

Perhaps surprisingly, fusing infinite loops involving repeat does not present a problem. Extending the fusion transformation to support the for and while constructs is more difficult. Ziria offers a for loop that iterates over a given, not necessarily statically known, range. It also provides a while loop with an arbitrary conditional. Once we provide these two constructs, fusion is necessarily partial—there is no way for us in general to know how many times a loop body will be executed. We handle while fusion similarly to if fusion. Consider a consumer while loop; if control flow converges after running the body of the while, which can happen if the producer is a repeat, we can fuse. If control flow diverges, fusion fails. Any for loop with bounds that are not statically known is handled the same way.

Loops with statically known bounds are handled differently. One option is simply to unroll the loop since we know its bounds. This results in tremendous code explosion, and we never unroll loops during fusion. There is a better approach—the intuition being that if we know the

$\boxed{\langle \bar{c}; \bar{c}_l; \bar{c}_r \rangle \rightrightarrows \langle \bar{c}'; \bar{c}'_l; \bar{c}'_r \rangle}$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [\text{let } x = e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [\text{let } x = e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-Let)}$$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [\text{letref } x = e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [\text{letref } x = e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-LetRef)}$$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [\text{return } e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [\text{return } e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-Return)}$$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [x \leftarrow \text{return } e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [x \leftarrow \text{return } e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-BindReturn)}$$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [\text{lift } e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [\text{lift } e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-Lift)}$$

$$\left\langle \bar{c}; \bar{c}_l^{l_1}; [x \leftarrow \text{lift } e]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [x \leftarrow \text{lift } e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-BindLift)}$$

$$\langle \bar{c}; \bar{c}_l; \text{repeat } \bar{c}_2, \bar{c}_r \rangle \rightrightarrows \langle \bar{c}; \bar{c}_l; \bar{c}_2, \text{repeat } \bar{c}_2, \bar{c}_r \rangle \quad \text{(R-UnrollRepeat)}$$

$$\frac{\langle \epsilon; \bar{c}_l; \bar{c}_1 \rangle \rightrightarrows \left\langle \bar{c}'_1; \bar{c}'_l; \epsilon \right\rangle \qquad \langle \epsilon; \bar{c}_l; \bar{c}_2 \rangle \rightrightarrows \left\langle \bar{c}'_2; \bar{c}'_l; \epsilon \right\rangle}{\left\langle \bar{c}; \bar{c}_l^{\ell_1}; [\text{if } e \text{ then } \bar{c}_1 \text{ else } \bar{c}_2]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, \left[\text{if } e \text{ then } \bar{c}'_1 \text{ else } \bar{c}'_2\right]^{\ell_1 \times \ell_2}; \bar{c}'_l; \bar{c}_r \right\rangle} \quad \text{(R-IfConverge)}$$

$$\frac{\langle \bar{c}; \bar{c}_l; c_1, c_2, \bar{c}_r \rangle \rightrightarrows \langle \bar{c}, c_1; \bar{c}_l; c_2, \bar{c}_r \rangle \qquad \langle \bar{c}, c_1; \bar{c}_l; c_2, \bar{c}_r \rangle \rightrightarrows \langle \bar{c}, c_1, c_2; \bar{c}_l; \bar{c}_r \rangle}{\langle \bar{c}; \bar{c}_l; c_1, c_2, \bar{c}_r \rangle \rightrightarrows \langle \bar{c}, c_1, c_2; \bar{c}_l; \bar{c}_r \rangle} \quad \text{(R-Seq)}$$

$$\left\langle \bar{c}; [\text{emit } e]^{\ell_1}, \bar{c}_l; [x \leftarrow \text{take}]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \left\langle \bar{c}, [\text{let } x = e]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}_r \right\rangle \quad \text{(R-TakeEmit)}$$

$$\begin{array}{l} \left\langle \bar{c}; [x \leftarrow \text{take}]^{\ell_1}, \bar{c}_l; [y \leftarrow \text{take}]^{\ell_2}, \bar{c}_r \right\rangle \rightrightarrows \\ \left\langle \bar{c}, [x \leftarrow \text{take}]^{\ell_1 \times \ell_2}; \bar{c}_l; [y \leftarrow \text{take}]^{\ell_2}, \bar{c}_r \right\rangle \end{array} \quad \text{(R-TakeLeft)}$$

$$\frac{\langle \bar{c}; c, \bar{c}_l; x \leftarrow \text{take}, \bar{c}_r \rangle \leftrightarrows \langle \bar{c}'; \bar{c}_l; x \leftarrow \text{take}, \bar{c}_r \rangle}{\langle \bar{c}; c, \bar{c}_l; x \leftarrow \text{take}, \bar{c}_r \rangle \rightrightarrows \langle \bar{c}'; \bar{c}_l; x \leftarrow \text{take}, \bar{c}_r \rangle} \quad \text{(R-StepLeft)}$$

Fig. 8. Fusing the right-hand side of a par.

input/output behavior of the body of a loop, we can potentially split loops so that, for example, the bodies of a producer loop and consumer loop can then be merged. Most loops in the 802.11 pipeline have fixed bounds, so this strategy is very successful. However, before we can fuse these forms of loop, we need to determine their input/output behavior.

