# Language Design for Distributed Stream Processing

by

Ryan Rhodes Newton

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
Jan 30, 2009

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Samuel Madden
Associate Professor
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arvind
Johnson Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Professor Terry P. Orlando
Chair, Department Committee on Graduate Students

# Language Design for Distributed Stream Processing

by

Ryan Rhodes Newton

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 30, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

Applications that combine live data streams with embedded, parallel, and distributed processing are becoming more commonplace. *WaveScript* is a domain-specific language that brings high-level, type-safe, garbage-collected programming to these domains. This is made possible by three primary implementation techniques, each of which leverages characteristics of the streaming domain. First, WaveScript employs an evaluation strategy that uses a combination of interpretation and reification to partially evaluate programs into stream dataflow graphs. Second, we use profile-driven compilation to enable many optimizations that are normally only available in the synchronous (rather than asynchronous) dataflow domain. Finally, an empirical, profile-driven approach also allows us to compute practical partitions of dataflow graphs, spreading them across embedded nodes and more powerful servers.

We have used our language to build and deploy applications, including a sensor-network for the acoustic localization of wild animals such as the Yellow-Bellied marmot. We evaluate WaveScript's performance on this application, showing that it yields good performance on both embedded and desktop-class machines. Our language allowed us to implement the application rapidly, while outperforming a previous C implementation by over 35%, using fewer than half the lines of code. We evaluate the contribution of our optimizations to this success. We also evaluate WaveScript's ability to extract parallelism from this and other applications.

Thesis Supervisor: Samuel Madden
Title: Associate Professor

Thesis Supervisor: Arvind
Title: Johnson Professor

# Acknowledgments

Acknowledgements are a place to recognize that which we take for granted. I confess that I always assumed I would go to graduate school and since entering graduate school the whole process has grown normal and its endpoint expected. But I'd like to recognize that it could have been otherwise—without supportive friends, family, and teachers things would have certainly turned out differently.

My parents could not be more supportive of my education. My wife, Irene Garcia Newton, is always there for me in more ways than I can count, including reading drafts and picking up the slack when I have a deadline. I am grateful to the IMACS program for setting me on my path early in life and to Dan Friedman for reaching out to me while I was still in high school. My undergraduate mentors, Dan Friedman, Kent Dybvig, and Amr Sabry, are responsible for stoking my academic interests and sending me to where I am now.

I am lucky to have a great advisor, Samuel Madden—always available, responsive, helpful in all the ways an advisor should be. I have also been fortunate to interact with many faculty members while at MIT. Chronologically, Gerald Sussman and Jonathan Bachrach got me interested in amorphous computing. Later Norman Ramsey introduced me to Matt Welsh who brought me into the sensor networking field, helping me get my sea legs as a researcher and publish my first paper. For that, I am indebted to him. Greg Morrisett has consistently made available to me his expertise in programming languages and compilers. And if it weren't for him, I may have had no one with whom to discuss compiler internals (and would surely have gone crazy).

Professor Arvind has been a source of insight throughout my graduate career. He is a great sage of parallelism, always helpful in cutting away surface details and getting to the heart of things. Hari Balakrishnan has provided enormous help both in the actual process of paper writing and in playing a great skeptic when it comes to the benefits I might claim of my systems.

# Contents

# List of Figures

# Chapter 1

# Introduction

The goal of this thesis is to provide an effective method for compiling an important class of future programming languages: those specialized to embedded and distributed processing of data streams. These programming languages will be driven to prominence by three intersecting trends. First, as embedded computing grows ever more ubiquitous, each person and every object of value will gain the capacity for computation and communication. Second, parallel and distributed computing is moving from niche applications into the mainstream of software development, forced there by stagnation in uniprocessor (single core) performance. Third, high-volume data streams are becoming more numerous and more widely available: stock ticks, weather information, audio and video feeds.

These developments pose serious difficulties for programmers both alone and in combination. Embedded programs are frequently more troublesome to debug than their desktop counterparts. For example, tight resource budgets encourage optimization at the expense of abstraction and modularity. Similarly, parallel programming has long been recognized as treacherous, given the potential for non-determinism and unpredictable interactions between stateful components. Finally, high-rate data streams require carefully designed programs that optimize communications and minimize per-element overheads.

Many applications touch all three of these trends—for example, sensor network applications processing data from acoustic, video, or vibrational sensors. In Chapter 3.1, we will review representative applications that have been undertaken during the course of this thesis, spanning several sensory modalities and various sensor network hardware platforms.

These include: audio processing applications (animal localization, speaker classification), computer vision, and processing of EEG signals and vibrational data (detecting pipeline leaks).

The central claim of this thesis is that the above programming problems have a single, simple solution: namely, that a sufficiently powerful stream programming language with the right kind of compiler can replace embedded programming tools, parallel programming with threads and locks, streaming databases, and DSP systems based on synchronous dataflow. The *WaveScript* programming language and attendant compiler provides such a system.

WaveScript is based on two fundamental design decisions. First, the WaveScript compiler manipulates and produces a complete and fixed *graph* of dataflow operators connected by *streams*. This assumption is typical of stream processing systems—and a surprisingly broad range of applications can meet this constraint—but it should still be acknowledged as substantially restricting the design space. For example, this decision excludes dynamically changing graphs and requires whole-program compilation. This is acceptable for most embedded and high performance applications, but presents obstacles to supporting on-the-fly addition of multiple user's queries, or other applications that require changes during execution. The second major design decision is to use a *non-synchronous*, or *event-driven* dataflow model. Synchronicity, in this case, refers to whether the data-rates on streams are known and constant or are variable at run time. This allows WaveScript to handle a broader range of applications than most stream processing languages. We will see that WaveScript replaces the synchronicity assumptions of other digital signal processing (DSP) and stream processing systems with a profile-driven approach that opportunistically leverages data-rate regularities where they occur, while allowing dynamically varying behavior elsewhere.

These two major decisions set up limitations and opportunities for the design. Whole-program compilation can mean long compile times, but provides several advantages to program optimization, including the ability to expand polymorphic source programs (i.e. generic programs which work for multiple types) into a monomorphic form. The monomorphic form allows for more efficient data representations because all types are known at all program points. Also, the profile-driven approach provides information that can be used to schedule computations onto parallel and distributed architectures, including automatically partitioning applications between embedded nodes and heavyweight servers (one of the primary features that WaveScript provides). The compiler can then implement these distributed schedules by automatically generating all communication and synchronization code.

WaveScript also includes many secondary design decisions that could be changed while maintaining much of the thrust of this thesis. For example, WaveScript is a type-safe, polymorphic, higher-order functional language. Rather than specify dataflow graphs directly, WaveScript uses a *multi-stage* evaluation model (metaprogramming) to reduce general purpose programs returning stream values into explicit graphs of operators; this allows expressive, programmatic construction of graphs, while simultaneously allowing compiler optimizations that require access to the graph structure. WaveScript is also minimalist, including only two fundamental dataflow operators: one for merging streams into a single stream, and one for executing a (possibly stateful) user-provided function on every element of a stream. WaveScript includes a means for expressing algebraic rewrite rules for these arbitrary user-provided functions, which compensates for the minimalism in built-in operators. Thus, when compared with a streaming database, WaveScript replaces the fixed set of operators with a general programming language, relying on extensible rewrite-rules (and on profile-driven general purpose stream optimizaitons) to substitute for the query optimizer[1].

Minimialism in the streaming model and runtime is buoyed by certain features in the source language: multi-stage evaluation (metaprogramming), higher-order functions, and first-class streams. These features make WaveScript powerful enough to encode more exotic streaming features or operators as WaveScript libraries rather than by changing the execution model. In Section 4 we will see examples of this: a library that enables synchronous dataflow programming with known data rates, with scheduling performed by library itself (during metaprogram evaluation), and another library that enables migrating operators, in spite of the runtime model supporting only static dataflow graphs.

[1] The use of rewrite rules as a foundation for database optimization has been attempted in several implementations [56].

## 1.1 Background: Design Space Exploration

The present work on this thesis has grown out of experience designing and implementing WaveScript's precursor, Regiment. Regiment is a language for programming sensor networks that was introduced in [48], and substantially revised in [49]. Regiment is based on the concept of *functional reactive programming* (FRP) [23]. Sensor network state is represented as time-varying *signals*. Signals might represent sensor readings on an individual node, the state of a node's local computation, or aggregate values computed from multiple source signals. Regiment also supports the notion of *regions*, which are spatially distributed signals. An example of a region is the set of sensor readings from nodes in a given geographic area. The use of regions make Regiment a "macroprogramming" language in the sense of "programming the network as a whole". Regiment abstracts away the details of sensor data acquisition, storage, and communication from the programmer, instead permitting the compiler to map global operations on signals and regions onto local structures within the network.