4.3 Rate Analysis

Rate analysis provides an approximation of the input/output behavior of a computation. A rate is a pair of multiplicities, one for the computation's input, and one for its output. We write computer rates as $[m, n]$, where m and n are multiplicities, and we write transformer rates as $[m, n]^*$. Multiplicities may be:

(1) A fixed constant, which could be zero, written n.
(2) Zero or more blocks of size $n > 1$, written n^*.
(3) One or more blocks of size $n > 1$, written n^+.

For a computer, a multiplicity describes the read/write behavior of the computer over its entire execution. For a transformer, a multiplicity describes the read/write behavior of the transformer for a single round of execution.

$\langle \bar{c}; \bar{c}_l; \bar{c}_r \rangle \leftrightarrows \langle \bar{c}'; \bar{c}'_l; \bar{c}'_r \rangle$

$$\left\langle \bar{c}; [\text{let } x = e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [\text{let } x = e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-Let)}$$

$$\left\langle \bar{c}; [\text{letref } x = e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [\text{letref } x = e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-LetRef)}$$

$$\left\langle \bar{c}; [\text{return } e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [\text{return } e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-Return)}$$

$$\left\langle \bar{c}; [x \leftarrow \text{return } e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [x \leftarrow \text{return } e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-BindReturn)}$$

$$\left\langle \bar{c}; [\text{lift } e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [\text{lift } e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-Lift)}$$

$$\left\langle \bar{c}; [x \leftarrow \text{lift } e]^{\ell_1}, \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \leftrightarrows \left\langle \bar{c}, [x \leftarrow \text{lift } e]^{\ell_1 \times \ell_2}; \bar{c}_l; [c_r]^{l_2}, \bar{c}_r \right\rangle \quad \text{(L-BindLift)}$$

$$\langle \bar{c}; \text{repeat } \bar{c}_1, \bar{c}_l; \bar{c}_r \rangle \leftrightarrows \langle \bar{c}; \bar{c}_1, \text{repeat } \bar{c}_1, \bar{c}_l; \bar{c}_r \rangle \quad \text{(L-UnrollRepeat)}$$

$$\frac{\langle \epsilon; \bar{c}_1; \bar{c}_r \rangle \leftrightarrows \langle \bar{c}'_1; \epsilon; \bar{c}'_r \rangle \qquad \langle \epsilon; \bar{c}_2; \bar{c}_r \rangle \leftrightarrows \langle \bar{c}'_2; \epsilon; \bar{c}'_r \rangle}{\left\langle \bar{c}; [\text{if } e \text{ then } \bar{c}_1 \text{ else } \bar{c}_2]^{\ell_1}, \bar{c}_l; \bar{c}_r^{\ell_2} \right\rangle \leftrightarrows \left\langle \bar{c}, \left[\text{if } e \text{ then } \bar{c}'_1 \text{ else } \bar{c}'_2\right]^{\ell_1 \times \ell_2}; \bar{c}_l; \bar{c}'_r \right\rangle} \quad \text{(L-IfConverge)}$$

Fig. 9. Fusing the left-hand side of a par.

A rate of n means that we know the computation will read/write exactly n elements. A rate of n^* means that *if* the computation reads/writes, it will read/write some multiple of n elements. Note that 1^* means that *if* the computation reads/writes, it will read/write some multiple of 1 elements, i.e., it is equivalent to saying we don't know anything about the computation's read/write behavior. A rate of n^+ means that the computation will read/write some multiple of n elements, i.e., it is a guarantee that some positive multiple of n elements will be read/written. Having both n^* and n^+ is important, as it lets us infer a useful rate for examples like the following:

```
{
  xs ← take 16;
  emit e1;
  while ( some condition ) {
    ys ← take 16;
    emit e2
  }
}
```

The first part of the computation has rate $[16, 1]$, and the second part of the computation has a rate $[16^*, 1^*]$, so the overall rate is $[16^+, 1^+]$.

Rate analysis is fairly straightforward, so we do not give details here. The only complication is that we sometimes need to take the gcd when combining, e.g., multiplicities i and j^+ to get the multiplicity $\gcd(i, j)^+$. The rules for inferring rates for a computer in sequence with a transformer are slightly less obvious because we must ensure that the computed rate is valid for *all* transformer rounds. Our analysis is compositional.

4.4 Pipeline Coalescing

Rate analysis gives the fusion transformation enough information to fuse for loops with statically-known bounds without unrolling them. Being able to aggressively fuse loops enables an additional optimization, *pipeline coalescing*. Pipeline coalescing attempts to do two things:

(1) Coalesce computation input/output into single blocks so that individual take and emit operations spread throughout a computation are converted into a single, block-sized IO operation. This opens up many new opportunities for the LUT transformation, which can only handle expressions that do not perform IO.
(2) Process data in larger chunks. This effectively involves running the body of the computation multiple times, a transformation that is only valid for transformers, not computers.

Pipeline coalescing makes use of arrays and array-based IO operations, which are present in the implementation but not in our formalization in Section 3. Adding them is straightforward—they would only clutter the semantics without demonstrating anything fundamentally new. These array language constructs are:

- Array types, $\texttt{arr}[n]\tau$, in the τ syntactic category (base types).
- **takes** n, which consumes n elements from the input stream of type α and returns a value of type $\texttt{arr}[n]\alpha$.
- **emits** e, where e has type $\texttt{arr}[n]\beta$, emits all n elements of its argument to the output stream of type β.

Coalescing relies on inserting variants of two transformers, the *left coalescing transformer* $co_{\mathrm{L}}{}^{n}_{\tau}$, and the *right coalescing transformer*, $co_{\mathrm{R}}{}^{n}_{\tau}$, into the pipeline to the left and right (resp.) of the computation being transformed. Fusion then optimizes the transformed pipeline, eliminating the coalescing transformers and fusing the computations whose communication they mediate.

The left coalescing transformer, $co_{\mathrm{L}}{}^{n}_{\tau}$, consumes array-sized chunks of input and parcels them out element-by-element. Its definition is:

```
repeat {
  xs ← takes n;
  emit xs[0];
  ...;
  emit xs[n-1];
}
```

The right coalescing transformer, $co_{\mathrm{R}}{}^{n}_{\tau}$, consumes input one element at a time and outputs array-size chunks. Its definition is:

```
var xs : arr[n]τ
repeat {
  x ← take; xs[0] := x;
  ...
  x ← take; xs[n-1] := x;
  emits xs
}
```

The goal behind pipeline coalescing is to line up producers and consumers so that they emit and take blocks of the same size. The only difficulty is in choosing which coalescing transformers to insert; once we have made this choice, we rely on fusion to fuse them away. Fortunately, we can use rate analysis to guide our choice. We write $i \rightarrow j$ for a computer *or* transformer that takes input in chunks of size i and produce outputs in chunks of size j. A computation whose shape is $i \rightarrow j$ can always be safely coalesced so that it consumes entire blocks of size i and produces entire blocks of size j. We might like to widen the data path by taking some multiple of i

elements each round, but this is not in general safe to do for the very reasons outlined in Section 2.2. We avoid over-consumption by ensuring that the block size is less than the multiplicity of the constraining computation. For example, consider a transformer whose downstream computer has input multiplicity m or m^+. In this case, we can only coalesce up to blocks of size m, since if we produce larger blocks, the consumer may terminate before having consumed all the data we have produced.

5 LOW-LEVEL OPTIMIZATIONS

To a first approximation, the speed of the 802.11 pipeline is inversely proportional to the number of memory copies performed. The primary purpose of the low-level optimizations we perform is simply-stated: avoid memory copies. By choosing a pure core language where dereferencing and assignment are explicit, monadic operations, it may seem as though we have shot ourselves in the foot. After all, consider the following code:

```
{ x ← !foo; compute something with x }
```

Surely this must result in an explicit data copy, since there is no other way to guarantee that `x` is immutable. In fact, this is not the case—a data flow analysis can determine when a read-after-modify occurs for variables to which `foo` flows; only these variables require a copy. This same data flow analysis has an added benefit.

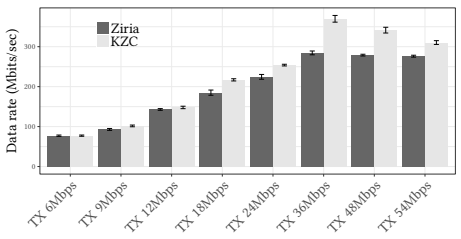
As described in Section 3.4, the single-threaded runtime exchanges a pointer between the producer and consumer. If the consumer consumes a value and that value is used *after* a second value is consumed, then we must perform a data copy first to save the original value. The safe solution, which the original Ziria compiler implemented, is to always copy data read from a queue. However, this defeats the purpose of "zero-copy" pointer-exchanging queues. By instead treating queues as another "reference" and treating take as a modification of the queue, our data flow analysis tells us when it is necessary to copy the data from the pointer yielded to the consumer. Intuitively, a copy is only required when a consumed value is used after a subsequent take, and this is exactly what the data flow analysis tells us. If consumed values are always used before a subsequent take, the data flow analysis allows us to eliminate all data copies related to queue consumption.