#### Regiment Concept #1: Signals

Signals represent continuously varying values. For example, the temperature sensor of a given node has type `Signal<float>`. Conceptually, a signal is a function that maps a time $t$ to a value $v$, but in practice the time $t$ will always be "now" and the signal must be sampled to produce a discrete stream; however, the nature of this sampling is at the discretion of the compiler.

Regiment provides a number of operations for building new signals from existing signals. For example, `smap` applies a function to each value of a signal, producing a new signal. Thus, the code fragment:

```
smap( fun(x) { x > THRESH }, tempvals)
```

converts a sensor's floating-point signal `tempvals` to a boolean signal that is *true* whenever the temperature is above a threshold `thresh`.

## Regiment Concept #2: Regions

Central to Regiment is the concept of a *region*, which, in contrast to a signal, represents a value varying over space as well as time. “Space” in this case refers to the set of nodes in the network rather than points in a continuous Cartesian space. A major job of the Regiment implementation is to enable the illusion of *cohesion*—that a region, sampled at a point in time, presents a coherent set of values drawn from different points in space: a “snapshot”. This must be accomplished without overly expensive synchronization operations, and without sacrificing distributed implementation.

It is important to note that membership in a region may vary with time; for example, the region defined as “temperature readings from nodes where temperature > `THRESH`” will consist of values from a varying set of nodes over time. The membership of a region can also vary due to node failures or communication failures or, alternatively, the addition of new nodes into the network. One of the advantages of programming at the region level is that the application can be insulated from the low-level details of network topology and membership.

Regiment provides three key operations on regions: `rmap`, `rfilter`, and `rfold`. `Rmap` is the region-level version of `smap` and applies a function to each value in the region. For instance,

```
  outreg = rmap(fun(x) {x / SCALEFACTOR}, inreg)
```

divides each value in the region `inreg` by a scaling factor. `Rfilter` applies a function to each member of a region to determine whether each member of the input region should remain in the output region. `Rfold` is used to aggregate the values in a region into a single signal using an associative and commutative combining function. Regiment hides the details of how the aggregation is performed at runtime. In our Regiment implementation aggregation is always performed using a spanning tree rooted at the node consuming the output signal of the `rfold` operation.

### 1.1.1 Evolution of Design

The language has changed substantially from the original Regiment design. This thesis focuses on the more recent WaveScript language. But before discussing the specifics of the language, we will examine the shape of the design space explored in coming to our current design. After several years of experience designing and implementing languages for distributed sensor networks, several abstractions have been tested and sometimes discarded, usually due to diffiulty of implementation. The following are some of the design trade-offs we faced:

- *Continuous Signals vs. Discrete Event Streams* The original Regiment design sought to separate sampling from algorithm specification. That is, the user writes a program over continuous signals, and only at the very end applies a sampling regime to the output. With experience, we felt that this abstraction hides too much, as sampling rates are powerfully relevant in low-powered sensor nodes. Further, arbitrary functions over continuous signals make it impossible to guarantee that discretization will yield a faithful approximation—a problem that's compounded if one adds stateful functions. Thus we eventually moved to a model where *streams* of discrete events replaced *signals*[2]. This means that operations that take windows over streams (for example, an FFT) are specified in terms of an integral numbers of samples rather than in terms of continuous time intervals.

- *Lazy vs. Strict* The original Regiment language was purely functional with a lazy evaluation strategy [37]. WaveScript, on the other hand, is an eager language that includes mutable variables and arrays. This was necessary for achieving the performance we desired. (Simply put, ML implementations are used for high-performance computations, Haskell is generally not.) It kept implementation simpler, and made it easier for those learning the language without previous experience in functional programming.

- *Regions vs. Sets of Streams*

[2]Note that in the remainder of this proposal, the term "signal" will only be used in the sense of "digital signal processing" (DSP) rather than in the sense of Regiment's Signal data-type.

Regions can be nested arbitrarily. They are perfectly well defined in an idealized model with continuous signals, no message loss, and instantaneous transmission and compute times. Given real-world constraints, on the other hand, the coherent snapshots implicit in Regiment's regions prove difficult to implement in a manner acceptable for all applications. This is not to say that the functionality is not desirable, but only that it is something to be achieved through libraries and specialized to each application rather than baked into the core language abstraction in a one-size-fits-all manner. For example, the *rfold* operation implicitly selects a snapshot of samples in each aggregate it produces—but there are a wide variety of methods and priorities that might drive this selection. How narrow must the time-slice be from which the samples are drawn? Is it critical that there is no overlap in the time-slices of adjacent aggregates?

WaveScript, in contrast, currently provides only basic communication facilities—data flows only from sensors to base stations without in-network aggregation. And instead we have focused on compiling, scheduling, and partitioning streaming applications for performance. In the future, WaveScript will regain some of the communication functionality present in the original Regiment[3].

## 1.2 Summary

Ultimately, for the applications we will consider in this thesis, WaveScript provided three key features:

1. **Embedded Operation:** A compiled WaveScript program yields an efficient, compact binary which is well suited to the low-power, limited-CPU nodes used in our applications. WaveScript also includes pragmatic necessities such as the ability to integrate with drivers that capture sensor data, interfaces to various operating system hooks, and a foreign function interface (FFI) that makes it possible to integrate

[3]If we exposed only a simple message intercept facility, that would be sufficient for in-network aggregation. Low-level primitives could be used to build various snapshot mechanisms and rfold operators, which should be carefully cataloged so as to make their different semantics clear to the programmer

legacy code into the system.

2. **Distribution:** WaveScript is a *distributed* language in that the compiler can execute a single program across many nodes in a network (or processors in a single node). Typically, a WaveScript application utilizes both in-network (embedded) computation, as well as centralized processing on a desktop-class "base station". On base stations, being able to parallelize across multiple processors is important, especially as it can speed up offline processing of batch records in after the fact analysis.

   Ultimately, distribution of programs is possible because WaveScript, like other languages for stream-processing, divides the program into distinct stream operators (functions) with explicit communication and separate state. This dataflow graph structure allows a great deal of leeway for automatic optimization, parallelization, and efficient memory management.

3. **Hybrid synchronicity:** WaveScript assumes that streams are fundamentally asynchronous, but allows regularly timed streams to group elements (via special windowing operations) into windows—called Signal Segments, or "Sigsegs"—that have a known time-interval between samples. For example, a stream of audio events might consist of windows of several seconds of audio that are regularly sampled, such that each sample does not need a separate timestamp, but where windows themselves arrive asynchronously, and with variable delays between them. Support for asynchronicity is essential in our applications.

Our research addresses how these requirements can be met effectively by a high-level language. Execution efficiency is accomplished by three key techniques, each of which is enabled or enhanced by the language's domain-specificity. First, the compiler employs a *novel evaluation strategy* that uses a combination of interpretation, reification (converting objects in memory into code), and compilation. This technique evaluates programs into stream dataflow graphs, enabling abstraction and modularity without runtime overhead. Second, WaveScript uses a *profile-driven optimization* strategy to enable compile-time optimization and parallelization of stream dataflow graphs in spite of their asynchronicity. Specifically, WaveScript uses representative sample data to estimate stream's rates and compute times of each operator, and then applies a number of well-understood techniques from the synchronous dataflow world. Third, WaveScript incorporates extensible *algebraic rewrite rules* to capture optimizations particular to a sub-domain or a library. As we will show, providing rewrite rules with a library (for example, containing DSP operators) can enable the user to compose functionality at a high-level, without sacrificing efficiency.

The next chapter, Chapter 2, lays out a representative toy language, called *MiniWS*. Subsequently, Chapter 3 will provide an informal description of the complete WaveScript system. Chapter 5.5 describes the compiler implementation, while leaving matters of optimization to Chapter 6, and Chapter 7 presents the WaveScript algorithms for automatically partitioning dataflow graphs across devices. Finally, Chapter 8 presents empirical results evaluating the system and applications.

# Chapter 2

# A Representative Mini-language

MiniWS is a representative subset of the WaveScript source language (the language fed to the frontend compiler). MiniGraph, on the other hand, is a language similar to the stream dataflow graphs that the backend compiler employs for profiling, partitioning, and code generation. The grammars for MiniWS and MiniGraph are shown in Figures 2-2 and 2-3 respectively. We can see that after the first stage of evaluation is complete (MiniWS is converted to MiniGraph) several language features have disappeared. All function definitions have been inlined. Programs are monomorphic rather than polymorphic. Stream values are no longer first class (they cannot be stored in variables). Instead, the graph structure is made explicit.