The semantics of Ziria state that values are initialized to a default value. This initialization can be quite expensive for large arrays. We perform an additional analysis that can eliminate most unnecessary initialization. Combined with the already mentioned data flow analysis and the continuation-based code generator, this results in generated C code with very few unnecessary memory operations, which is a significant source of the improvements we report in the following section.

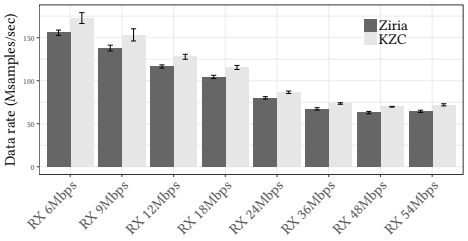
6 EVALUATION

The primary contribution of this paper is a formalization of Ziria and generalizations of the transformations that enable the compilation of high-level Ziria programs into efficient low-level code. Nonetheless, in this section we benchmark our compiler, kzc[3], against the original Ziria compiler, wplc[4]. 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[5], all runs were repeated 100 times, and we assume runtimes are normally distributed. All Ziria programs evaluated in this section are taken from the publicly available Ziria release [Stewart et al. 2016]. Although both kzc and wplc can generate multi-threaded code, wplc's multithreading support is limited to Windows, so all

[3]The **K**yllini **Z**iria **c**ompiler—Kyllini is another name for the mountain Ziria.
[4]The **w**ireless **p**rogramming **l**anguage **c**ompiler
[5]`-march=native -mtune=native -Ofast`.

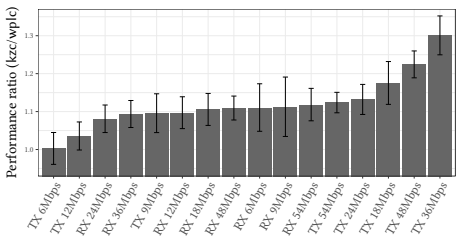


(a) Transmitter Performance

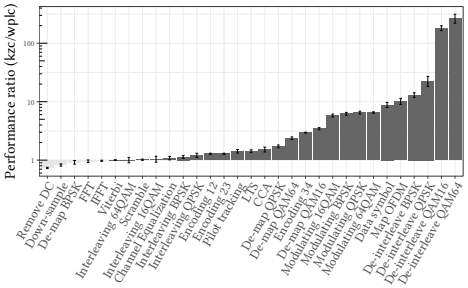


(b) Receiver Performance

Fig. 10. Transmitter and receiver data rates. The receiver consumes a quadrature signal consisting of pairs of 16-bit numbers representing IQ samples. The transmitter consumes bits. Error bars show one standard deviation above and below the mean.



(a) Performance improvement of transmitter/receiver pipelines at different 802.11 data rates



(b) Performance improvement of individual blocks

Fig. 11. Performance improvement ratios. These figures show the relative improvement of kzc over wplc, both for entire transmitter/receiver pipelines and for individual blocks. The vertical axis gives the ratio of the throughput of the kzc-compiled version to the throughput of the wplc-compiled version. Error bars show the bound of the ratio when the two metrics being compared range from one standard deviation below the mean to one standard deviation above the mean. Note that Figure 11a uses a linear scale, whereas Figure 11b uses a log scale.

benchmarks are single-threaded. This is also appropriate since we are benchmarking single-threaded optimizations like fusion, not the runtime's queue implementation.

Figures 10a and 10b compare the transmitter and receiver performance, respectively, of the two compilers. The ratios of the data rates of the kzc and wplc-compiled implementations of these same transmit and receive pipelines are shown in Figure 11a. Code compiled by kzc is always as fast as code compiled by wplc, and in most cases, it is at least 10% faster. The relative performance of individual pipeline blocks is broken out in Figure 11b.

In our experience, once high-level optimization passes like fusion have been performed, there are two factors that explain most of the performance variation: vectorization and extraneous memory traffic. Extra memory traffic simply kills performance; the performance benefits of using kzc over wplc result almost entirely from the low-level optimizations described in Section 5, which are designed to eliminate extra memory traffic. For blocks in Figure 11b with improvements on the order of 10–100×, such as the de-interleaving blocks, kzc produces code that performs a single LUT lookup for each input and outputs the LUT entry directly, whereas wplc produces code than performs one or more memory copies. For the two blocks where kzc performs measurably worse

than wplc, "Remove DC" and "Down-sample," wplc generates code that buffers the block's output whereas kzc does not. Without fusion, performance can be orders of magnitude worse than the performance shown; fusion is a critical transformation. Many of the low-level optimizations kzc performs are baked in to the structure of the compiler, so we unfortunately have no way to easily disable them and cannot measure the performance of fusion alone.