Both MiniWS and MiniGraph are fully typed languages. Neither MiniWS nor MiniGraph are concerned with type inference. (WaveScript does provide type inference.) Both languages include product types, but, for simplicity, omit sum types. Sum types could be (inefficiently) mapped onto product types if one were willing to choose dummy values for each type. (That is, $\alpha \mid \beta$ can be represented by $(\texttt{Int}, \alpha, \beta)$ where the integer index selects one of the $\alpha$ or $\beta$ values.) MiniWS omits recursion (letrec), which does appear in the full source language, but not in the full stream graph language. MiniGraph, on the other hand, contains no function definitions at all; whereas in practice it is more desirable to allow some code sharing in the stream graph through global monomorphic, first-order functions.

The `iterate` operator appearing in these grammars is a purely functional formulation of WaveScript's `iterate`. It's type is the following:

$$\texttt{iterate} :: ((\alpha, \sigma) \to (\texttt{List}\,\beta, \sigma)) \to \sigma \to \texttt{Stream}\,\alpha \to \texttt{Stream}\,\beta$$

`Iterate` receives a work function, an initial state, and an input stream. The work function is in invoked on every $\alpha$-typed item in the input stream together with the current state. It produces a new state, and a list of $\beta$-typed elements to be added to the output stream.

## 2.1 Operational Semantics

Here we lay out a simple operational semantics for MiniGraph. We will model the state of the execution by (1) an infinite stream of incoming events to process, and (2) a queue of incoming events for each stream operator. We will concern ourselves with the semantics of the graph and communications between operators, rather than the evaluation of terms within one operator.

Specifically, the ⟨primitive⟩ production in Figure 2-2 is left abstract. These primitives could include various arithmetic and boolean operations. Further, we will not give an evaluation semantics for the standard constructs in MiniGraph or MiniWS. Instead, we will assume an evaluation function $\langle\rangle$, such that $\langle e \rangle \to val$, iff expression $e$ evaluates to $val$ (within an empty environment and store).

Next, we will define two functions for extracting information from MiniGraph programs. The *Init* function constructs an initial state of the model based on a particular MiniGraph program. The *Init* function requires a substitution relation $R$ as input. $R(v)$ denotes the set of symbols for which $v$ is an alias. Evaluating a top-level MiniGraph program is done with the empty relation {}. These substitutions allow us to ignore `merge` operations in our semantics. That is, as long as our graphs support the notion of multiple stream sources connecting to a single sink, then `merge` serves only a renaming purpose. Subscribing to the output stream of a `merge` is the same as subscribing to both its input streams.

⟨exp⟩ ⟶ ⟨literal⟩ | $v$
    | `tuple(` ⟨exp⟩$^+$ `)`
    | `tupleref(`⟨exp⟩`:`⟨type⟩`,` ⟨literal⟩`,` ⟨literal⟩`)`
    | `if` ⟨exp⟩ ⟨exp⟩ ⟨exp⟩
    | `let` $v$ `:` ⟨type⟩ `=` ⟨exp⟩ `in` ⟨exp⟩
    | `while` ⟨exp⟩ `do` ⟨exp⟩
    | `ref(` ⟨exp⟩ `)`
    | `deref(` ⟨exp⟩ `)`
    | $v$ `:=` ⟨exp⟩
    | `begin(` ⟨exp⟩$^+$ `)`
    | ⟨primitive⟩ ⟨exp⟩*

Figure 2-1: Shared expression productions used by both the MiniGraph and MiniWS languages.

⟨prog⟩ ⟶ $v$ | `let` $v$ `=` ⟨stream⟩ `in` ⟨prog⟩
⟨stream⟩ ⟶ `timer` ⟨literal⟩
    | `merge` $v$ $v$
    | `iterate` `(`$\lambda$ $v_1$`:`⟨type⟩ `.` ⟨exp⟩`)` ⟨exp⟩ $v_2$
⟨type⟩ ⟶ `Int` | `Bool` | `List` ⟨type⟩ | `(`⟨type⟩*`)`

Figure 2-2: Grammar for MiniGraph. This is a reduced version of the stream graph representation manipulated by the compiler backend. It is monomorphic and contains no function applications. Only `while` is used for looping.

⟨prog⟩ ⟶ ⟨exp⟩
⟨exp⟩ ⟶ `...` | `timer` ⟨exp⟩
    | `merge` ⟨exp⟩ ⟨exp⟩
    | `iterate` ⟨exp⟩ ⟨exp⟩ ⟨exp⟩
    | $\lambda$ ⟨identifier⟩ `.` ⟨exp⟩
    | `app(`⟨exp⟩`,` ⟨exp⟩`)`

⟨type⟩ ⟶ `Stream` $\langle\overline{type}\rangle$ |
    `Int` | `Bool` | $\alpha$ | `List` ⟨type⟩ | `Ref` ⟨type⟩ | `(`⟨type⟩*`)` | ⟨type⟩ → ⟨type⟩
$\langle\overline{type}\rangle$ ⟶ `Int` | `Bool` | $\alpha$ | `List` $\langle\overline{type}\rangle$ | `Ref` $\langle\overline{type}\rangle$ | `(`$\langle\overline{type}\rangle$*`)`

Figure 2-3: Grammar for MiniWS source language. This language includes abstraction (function definitions), including those that accept and return stream values. For simplicity, the language omits recursion, relying on `while` loops. Note that Stream types may not contain arrows within them.

$\mathit{Init}(\texttt{let}\ v\ \texttt{=}\ \texttt{timer}\ f\ \texttt{in}\ p\texttt{,}\ R) = \mathit{Init}(p, R)$

$\mathit{Init}(\texttt{let}\ v\ \texttt{=}\ \texttt{merge}\ a\ b\ \texttt{in}\ p\texttt{,}\ R) = \mathit{Init}(p,\ \{(v, a),\ (v, b)\}\ \cup\ R)$

$\mathit{Init}(\texttt{let}\ v\ \texttt{=}\ \texttt{iterate}\ t\ i\ s\ \texttt{in}\ p\texttt{,}\ R) =\ \{(R(s),\ v,\ [],\ t,\ \langle i, \sigma_0\rangle)\}\ \cup\ \mathit{Init}(p, R)$

$\mathit{Init}(v) = \{\}$

Here we have introduced a convention of using the let-bound variable names as proxies for operator names. For this purpose, $V$ be the set of all variables used in a program. The state of an operator is a 5-tuple, $(\mathit{in}, \mathit{out}, \mathit{queue}, \mathit{term}, \mathit{state})$, where $\mathit{in} \subset V$ and $\mathit{out} \in V$ name input and output streams. The *queue* is a list of input *messages*, which are values in a simple domain consisting of literals and tuples. Together with an incoming message, *term* and *state* provide all thats needed to execute the work function.

Below, we provide an operational semantics for MiniGraph in terms of a $(S, E)$: a *system state* $S$, which is a set of $(\mathit{in}, \mathit{out}, \mathit{queue}, \mathit{term}, \mathit{state})$ tuples, and an infinite stream $E$ of input timer events. We write $E$ stream simply as a sequence of variable names, corresponding to the `timer` operators in the program. We are not concerned with the timing of these events, only their interleaving (for example, $E = aabaaab...$). We will also use notation for lists. The empty list is written `[]`, adding to the front of a list `h.t` and appending lists `a@b`. We will write tuple values simply as $(a, b, c)$.

First, we define a simple function that appends a list of new messages to the input queue of every operator subscribed to a given name:

$$\begin{aligned} \mathit{Insert}(S, l, v) = S \quad & \cup\ \{(i, o, q@l, e, \sigma) \mid v \in i \wedge (i, o, q, e, \sigma) \in S\} \\ & -\ \{(i, o, q, e, \sigma) \mid v \in i \wedge (i, o, q, e, \sigma) \in S\} \end{aligned}$$

Using $\mathit{Insert}$, we now define an evaluation relation $(S, E) \Rightarrow (S', E')$ to evaluate complete stream graphs together with their input streams. The behavior of $(\Rightarrow)$ is governed by only two rules: one handling the processing of input events, and the other handling of messages in operator queues.

$$\frac{\textsc{TimerInput}}{(S, v.E) \Rightarrow (\mathit{Insert}(S, [()], v), E)}$$

$$\textsc{HandleMsg}\quad \frac{(i, o, h.t, e, \sigma) \in S \qquad \langle \texttt{app}(e,\texttt{tuple}(h, \sigma))\rangle \rightarrow (l, \sigma')}{(S, E) \Rightarrow (\mathit{Insert}(S, l, o) \cup \{(i, o, t, e, \sigma')\} - \{(i, o, h.t, e, \sigma)\}, E)}$$

The above rules leave evaluation order entirely unconstrained. The system may execute any operator that has messages in its input queue, and queues need not be empty before more messages are accepted from the timer sources. This is what we mean by an *asynchronous* stream processing system. Without `merge` both synchronous and asynchronous semantics would give the same deterministic evaluation. In the presence of `merge`, asynchrony results in a non-deterministic interleaving of merged streams.

Unfortunately, without further constraints, the above semantics would admit undesirable executions that never produce output. (For example, the timers could be processed repeatedly but other queues never drained, left to grow indefinitely.) Thus we add a fairness constraint [1] on scheduling, which ensures that any message in any queue will eventually be processed:

- In any infinite execution, all queues are processed an infinite number of times.

In an unbounded memory model, this constraint is sufficient. In practice, operators consuming multiple streams must internally buffer messages until they have sufficient information to complete a computation. Therefore unfairness in the scheduler will result in greater memory usage as operators are forced to buffer more state. So more fairness is better in practice. Examples of fair scheduling strategies include round-robin (queue's "take turns" in a fixed order) or depth-first (after one queue is processed, any new messages enqueued are processed before proceeding). Our current implementation executes on

[1] Other systems require more complex notions of weak and strong fairness [17]. These are equivalent here, because processing each queue is an independent event that neither depends on, or disables the readiness of queues.

multiple processors, where each processor possesses a subset of the operators and checks its input streams in round-robin order, but performs a depth-first traversal upon receiving a message. Many alternatives are possible.

## 2.2 Evaluating MiniWS to MiniGraph

We have not yet addressed how MiniWS is reduced to MiniGraph. Each MiniWS program is reduced to a normal form through the through standard applicative-order reduction rules, just as any other call-by-value language. However, these reductions are insufficient to reduce *any* well formed program in MiniWS into a well formed program in MiniGraph. There are a couple complications. First, the reduction process may not terminate. This we consider an acceptable behavior, as it will only occur when there is a genuine infinite loop in the program that would have surfaced at runtime (in a single-stage instead of multi-stage execution). Second, some programs, even when fully reduced will not conform to the grammar in Figure 2-2. Specifically, that grammar requires that applications and $\lambda$ expressions be eliminated. Consider the following example:

```
let f = λ ...
let g = λ ...
let s = iterate (λb . app(if b then f else g, 99)) init strm
in ...
```

The `app` construct above cannot be reduced, because the expression occupying its operator position is a conditional, in this case, a conditional that depends on stream data that will not be available till runtime [2].

In the present WaveScript implementation, these failures of metaprogram evaluation are reported as compile-time errors (i.e. failure to inline application at code location such and such). For most practical purposes this is sufficient. The errors are encountered at compile-time and are localized to a code location; therefore, they are typically easy for a programmer to understand and fix.

[2]Of course, being smart about it, one might duplicate the conditional's continuation, pushing "99" inside both branches of the "`if`" and fixing this particular example. Yet, in general, higher-order programs contain call-sites invoking indeterminate functions.

In the future, however, it would be desirable to enforce through a type system that MiniWS programs (and likewise WaveScript programs) reduce to MiniGraph programs (or diverge). This would have two primary benefits. First, the point at which an error is encountered would be moved forward in the compiler pipeline. One could imagine a situation where metaprogram evaluation takes hours, making it better to receive errors earlier. Second, a type system would result in a more concise definition for well-formed WaveScript programs; without one, the metaprogram evaluator becomes part of the language specification.

As a first step, Appendix B contains a prototype type system that enforces the successful reduction of metaprograms by restricting the use of higher-order functions (but still permitting many useful usages, such as *map* or *fold*). This prototype should be thought of a sketch—it has not been proven sound.

# Chapter 3

# WaveScript: Full Language

In this section, we informally describe the features of the full WaveScript language as it is implemented and currently available from `http://wavescope.csail.mit.edu`. We introduce the language and provide code examples drawn from our marmot-detection application.

WaveScript is an ML-like functional language with special support for stream-processing. Although it employs a C-like syntax, WaveScript provides type inference, polymorphism, and higher-order functions in a call-by-value language. And like other SP languages [66, 13, 64], a WaveScript program is structured as a set of communicating stream operators. A complete WaveScript program specifies a complete stream graph, a directed graph with stream sources at one end, and a single designated output stream at the other.

In WaveScript, rather than directly define operators and their connections, the programmer writes a declarative program that manipulates named, first-class streams and stream *transformers* (functions that accept and produce streams). Those stream transformers in turn do the work of assembling the stream *operators* that make up the vertices in an executable stream graph. The first stage of evaluation that happens during compile time and assembles this graph is called *meta-program evaluation*. The implementation of this technique will be described in Chapter 5.

## 3.1 Prelude: Applications Overview

While the purpose of this Chapter is to describe the WaveScript language, we will first put it in the context of the applications that we have built using it.

- **Marmot Localization:** This application involves the acoustic localization of animals in the wild and has been deployed in Colorado. We evaluate the performance of this implementation in various ways.
- **EEG Seizure Detection:** This application, introduced in Section 8.3.1, seeks to detect the onset of seizures in 22-channels of EEG data. It is based on work being done by Eugene Shih on mobile health monitoring [61].
- **Speech Detection:** This application, introduced in Section 8.3.3 and based on the approach in [19], detects human speech on a variety of sensor network platforms. Together with the EEG application, the speech detection application is used to evaluate WaveScript's program partitioning capabilities.
- **Background subtraction:** This application is part of a computer vision system used to detect and identify birds in the wild based on work being done in the UCLA Vision lab. It is introduced in Section 8.4, and serves to illustrate the ability of metaprogramming to extend stream-based parallelism to also encompass stateful, parallel processing of matrices.

The marmot localization application will serve as a motivating example in this chapter. Other applications will be described in more detail as they become relevant.

### 3.1.1 Application: Locating Yellow-Bellied Marmots

Marmots, medium-sized rodents native to the southwestern United States, make loud alarm calls when their territory is approached by a predator, and field biologists are interested in using these calls to determine their locations when they call. During our recent deployment, we used WaveScript to build a real-time, distributed localization system that biologists can use in the field, while also archiving raw-data for offline analysis.

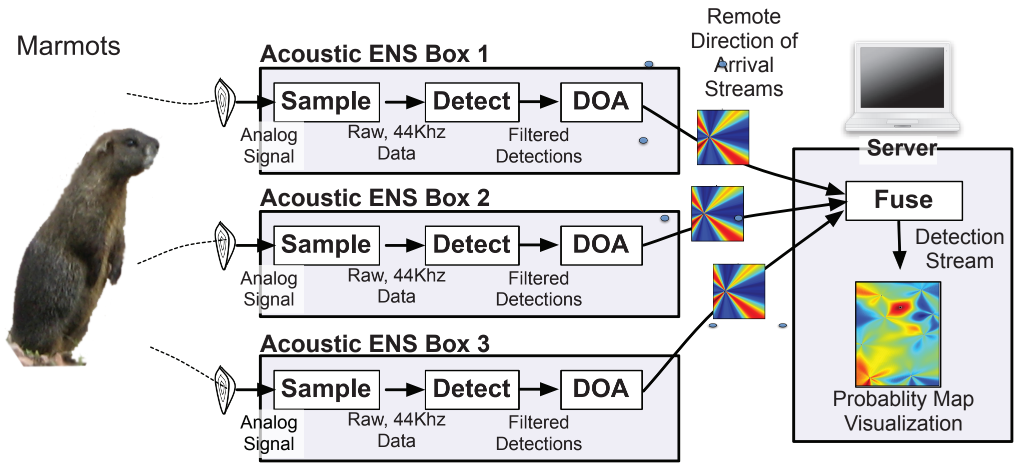


Figure 3-1: Diagram illustrating the major components of the marmot-detection application.

The marmot localization application uses an eight-node *VoxNet* network, based on the earlier acoustic ENSBox nodes [25], using an XScale PXA 255 processor with 64 MB of RAM. Each sensor node includes an array of four microphones as well as a wireless radio for multi-hop communication with the base station (a laptop). The structure of the marmot application is shown in Figure 3-1. The major processing phases implemented by the system are the following.

- **Detect an event**. Process audio input streams, searching for the onset of energy in particular frequency bands.
- **Direction of arrival** (DOA). For each event detected, and for each possible angle of arrival, determine the likelihood that the signal arrived from that angle.
- **Fuse DOAs**. Collect a set of DOA estimates from different nodes that correspond to the same event. For every location on a grid, project out the DOA estimates from each node and combine them to compute a joint likelihood.

## 3.2 A taste of the language

Here's a simple but complete WaveScope program:

```
main = timer(10.0)
```

This creates a timer that fires at 10 Hz. The *return value* of the timer function is a stream of empty-tuples (events carrying no information). The return value of the whole program is, by convention, the stream named "main". The *type* of the above program is Stream (), where () designates the empty-tuple.

In our next program, we will print "Hello world" forever.

```
main = iterate x in timer(10.0) {
         emit "Hello world!";
       }
```

The iterate keyword provides a special syntax for accessing every element in a stream, running arbitrary code using that element's value, and producing a new stream as output. In the above example, our iterate ignores the values in the timer stream ("x"), and produces one string value on the output stream on each invocation (thus, it prints "Hello world!" at 10 Hz). The type of the program is `Stream String`.