We use the same runtime primitives as wplc, many of which have been hand-vectorized, so vectorization does not contribute to the performance differences in Figure 11b (kzc does not produce code that is more easily vectorized by gcc than that produced by wplc.) In fact, we were frankly surprised that we could attain even a 10% improvement for whole pipelines in many cases, as the bottleneck in the transmitter and receiver pipelines tend to be primitive blocks like FFT, IFFT, and Viterbi. As one would expect, for these blocks kzc and wplc generate code that performs identically.

One avenue for future performance gains is to write these primitive blocks in Ziria, thereby opening them up to cross-block optimizations like fusion. We have written implementations of these blocks in Ziria. The Ziria versions of FFT and Viterbi do not perform quite as well as the handwritten primitives in the wplc runtime when run as individual blocks, but when they are part of a longer pipeline, fusion and inlining result in code that is faster overall. We plan to fully exploit this synergy in the future. For now, we are pleased that engineering a compiler around a well-defined core language and semantics wrung a 10% performance increase over an already highly-optimizing compiler.

7 RELATED WORK

SDR. Our work is inspired by and based on the original Ziria system [Stewart et al. 2015]. We benchmark our system against the original Ziria compiler using its accompanying WiFi implementation and much of its standard library routines, such as FFT, IFFT, and Viterbi. However, our compiler does not share any code with wplc. Mainland [2017] describes how the impure Ziria surface language can be elaborated to a pure, monadic core language. We use those techniques to elaborate to our core language. Our primary contributions relative to these two works are a formalized core language and operational semantics, the fusion and pipeline coalescing transformations, and a full implementation of both in the kzc compiler.

SDR platforms are usually based on an FPGA. Murphy et al. [2006] and Chacko et al. [2014] use MATLAB's Simulink tool to design 802.11 implementations that run directly on an FPGA. Ng et al. [2010] instead use Bluespec [Nikhil 2008] to write an FPGA 802.11 implementation. More typically, the FPGA is used as a fixed-function block, and the "software" part of the SDR platform runs on a commodity CPU [Balan et al. 2013; Ettus Research 2017; Tan et al. 2009]. The most common tool for programming these sorts of SDR platforms is the GNU Radio environment [Blossom 2004], although there are numerous other approaches to programming SDR applications [Bansal et al. 2012; Dutta et al. 2013; Rondeau et al. 2013; Sora Core Team and Tan 2012; Voronenko et al. 2010].

Most of these approaches, including GNU Radio, use graph-based programming models: vertices represent computation, and edges represent communication. These models do not make a clean distinction between control-flow and data-flow, often piggy-backing control messages on data channels or communicating via global state. They also leave vague the details of state initialization for a vertex. Furthermore, vertices are black boxes, leaving no opportunity for cross-vertex optimizations (like fusion).

None of the available SDR platforms simultaneously provide both the performance and powerful abstractions needed for the SDR domain. Instead, one must choose either programmer convenience or performance. Performance also typically requires manual optimization and specialization to, for example, the bit-width of a particular pipeline. Ziria allows much of this previously manual work to be performed automatically by a compiler. The optimizations we describe in this paper

are a large step towards a language that can provide programmer convenience without sacrificing performance.

Signal Processing. FFTW [Frigo 1999] is a specialized framework for generating only FFT implementations. The original motivation behind SPIRAL [Püschel et al. 2005] was also to generate FFTs [Franchetti et al. 2009; Xiong et al. 2001], although it is based on a more general computational framework [Van Loan 1992] for expressing signal transforms. SPIRAL searches over a space of implementations, which can be time consuming, using algebraic rules that specify how to decompose signal transforms into smaller transforms. Our compiler is more traditional; it does not perform search, but instead uses static analyses to guide the optimizer. de Mesmay [2010] uses SPIRAL to implement the ACS butterfly component of the Viterbi algorithm, which is similar to the FFT butterfly already supported by the framework. They rely on an existing Viterbi implementation for the traceback stage. Viterbi has been fully implemented in Ziria. We are working on incorporating SPIRAL-like techniques in our compiler.

Dataflow Languages. Synchronous dataflow languages such as Esterel [Berry and Gonthier 1992], LUSTRE [Halbwachs et al. 1991], SIGNAL [Gautier et al. 1987], and Lucid Synchrone [Caspi and Pouzet 1995] have been used to both verify and implement low-level reactive systems. Unlike these languages, Ziria allows for the possibility of asynchronous communication and provides no guarantees about timing. Ziria also cannot guarantee bounded queue sizes—in the serial execution model, the consequence is not unbounded memory usage, but that one member of the consumer/producer pair must execute multiple iterations for each iteration of the other member of the pair. However, our rate analysis can take advantage of code for which queue sizes *can* be bounded, and the compiler can warn the user when it cannot find such a bound.

Lucid [Ashcroft and Wadge 1977; Wadge and Ashcroft 1985] is an untyped dataflow language that has a demand-driven execution model, somewhat reminiscent of the semantics given in Section 3. However, Lucid programs may require values computed at an arbitrary time in the past, leading to potentially arbitrarily large memory requirements. StreamIt [Thies et al. 2002] is a more modern dataflow language. It has been used to implement an 802.11 PHY [Thies and Amarasinghe 2010], although the performance characteristics of this implementation are not provided. StreamIt filters must declare their data rates, whereas the Ziria compiler infers data rates using rate inference. Whereas Ziria separates control flow and dataflow, StreamIt uses teleport messages [Thies et al. 2005] to communicate control decision between different components of a pipeline, which makes control flow difficult to reason about. Our fusion transformation does not have an analog in StreamIt; this is not surprising since one of the original goals of StreamIt was to target the streaming RAW processor architecture [Taylor et al. 2002]. We expect that many transformations implemented by the StreamIt compiler, such as optimization of linear filters [Dubé and Feeley 2005], could be added to kzc.

Fusion. There is a large body of work on fusion in the functional programming community. Wadler [1990] introduced the problem of deforestation, that is, of eliminating intermediate structures in programs written as compositions of list transforming functions. Follow-on work [Gill et al. 1993; Hamilton 2001; Johann 2001; Marlow and Wadler 1993; Svenningsson 2002; Takano and Meijer 1995] attempted to improve the ability of compilers to automate deforestation through program transformations. Our fusion transformation is more closely related to Coutts et al. [2007], which subsumes much of the earlier work and accomplishes fusion by rewriting standard functional operations in terms of stepwise computations similar to take and emit. Mainland et al. [2013] build on this work and show how to produce fused code that leverages vector instructions.

The fusion framework of Coutts et al. [2007] deals with producer/consumer pairs that operate in lockstep on single elements. Mainland et al. [2013] extend this to producer/consumer pairs that can handle multiple elements at a time, but they still operate in lockstep; for example, a producer can yield a 4-vector, but the consumer must consume 4-vectors. Our fusion transformation, combined with rate analysis and pipeline coalescing, removes this restriction—producers and consumers will be "rate matched" by our transformations. Our fusion transformation also handles imperative loop constructs intelligently, avoiding code blow-up. Writing an equivalent recursive function in Haskell can frustrate fusion, leading to either code blow-up or silent fusion failure. In general, fusion fails silently in Haskell code. Our compiler allows the user to specify a bound on code size blow-up resulting from fusion, and it can also warn the user when fusion fails.

Functional Programming. Ziria's sequence and bind constructs are analogous to the sequence and bind construct of monads [Peyton Jones and Wadler 1993], whereas its par operator is much like the arrow operator (≫) [Hughes 2000]. Similar constructs appear throughout the functional programming literature, especially in the context of Functional Reactive Programming (FRP). FRP's switch combinator [Courtney and Elliott 2001; Wan and Hudak 2000] is related to Ziria's sequence, and Fudgets' stream processors [Carlsson and Hallgren 1998] are similar to the ST construct in Ziria.

8 CONCLUSIONS AND FUTURE WORK

We have presented a core language and operational semantics for Ziria. Both serve as the basis for several optimizations, like fusion and pipeline coalescing, that are applicable to other domains beyond software defined radio—we expect that, with a few extensions, Ziria will be useful as a language for describing video codecs as well as many computer vision algorithms. These two transformations are generalizations of optimizations performed by the original Ziria compiler; by presenting them formally, we hope that they find broader use. Using the core language and operational semantics as the basis for a new compiler for the Ziria surface language also resulted in insights that yielded a significant increase in the performance of compiled Ziria code.

Future work. We have already extended Ziria with full support for polymorphism over base types. We have had no need for higher-order functions; higher-order *computations*, on the other hand, are very useful and were part of the original Ziria language. Our compiler monomorphizes polymorphic programs. We have also added a restricted form of dependent types by introducing type-level natural numbers and Nat-kinded type variables. This allows Ziria functions to be length-polymorphic. Although the original Ziria compiler supported a limited form of "array length polymorphism," this support was bolted on to the implementation and not cleanly captured in the type system.

In addition to polymorphism, we are adding fixed point types and a restricted form of type classes [Wadler and Blott 1989] to Ziria. The primary motivation is to allow polymorphism over fractional types so that code that uses floating point computation does not have to be rewritten to use fixed point computation. The conversion from floating point to fixed point is important when targeting hardware or digital signal processors, such as the TI C66x series. Being able to write code once that can then be instantiated to either floating point or fixed point types is the first step to automating the process of choosing an appropriate fixed point precision and range.