Note that we may produce two or more elements on the output stream during each invocation. For example, the following would produce two string elements for each input element on the timer stream.

```
main = iterate x in timer(10.0) {
         emit "Hello world!";
         emit "Hello again!";
       }
```

The above program creates a *stream graph* with two operators: a timer source operator, and an iterate operator that processes the timer's output stream. Timers are the only built-in WaveScope stream sources (aside from foreign sources, see Section 3.9). All other stream procedures only transform existing streams.

Iterate also allows persistent, mutable state to be kept between invocations. This is introduced with the sub-keyword state. For example, the following produces an infinite stream of integers counting up from zero.

```
main = iterate x in timer(10.0) {
         state { cnt = 0; }
         emit cnt;
         cnt := cnt + 1;
       }
```

Notice that the assignment operator for mutating mutable variables (`:=`) is different than the operator used for declaring new variables (`=`)[1]. (WaveScope also has `+=`, `-=` etc for use on mutable variables.) These are second-class mutable variable bindings (i.e., they cannot be passed by reference to functions).

As a final example, we'll merge two streams operating at different frequencies.

```
s1 = iterate x in timer(3.0) { emit 0; }
s2 = iterate x in timer(4.0) { emit 1; }
main = merge(s1,s2)
```

This will output a sequence of zeros and ones, with four ones for every three zeroes. The `merge` operator combines two streams of the same type, interleaving their elements in real time.

## 3.3 Examples using Marmot Application

Figure 3-2 shows the main body of our marmot-detection application. It consists of two sets of top-level definitions—one for all nodes and one for the server. Note that `::` statements declare the types of variables or expressions. In this program, variable names are bound to streams, and function applications transform streams. Network communication occurs wherever node streams are consumed by the server, or vice-versa. (One stream is designated the "return value" of the program by `main = grid`.) First-class streams make wiring stream dataflow graphs convenient.

[1]Actually, all variable bindings are mutable. An `iterate` operator can also be made stateful simply by referencing any mutable state in its lexical scope. However, state references by more than one `iterate` generates an error at meta-program evaluation. So the explicit `state` syntactic form is a good way of enforcing that state not be shared between `iterate`s.

```
// Node-local streams, run on every node:
namespace Node {
  (ch1,ch2,ch3,ch4) = VoxNetAudioAllChans(44100);

  // Perform event detection on ch1 only:
  scores ::  Stream Float;
  scores = marmotScores(ch1);
  events ::  Stream (Time, Time, Bool);
  events = temporalDetector(scores);

  // Use events to select audio segments from all:
  detections = syncSelect(events, [ch1,ch2,ch3,ch4]);

  // In this config, perform DOA computation on VoxNet:
  doas = DOA(detections);
}

// What is not in the Node partition is on the Server:
// Once on the base station, we fuse DOAs:
clusters = temporalCluster(doas);
grid = fuseDOAs(clusters);

// We return these likelihood maps to the user:
main = grid;
```

Figure 3-2: Main program composing all three phases of the marmot-detection application. WaveScript primitives and library routines are in bold. Type annotations are for documentation only.

```
  fun marmotScores(strm) {
    filtrd = bandpass(32, LO, HI, strm);
    freqs  = toFreq(32, filtrd);
    scores =
     iterate ss in freqs {
       emit Sigseg.fold((+), 0,
              Sigseg.map(abs, ss));
     };
    scores
  }
```

Figure 3-3: A stream transformer that sums up the energy in a certain frequency band within its input stream. Energy in this band corresponds to the alarm call of a marmot.

Defining functions that manipulate streams is straightforward. Figure 3.3 shows a stream transformer (`marmotScores`) that implements the core of our event detector—scoring an audio segment according to its likelihood of containing a marmot alarm call. It uses a bandpass filter (window-size 32) to select a given frequency range from an audio signal. Then it computes the power spectral density (PSD) by switching to the frequency domain and taking the sum over the absolute values of each sample.

The return value of a function is the last expression in its body—whether it returns a stream or just a "plain" value. The `marmotScores` function declares local variables (`filtrd`, `freqs`), and uses the `iterate` construct to invoke a code block over every element in the stream `freqs`. The `iterate` returns a stream which is bound to `scores`. In this case, each element in the stream (`ss`) is a window of samples, a *Sigseg*. The Sigseg map and fold (e.g. reduce) functions work just as their analogs over lists or arrays.

The code in Figures 3-2 and 3.3 raises several issues which we will now address. First, we will explain the fundamental `iterate` construct in more depth. Second, we will discuss synchronization between streams. Third, we will address Sigsegs and their use in the marmot-detection application. Finally, we describe how programs are distributed through a network.

## 3.4 Core Operators: iterate and merge

WaveScript is a language built on a small core. In this section, we will examine the primitives that make up the kernel of the language, and serve as the common currency of the compiler. Aside from data sources and network communication points, only two stream primitives exist in WaveScript: `iterate` and `merge`. `Iterate` applies a function to each value in a stream, and `merge` combines streams in the real-time, asynchronous order that their elements arrive.

Both these primitives are stream transformers (functions applied to stream arguments) but they correspond directly to operators in the stream graph generated by the compiler, and are referred to interchangeably as *functions* or *operators*. WaveScript provides special syntactic sugar for `iterate`, as seen above. We will return to this syntax momentarily, but first we present a formulation of `iterate` as a pure, effect-free combinator.

```
iterate :: (((α, σ)→(List β, σ)), σ, Stream α) → Stream β
merge :: (Stream α, Stream α) → Stream α
```

Notice that `iterate` takes only a single input stream; the only way to process multiple streams is to first `merge` them. Also, it is without loss of generality that `merge` takes two streams of the same type: an algebraic sum type (discriminated union) may be used to lift streams into a common type. Similarly, a sum type may be used to multiplex two distinct output streams onto the single output of an iterate operator. It is a matter of optimization to implement this efficiently (i.e. avoid broadcasting all elements of the multiplexed stream to all consumers).

`Iterate` is similar to a *map* operation, but more general in that it maintains state between invocations and it is not required to produce exactly one output element for each input element. The function supplied to `iterate` is referred to as the *kernel function*. The kernel function takes as its arguments a data element from the input stream ($\alpha$), and the current state of the operator ($\sigma$). It produces as output zero or more elements on the output stream (List $\beta$) as well as a new state ($\sigma$). `Iterate` must also take an additional argument specifying the initial state ($\sigma$).

## 3.5 Defining Custom Synchronization Policies

In asynchronous dataflow, synchronization is an important issue. Whereas in a synchronous model, there is a known relationship between the rates of two streams—elements might be matched up on a one-to-one or *n*-to-*m* basis—in WaveScript two event streams have no *a priori* relationship. Yet it is possible to build arbitrary synchronization policies on top of `merge`. And in fact, we find it desirable to make synchronization of streams application-specific.

One example, shown in Figure 3-2, is `syncSelect`. `SyncSelect` takes *windowed* streams (streams of Sigsegs) and produces an output stream containing aligned windows from each source stream. `SyncSelect` also takes a "control stream" that instructs it to

```
fun zip(s1,s2) {
  buf1 = Fifo:new();
  buf2 = Fifo:new();
  iterate msg in mergeTag(s1,s2) {
    switch msg {
      Left (x): Fifo:enqueue(buf1,x);
      Right(y): Fifo:enqueue(buf2,y);
    }
    if (!Fifo:empty(buf1) &&
        !Fifo:empty(buf2))
    then emit(Fifo:dequeue(buf1),
              Fifo:dequeue(buf2));
  }
}
```

Figure 3-4: *Zip*—the simplest synchronization function.

sample only particular time ranges of data. In Figure 3-2, `syncSelect` extracts windows (specified by the `events` stream) containing event detections from all four channels of microphone data on each node.

In Figure 3-4 we define a simpler synchronization function, `zip`, that forms one-to-one matches of elements on its two input streams, outputting them together in a tuple. If one uses only `zip`s to combine streams, then one can perform a facsimile of synchronous stream processing. As prerequisite to `zip`, we define `mergeTag`, which lifts both its input streams into a common type using a tagged union. It tags all elements of both input streams *before* they enter `merge`. (`Left` and `Right` are data constructors for a two-way union type.)

```
fun mergeTag(s1, s2) {
  s1tagged = iterate x in s1 { emit Left(x) };
  s2tagged = iterate y in s2 { emit Right(y) };
  merge(s1tagged, s2tagged);
}
```

Here we have returned to our syntactic sugar for `iterate`s. In Figure 3-4, `Zip` is defined by iterating over the output of `mergeTag`, maintaining buffers of past stream elements, and producing output only when data is available from both channels. These buffers are mutable state, private to `zip`. Note that this version of `zip` may use an arbitrarily large amount of memory for its buffers, but in practice a fair scheduler will keep memory usage down.