We are also working on a VHDL back end for Ziria. Our medium-term goal is to fully re-implement SOFDM [Chacko et al. 2014], which is written in MATLAB's Simulink, in Ziria. Simultaneously, we hope to port Ziria to the TI C66x DSP family. We hope these experiences will make Ziria a viable language for hardware development as well as for CPU and DSP development. We also plan to broaden the applicability Ziria by attacking domains such as wireless MAC protocols, video

codecs, and computer vision algorithms. Finally, we plan to further formalize our semantics and transformations and produce machine-checked proofs of their correctness.

REFERENCES

E. A. Ashcroft and W. W. Wadge. 1977. Lucid, a Nonprocedural Language with Iteration. *Commun. ACM* 20, 7 (July 1977), 519–526. https://doi.org/10.1145/359636.359715

Horia Vlad Balan, Marcelo Segura, Suvil Deora, Antonios Michaloliakos, Ryan Rogalin, Konstantinos Psounis, and Giuseppe Caire. 2013. USC SDR, an Easy-to-Program, High Data Rate, Real Time Software Radio Platform. In *Proceedings of the Second Workshop on Software Radio Implementation Forum (SRIF '13)*. Hong Kong, 25–30. https://doi.org/10.1145/2491246.2491254

Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. 2012. OpenRadio: A Programmable Wireless Dataplane. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. Helsinki, Finland, 109–114. https://doi.org/10.1145/2342441.2342464

Gerard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

Eric Blossom. 2004. GNU Radio: Tools for Exploring the Radio Frequency Spectrum. *Linux Journal* 2004, 122 (June 2004), 4. https://www.linuxjournal.com/article/7319

Magnus Carlsson and Thomas Hallgren. 1998. *Fudgets - Purely Functional Processes with Applications to Graphical User Interfaces*. Ph.D. Dissertation. Chalmers University of Technology, Göteborg, Sweden.

Paul Caspi and Marc Pouzet. 1995. A Functional Extension to Lustre. In *Eighth International Symposium on Languages for Intentional Programming (ISLIP '95)*, Mehmet A. Orgun and Edward A. Ashcroft (Eds.). World Scientific, Sydney, Australia.

James Chacko, C. Sahin, Danh Nguyen, Doug Pfeil, Nagarajan Kandasamy, and Kapil Dandekar. 2014. FPGA-Based Latency-Insensitive OFDM Pipeline for Wireless Research. In *Proceedings of the 2014 IEEE Conference on High Performance Extreme Computing Conference (HPEC '14)*. Waltham, MA, 1–6. https://doi.org/10.1109/HPEC.2014.7040969

Antony Courtney and Conal Elliott. 2001. Genuinely Functional User Interfaces. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW '01)*. Firenze, Italy.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Freiburg, Germany, 315–326. https://doi.org/10.1145/1291151.1291199

Frédéric de Mesmay. 2010. *On the Computer Generation of Adaptive Numerical Libraries*. Ph.D. Dissertation. Electrical and Computer Engineering, Carnegie Mellon University.

Danny Dubé and Marc Feeley. 2005. BIT: A Very Compact Scheme System for Microcontrollers. *Higher-Order and Symbolic Computation* 18, 3 (Dec. 2005), 271–298. https://doi.org/10.1007/s10990-005-4877-4

Aveek Dutta, Dola Saha, Dirk Grunwald, and Douglas Sicker. 2013. CODIPHY: Composing on-Demand Intelligent Physical Layers. In *Proceedings of the Second Workshop on Software Radio Implementation Forum (SRIF '13)*. Hong Kong, 1–8. https://doi.org/10.1145/2491246.2491247

Ettus Research. 2017. Ettus Research, USRP: Universal Software Radio Peripheral. (Feb. 2017). http://www.ettus.com/

Matthias Felleisen. 1987. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph.D. Dissertation. Indiana University, Indianapolis, IN.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. The MIT Press, Cambridge, MA.

Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. 2009. Discrete Fourier Transform on Multicore. *IEEE Signal Processing Magazine* 26, 6 (Nov. 2009), 90–102. https://doi.org/10.1109/MSP.2009.934155

Matteo Frigo. 1999. A Fast Fourier Transform Compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*. Atlanta, Georgia, 169–180. https://doi.org/10.1145/301631.301661

Thierry Gautier, Paul Le Guernic, and Loïc Besnard. 1987. SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA '87)*, Gilles Kahn (Ed.). Number 274 in Lecture Notes in Computer Science. Springer, Portland, Oregon, 257–277. https://doi.org/10.1007/3-540-18317-5_15

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, Copenhagen, Denmark, 223–232. https://doi.org/10.1145/165180.165214

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language Lustre. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320. https://doi.org/10.1109/5.97300