As a final example, let's take a look at the definition for `unionList`. This stream operator takes a list of streams, merges them and tags output tuples with the index in the list of its originating stream. A few notes on list operations: `fold` (reduce) operations collapse a list using a binary operator. `Fold1` is variant that assumes the list is not empty, and `mapi` is a variant of map that also passes an elements index in the list to its function argument. The `unionList` function produces a stream graph with a number of merge operators proportional to the length of its input list.

```
fun unionList(ls) {
  using List;
  fold1(merge,
   mapi(fun(i,strm)
         stream_map(fun(x) (i,x), strm),
        ls));
}
```

## 3.6 Windowing and Sigsegs

The `marmotScores` function in Figure 3.3 consumes a stream of Sigsegs. In addition to capturing locally isochronous (regularly spaced in time) ranges of samples, Sigsegs serve to logically group elements together. For example, a fast-Fourier transform operates on windows of data of a particular size, and in WaveScript that window size is dictated by the width of the Sigsegs streamed to it.

A Sigseg contains a sequence of elements, a timestamp for the first element, and a time interval between elements. We refer to a stream of type `Stream (Sigseg` $\tau$`)` as a "windowed stream". All data produced by hardware sensors comes packaged in Sigseg containers, representing the granularity with which it is acquired. For example, the microphones in our acoustic localization application produce a windowed stream of type `Stream (Sigseg Int16)`.

Of course, the audio stream produced by the hardware may not provide the desired window size. WaveScript makes it easy to change the window size of a stream using the *rewindow* library procedure. `Rewindow(size,overlap,s)` changes the size of the windows, and, with a nonzero overlap argument, can make windows overlapping. In our implementation, Sigsegs are read only, so it is possible to share one copy of the raw data between multiple streams and overlapping windows. The efficient implementation of the Sigseg ADT was addressed in [26].

Because windowing is accomplished with Sigsegs, which are first-class objects, rather than a built-in property of the communication channel or an operator itself, it is possible to define functions like `rewindow` directly in the language.

## 3.7 Distributed Programs

A WaveScript program represents a graph of stream operators that is ultimately partitioned into subgraphs and executed on multiple platforms. This partitioning can be user-controlled or automated by the compiler. The current WaveScript implementation is based on separating dataflow graphs into two partitions: a node and a server partition[2]. In the future, WaveScript may support more distinctions between different kinds of platforms in one, heterogeneous network. The code in Figure 3-2 defines streams that reside on the node (namespace `Node`) as well as those on the server. (The "`Node`" namespace is the same irrespective of what physical platform is targetted, platform choice is selected on the command line when invoking the compiler.) Top-level definitions may be used by either. The crossings between the node and server partitions (e.g. named streams declared in the `Node` namespace and used outside of it), become the points at which to cut the graph. Presently, the default is that the cut edges may experience message loss but other streams in the program may not. Note that with the application in Figure 3-2, moving the DOA computation from node to server requires only cutting and pasting a single line of code.

The WaveScript backend compiles individual graph partitions for the appropriate platforms. The runtime deals with disseminating and loading code onto nodes. The networking system takes care of transferring data over the edges that were cut by graph partitioning, for example using TCP sockets. We will return to the issue of graph partitioning in greater detail in Chapter 7 and Section 5.5.

[2]It is a separate matter to schedule operators within a partition onto a multiprocessor platform.

## 3.8 Other Language Features

This thesis does not describe in detail all the features of the WaveScript language. For that purpose, please refer to the user manual, which can be found on the WaveScript website: `http://wavescope.csail.mit.edu/`. However, here we will briefly summarize the major language features that go into the full-fledged WaveScript language. Most of these are a selection of off-the-shelf components from the literature that are implemented for WaveScript using standard techniques.

- **Type Inference** WaveScript employs the standard Hindley-Milner/Damas-Milner type-checking and type-inference algorithms [45] along with a few extensions that do not affect the basic structure of the type-checker. These extensions are as follows.

- **Numeric subkind**

  WaveScript's type system includes a new kind for numeric type variables. These type variables may be bound to only scalar numeric types supporting arithmetic operations: `Ints`, `Floats`, `Complex` numbers and so on. These numeric type variables are written as "`#a`" rather than "`a`". For example, the type of + is "`(#a, #a) → #a`".

  This enables writing code that is reusable with different numeric types (especially relevant in the presence of so many numeric types at various precisions).

  This is a feature that would be subsumed by type classes, were they added to WaveScript. But note that rather than the detailed numeric class hierarchy present in Haskell, WaveScript presently makes no type-level distinctions between the different numeric operations. This design presents simpler types to the user (lacking the complex qualified types one would see in Haskell: Fractional, Integral, etc). Yet not all numeric operations support all types. What happens if you try to take `sin` of an integer? WaveScript cannot catch this error in the initial type-checking pass. However, because of the multi-stage structure of WaveScript, and because after meta-program evaluation the program becomes monomorphic, WaveScript does detect these errors at compile time, which is effectively as good as an error during the initial type check.

- **Generic Printing**

The type of the WaveScript `show` function is "$\alpha \rightarrow$ `String`". Once the program is in a monomorphic form, WaveScript can generate printing code for any type. This same functionality is accomplished by "`deriving Show`" in Haskell.

- **Marshaling**

  WaveScript also has support for marshaling (or serializing) any data structure. This simply requires a pair of functions with the following signatures. (Calls to `unmarshal` requires an explicit type annotation.)

  ```
  marshal :: a -> Array Uint8;
  unmarshal :: Array Uint8 -> a;
  ```

  These functions are necessary for writing streams of data to disk and sending them over network devices. They are used internally by the compiler when generating networking code, as well as directly by the user when they want to add addition communication or storage capabalities.

- **Records** WaveScript implements the extensible record system described in the paper "Extensible Records With Scoped Labels" [40]. These are uniquely important for stream processing applications, which frequently need to pass an increasing amount of metadata attached on each stream element as it passes through layers of processing. The implementation in WaveScript is more efficient than those described in the paper, owing again to the multi-stage structure of the WaveScript language. After meta-program evaluation, when the program is monomorphic, it is possible to rewrite record types as ordered tuples in a straightforward fashion. WaveScript currently represents all records as a flat tuple with fields in alphabetical order by label.

  In some cases, it may be more efficient to instead use a different physical representation. A nested tuple representation could make extension cheaper if tuples are boxed (don't copy all the fields, just add new fields and keep a poniter to the parent record). It's an open question as to whether an optimizer could derive any benefit from using profiling information to globally select the representation for each record type in the program.

- **Foreign Function Interface** WaveScript has a natural and easy to use FFI for interfacing with legacy C code. This system is described in Section 3.9.

## 3.9 Pragmatics: Foreign Interface

The WaveScript compiler provides a facility for calling external (foreign) functions written in C or C++. This functionality is essential to WaveScript, first, for reusing existing libraries (especially DSP related, e.g. Gnu Scientific Library, FFTW, etc), and second, for adding new data sources and data sinks — for network communication, disk access, new hardware sensor types and so on — without modifying the WaveScript compiler itself.

The foreign interface provides a natural mapping between WaveScript types and C types. Barring some documented implementation limitations, the foreign interface exhibits the same behavior irrespective of which WaveScript backend is employed. (Different backends produce code for different runtimes, including Scheme, ML, and C.) The one exception is the TinyOS backend, which represents a sufficiently different platform that it requires its own distinct foreign interface.

There are three WaveScript primitives used to interface with foreign code. The `foreign` primitive registers a single C function with WaveScript. Alternatively, `foreign_source` imports a stream of values from foreign code. It does this by providing a C function that can be called to add a single tuple to the stream. Thus we can call from WaveScript into C and from C into WaveScript. The third primitive is `inline_C`. It allows the programmer to construct arbitrary C code at compile time which is then linked into the final stream query. The programmer can of course call functions within the generated C code from WaveScript just as any other C functions.

### 3.9.1 Foreign functions

The basic foreign function primitive is called as follows: "`foreign(`*`function-name`*`,` *`file-list`*`)`". Like any other primitive function, `foreign` can be used anywhere within a WaveScript program. It returns a WaveScript function representing the corresponding C function of the given name. The only restriction is that any call to the `foreign` primitive *must* have a type annotation. The type annotation lets WaveScript type-check the program, and tells the WaveScript compiler how to convert (if necessary) WaveScript values into C-values when the foreign function is called.

The second argument is a list of *dependencies*—files that must be compiled/linked into the query for the foreign function to be available. For example, the following would import a function "foo" from "foo.c".

```
c_foo ::  Int -> Int = foreign("foo", ["foo.c"])
```

Currently C-code can be loaded from source files (`.c`, `.cpp`) or object files (`.o`, `.a`, `.so`). When loading from object files, it's necessary to also include a header (`.h`, `.hpp`). For example:

```
c_bar =
   (foreign("bar", ["bar.h", "bar.a"])
    :: Int -> Int)
```

Of course, you may want to import many functions from the same file or library. WaveScript uses a very simple rule. If a file has already been imported once, repeated imports are suppressed. (This goes for source and object files.) Also, if you try to import multiple files with the same basename (e.g. "bar.o" and "bar.so") the behavior is currently undefined.

### 3.9.2 Foreign Sources

A call to register a foreign source has the same form as for a foreign function:

"`foreign_source(`*`function-name`*`, `*`file-list`*`)`"

However, in this case the *function-name* is the name of the function being *exported*. The call to `foreign_source` will return a stream of incoming values. It must be annotated with a type of the form `Stream` $T$, where $T$ is a type that supports marshaling from C code.

We call the function exported to C an *entrypoint*. When called from C, it will take a single argument, convert it to the WaveScript representation, and fire off a tuple as one element of the input stream. The return behavior of this entrypoint is determined by the scheduling policy employed by that particular WaveScope backend. For example, it may follow the tuple through a depth-first traversal of the stream graph, returning only when there is no further processing. Or the entrypoint may return immediately, merely enqueuing the tuple for later processing. The entrypoint returns an integer error code, which is zero if the WaveScope process is in a healthy state at the time the call completes. Note that a zero return-code does not guarantee that an error will not be encountered in the time between the call completion and the next invocation of the entrypoint.

Currently, using multiple foreign sources is supported (i.e. multiple entrypoints into WaveScript). However, if using foreign sources, you cannot also use built-in WaveScript "timer" sources. When driving the system from foreign sources, the entire WaveScript system becomes just a set of functions that can be called from C. The system is dormant until one of these entrypoints is called.

Because the main thread of control belongs to the foreign C code, there is another convention that must be followed. When using one or more foreign sources, the user must implement *three* functions that WaveScript uses to initialize, start up the system, and handle errors respectively.

```
void wsinit(int argc, char** argv)
void wsmain(int argc, char** argv)
void wserror(const char*)
```

`Wsinit` is called at startup, before any WaveScript code runs (e.g. before `state{}` blocks are initialized, and even before constants are allocated). `Wsmain` is called when the WaveScript dataflow graph is finished initialing and is ready to receive data. `Wsmain` should control all subsequent acquisition of data, and feed data into WaveScript through the registered `foreign_source` functions. `Wserror` is called when WaveScope reaches an error. This function may choose to end the process, or may return control to the WaveScope process. The WaveScope process is thereafter "broken"; any pending or future calls to entrypoints will return a non-zero error code.

### 3.9.3 Inline C Code

The function for generating and including C code in the compiler's output is `inline_C`. We want this so that we can *generate* new/parameterized C code (by pasting strings together) rather than including a static `.c` or `.h` file, and instead of using some other mechanism (such as the C preprocessor) to generate the C code. The function is called as "`inline_C(`*`c-code`*`,` *`init-function`*`)`". Both of its arguments are strings. The first string contains raw C-code (top level declarations). The second argument is either the null string, or is the name of an initialization function to add to the list of initializers called before `wsmain` is called (if present). This method enables us to generate, for example, an arbitrary number of C-functions dependent on an arbitrary number of pieces of global state. Accordingly we also generate initializer functions for the global state, and register them to be called at startup-time.

The return value of the `inline_C` function is a bit odd. It returns an empty stream (a stream that never fires). This stream may be of any type; just as the empty list may serve as a list of any type. This convention is an artifact of the WaveScript metaprogram evaluation. The end result of metaprogram evaluation is a dataflow graph. For the inline C code to be included in the final output of the compiler, it must be included in this dataflow graph. Thus `inline_C` returns a "stream", that must in turn be included in the dataflow graph for the inline C code to be included. You can do this by using the `merge` primitive to combine it with any other Stream (this will not affect that other stream, as `inline_C` never produces any tuples). Alternatively, you can return the output of `inline_C` directly to the "main" stream, as follows:

```
main = inline_C(...)
```

### 3.9.4 Converting WaveScript and C types

An important feature of the foreign interface is that it defines a set of mappings between WaveScript types and native C types. The compiler then automatically converts, where necessary, the representation of arguments to foreign functions. This allows many C functions

| *WaveScript* | *C* | *explanation* |
|---|---|---|
| `Int` | `int` | native ints have a system-dependent length, note that in the Scheme backend WaveScript `Ints` may have less precision than C `ints` |
| `Uint16` | `unsigned short` | WaveScript supports 8, 16, 32, and 64 bit signed and unsigned integers. |
| `Float` | `float` | WaveScript floats are single-precision |
| `Double` | `double` | |
| `Bool` | `int` | |
| `String` | `char*` | pointer to null-terminated string |
| `Char` | `char` | |
| `Array T` | *T** | pointer to C-style array of elements of type `T`, where `T` must be a scalar type |
| `Pointer` | `void*` | Type for handling C-pointers. Only good for passing back to C. |

Figure 3-5: Mapping between WaveScript and C types. Conversions performed automatically.

to be used without modification, or "wrappers". Figure 3-5 shows the mapping between C types and WaveScript types.

### 3.9.5 Importing C-allocated Arrays

A WaveScript array is generally a bit more involved than a C-style array. Namely, it includes a length field, and potentially other metadata. In some backends it is easy to pass WaveScript arrays to C without copying them, because the WS array contains a C-style array within it, and that pointer may be passed directly.

Going the other way is more difficult. If an array has been allocated (via `malloc`) in C, it's not possible to use it directly in WaveScript. It lacks the necessary metadata and lives outside the space managed by the garbage collector. However, WaveScript does offer a way to *unpack* a pointer to C array into a WaveScript array. Simple use the primitive `‘‘ptrToArray’’`. As with foreign functions, a type annotation is required.

### 3.9.6 "Exclusive" Pointers

Unfortunately, `ptrToArray` is not always sufficient for our purposes. When wrapping an external library for use in WaveScript, it is desirable to use memory allocated outside WaveScript, while maintaining a WaveScript-like API.

For instance, consider a Matrix library based on the Gnu Scientific Library (GSL) (as is included with the WaveScript distribution). GSL matrices must be allocated outside of WaveScript; yet we wish to provide a wrapper to the GSL matrix operations that feels natural within WaveScript. In particular, the user should not need to manually deallocate the storage used for matrices.

For this purpose, WaveScript supports the concept of an *exclusive* pointer. "Exclusive" means that no code outside of WaveScript holds onto the pointer. Thus when WaveScript is done with the pointer the garbage collector may invoke `free` to deallocate the referenced memory. (This is equivalent to calling `free` from C, and will not, for example, successfully deallocate a pointer to a pointer.)

Using exclusive pointers is easy. There is one function `exclusivePtr` that converts a normal `Pointer` type (machine address) into a managed exclusive pointer. By calling this, the user guarantees that that copy of the pointer is the only one in existence. Converting to an exclusive pointer should be thought of as "destroying" the pointer—it cannot be used afterwards. To retrieve a normal pointer from the exclusive pointer, the `getPtr` function can be used, however, when the garbage collector collects the exclusivePtr the pointer returned from getPtr will be invalidated.

# Chapter 4

# Discussion: Language Extensibility

Stream transformers, combined with first-class windowing, has enabled us to implement a variety of libraries that extend WaveScript with new abstractions. In addition to custom synchronization policies, we use WaveScript to define new abstract stream datatypes that model streaming domains other than the push-based asynchronous one. Here we highlight some of the libraries that we have implemented. Implementing these abstractions in most stream-processing languages is difficult or impossible without modifying the underlying compiler or runtime.

## 4.1 Pull-based Streams

We have built a library that provides an ADT for *pull-based* or demand-driven streams. These are useful for a number of purposes. For example, some data sources (such as files), don't have natural data rates and instead can be read on demand. However, if the data file is consumed using the pull-based ADT, then the subsequent results will be available on a demand-driven basis. For example, one could then subsequently *zip* the pull-based stream with a push-based stream, pulling one value on the former for each pushed on the latter, and pairing the results.