G. W. Hamilton. 2001. Extending Higher-Order Deforestation: Transforming Programs to Eliminate Even More Trees. In *Selected Papers from the 3rd Scottish Functional Programming Workshop (SFP '01)*, Kevin Hammond and Sharon Curtis (Eds.). Intellect Books, Exeter, UK, 25–36. http://dl.acm.org/citation.cfm?id=644403.644407

John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37, 1–3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

IEEE. 2012. IEEE Standard for Information Technology—Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks—Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)* (March 2012). https://doi.org/10.1109/IEEESTD.2012.6178212

Patricia Johann. 2001. Short Cut Fusion: Proved and Improved. In *Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '01)*. Lecture Notes in Computer Science, Vol. 2196. Springer-Verlag, Florence, Italy, 47–71. https://doi.org/10.1007/3-540-44806-3_4

Geoffrey Mainland. 2017. A Domain-Specific Language for Software-Defined Radio. In *Proceedings of the 19th International Symposium on Practical Aspects of Declarative Languages (PADL '17)*. Lecture Notes in Computer Science, Vol. 10137. Springer International Publishing, Paris, France, 173–188. https://doi.org/10.1007/978-3-319-51676-9_12

Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2013. Exploiting Vector Instructions with Generalized Stream Fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Boston, MA, 37–48. https://doi.org/10.1145/2500365.2500601

Simon Marlow and Philip Wadler. 1993. Deforestation for Higher-Order Functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming (Workshops in Computing)*. Springer-Verlag, Ayr, Scotland, 154–165. https://doi.org/10.1007/978-1-4471-3215-8_14

Patrick Murphy, Ashnu Sabharwal, and Behnaam Aazhang. 2006. Design of WARP: A Flexible Wireless Open-Access Research Platform. In *Proceedings of the 14th European Signal Processing Conference (EUSIPCO '06)*. Florence, Italy.

Man Cheuk Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. 2010. 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)*. La Jolla, CA, 4:1–4:11. https://doi.org/10.1145/1872007.1872013

Rishiyur S. Nikhil. 2008. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In *High-Level Synthesis*, Philippe Coussy and Adam Morawiec (Eds.). Springer Netherlands, 129–146. https://doi.org/10.1007/978-1-4020-8588-8_8

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, Charleston, South Carolina, 71–84. https://doi.org/10.1145/158511.158524 ACM ID: 158524.

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. 2005. Spiral: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb. 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

T. Rondeau, Nicholas McCarthy, and Timothy O'Shea. 2013. SIMD Programming in GNURadio: Maintainable and User-Friendly Algorithm Optimization with VOLK. In *Proceedings of SDR-WInnComm 2013*. Washington, D.C.

Sora Core Team and Kun Tan. 2012. *Brick Specification*. Technical Report MSR-TR-2012-26. Microsoft Research.

Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agulló. 2015. Ziria: An Optimizing Compiler for Wireless PHY Programming. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Istanbul, Turkey. https://doi.org/10.1145/2694344.2694368

Gordon Stewart, Dimitrios Vytiniotis, Geoffrey Mainland, Bozidar Radunovic, and Edsko de Vries. 2016. Ziria. (April 2016). https://github.com/dimitriv/Ziria Version 85cc34db.

Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, Pittsburgh, PA, 124–132. https://doi.org/10.1145/581478.581491

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the Seventh International Conference on Functional Programming and Computer Architecture (FPCA '95)*. La Jolla, CA, 306–313. https://doi.org/10.1145/224164.224221

Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. 2009. Sora: High Performance Software Radio Using General Purpose Multi-Core Processors. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. Boston, MA, 75–90.

Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (March 2002), 25–35. https://doi.org/10.1109/MM.2002.997877

William Thies and Saman Amarasinghe. 2010. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, Vienna, Austria, 365–376. https://doi.org/10.1145/1854273.1854319

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Grenoble, France, 179–196. https://doi.org/10.1007/3-540-45937-5_14

William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. 2005. Teleport Messaging for Distributed Stream Programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. Chicago, IL, 224–235. https://doi.org/10.1145/1065944.1065975

Charles F. Van Loan. 1992. *Computational frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Yevgen Voronenko, Volodymyr Arbatov, Christian Berger, Ronghui Peng, Markus Püschel, and Franz Franchetti. 2010. Computer Generation of Platform-Adapted Physical Layer Software. In *Proceedings of Software Defined Radio (SDR '10)*.

William W. Wadge and Edward A. Ashcroft. 1985. *Lucid, the Dataflow Programming Language*. Number 22 in A.P.I.C. Studies in Data Processing. Academic Press, London.

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73, 2 (June 1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Philip Wadler and Stephen Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. Austin, Texas, 60–76. https://doi.org/10.1145/75277.75283

Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. Vancouver, British Columbia, Canada, 242–252. https://doi.org/10.1145/349299.349331

Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. 2001. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. Snowbird, Utah, 298–308. https://doi.org/10.1145/378795.378860