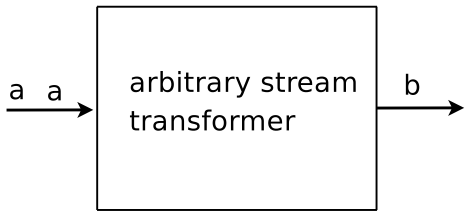
This library is based on the type *`PullStream`*. Internally, a `PullStream` is implemented as a function that takes a stream of requests, and returns a stream of responses.

```
  type PullStream t = (Stream ()) -> Stream t;
```

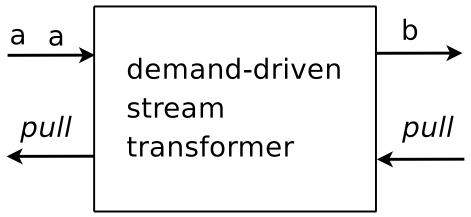
Using that type definition, it is straightforward to construct a library that provides all the desired operations over PullStreams, including, filtering, mapping, applying (stateful) stream transformers. The function for applying arbitrary stream transformers is interesting. The function is called Pull:applyST, and has the following type.

```
(Int, PullStream a, Stream a -> Stream b) -> PullStream b;
```

The integer is a buffer size, which is necessary because the iterate operators produced by the Stream a →Stream b transformer may produce multiple outputs on a single execution, which need to be buffered until there is a demand for them[1]. The stream transformer (`Stream a -> Stream b`) can be visualized as a component with one input and one output. (Of course, this box may internally actually consist of many stream operators.)



Essentially, the job of this library is to add extra wiring to provide a back-channel for pull requests, in addition to the forward channels for data.



The implementation of Pull:applyST is shown in Figure 4-1. Pull-based stream graphs have cycles. Thus, `Pull:applyST` is defined using the WaveScript `feedbackloop` operator, which takes an initial stream and a stream transformer. It ties the stream transformer in a knot (its output is fed back in as input), and merges in the initial stream to provide external input. The event handler has to deal with four kinds of messages corresponding to the four wires in the previous diagram.

[1]If a resizing Fifo implementation is used, this is unnecessary.

For the purposes of the current library's implementation, we assume that the original stream transformer provided by the user will always produce some output for each input. In the future, this could be relaxed by using a timeout—if the user's subgraph does not respond with results after $T$ elapsed time, then send it more data. Further improvement in that respect would require a more intrusive API, where the user provides only a single work function (that takes ($state, streamelement$), and produces ($newstate, outputelements$)) rather than an arbitrary stream transformer.

Also, there are some interesting optimizations that could be applied within the PullStream library that are not currently exploited. Presently, a request for data is relayed up the chain of operators one-at-a-time. If a chain of operators consists of, for example, maps (which are one-to-one input/output) then it is reasonable to short-circuit the backchannels and send the pull request directly to the start of the chain, eliminating unnecessary relaying.

## 4.2 Peek/Pop/Push with Explicit Rates

Another library allows windowed streams to be accessed with a *peek/pop/push* interface rather than by manipulating Sigsegs directly. Each operator is annotated with explicit rates for the peek/pop/pushing. This mimics the style in which StreamIT programs are written. (If these annotations are violated at runtime, an exception is raised.) Like StreamIT, scheduling happens at compile time (metaprogram evaluation time, in this case). The WaveScript library code selects buffer sizes and schedules executions to match input and output rates. The library is invoked by calling "filter" (the StreamIT term for a stream operator) with explicit rates and a user work function as arguments. That work function in turn takes peek, pop, and push functions as arguments.

```
fun Combine(N)
  // Peek/pop N elements, Push just one:
  filter(N,N,1, fun(peek,pop,push) {
    sum = 0;
    for i = 0 to N-1 { sum += pop(1); }
    push(sum);
  })
```

```
uniontype PullIterEvent a b =
  UpstreamPull   ()   |
  DownstreamPull ()   |
  UpstreamResult a    |
  FinalResult    b    ;

fun Pull:applyST(qsize, src, transformer) {
  fun(pullstring) {
    filtFinalResult(
    feedbackloop(stream_map(fun(x) DownstreamPull(()), pullstring),
      fun(loopback) {
        // First any pulls to the upstream we have to route appropriately:
        source = src(filtUpstreamPull(loopback));
        // We apply the user's stream transformer to that upwelling stream of results:
        results = stream_map(UpstreamResult, transformer(source));
        // This is the central event-dispatcher:
        iterate evt in merge(results,loopback) {
         state { buf = FIFO:make(qsize);
                 owed = 0 }
         case evt {
           FinalResult(x)    : {} // This is just the echo from the outbound stream.
           UpstreamPull (_)  : {} // This will have been handled above.

            DownstreamPull(_) :
             {
               // If we have buffered data, respond with that:
               if not(FIFO:empty(buf))
               then emit FinalResult(FIFO:dequeue(buf))
               else {
                 // Otherwise we have to pull our upstream to get some tuples.
                 owed += 1;
                 emit UpstreamPull(())
               }
             }
           UpstreamResult(x) :
               if owed > 0 then {
                 owed -= 1;
                 emit FinalResult(x);
               } else
                 FIFO:enqueue(buf,x)
         }
       }
     }))
  }
```

Figure 4-1: Code for pull-based streams.

This extension shows two things. First, the generality of the WaveScript system, in this case evidenced by its ability to imitate a more restrictive stream model. And, second, that metaprogram-evaluation provides an additional phase, before the dataflow-graph is consumed by the WaveScript compiler, in which additional scheduling or graph manipulation can occur. We will return to this theme in Section 8.4.

## 4.3 Teleporting Messages

Another StreamIT feature worthy of imitation is the teleporting message. Frequently distant operators within an stream graph wish to communicate (for example, when there's a change of configuration). It is painful, however, to refactor all of the operators in between the communication endpoints (to pass along the additional message). Nor should one always add a new edge in the graph directly connecting the two endpoints. In that case, when would the message arrive? Frequently, the messages are meant to remain synchronized with the data that flows through the intervening operators. StreamIT's teleporting messages accomplishes exactly this, without requiring refactoring of intervening operators.

Unfortunately, by tacking on this language feature, StreamIT's semantics and runtime are made more complicated. In WaveScript, we can implement this abstraction in a library without extending the language. We define a datatype for *pass-through* operators. These are stream operators that process their inputs normally, but also pass through an additional channel of data without touching it. Specifically, all outputs in the outgoing normal channel are tagged with the value most recently received on the pass-through channel.

Similar to the `PullStreams` above, this library is based on a definition of pass-through stream transformers. These simply expect both their inputs and outputs to be annotated with an extra field.

```
type PassThruST (a,b,t) = Stream (a * t) -> Stream (b * t);
```

The library then provides various ways to construct and compose these stream transformers (map, filter, etc). The tricky part is to compose stream transformers in the presence of multiple, distinct teleporting messages traveling overlapping paths. The current library leverages WaveScript's extensible record system for this purpose. A teleporting message is sent with a particular record label, and the record system deals with combining multiple distinct labels to build final type that will annotate stream elements. Because WaveScript does not yet have variant types, we use records of option types. For example, when passing a message with label "Msg1", the receiver could test its input to see if it contains a `Msg1` message: `x.Msg1 == None`.

An open question is whether, using only general purpose optimizations, this sort of embedding can be made as efficient as StreamIT's native support for teleporting messages.

## 4.4 Swappable Components

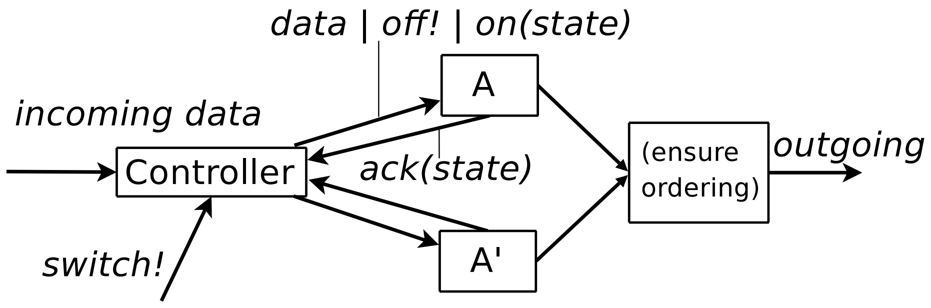
Finally, the most ambitious new stream abstraction that we've built atop WaveScript is a library for building *swappable* components. This allows one to instantiate multiple versions of a stream operator that are swappable at runtime. This can be useful, for example, when there are different implementations of an algorithm that perform better in different situations. Or, it can be used to swap between two identical scopies of a component that are mapped to different physical machines, thereby *migrating* the component.

Like the other stream abstractions above, this library requires that one use its API to build stream transformers within the library's ADT for swappable components. Because swapping components requires freezing and migrating state, it is not possible to swap between arbitrary stream transformers. Instead the user may use the below `makeSwapable` function to build a pair of swappable components by providing two alternate work functions. (These work functions must explicitly take and return their state, $\sigma$, so that the library can manage it.) `Instantiate`, in turn, can invoke a swappable component with an input stream as well as a *control stream* that carries no data (`Stream ()`), but signals when the swap should occur.

```
type WorkFun (a,b,σ) = (a, σ) -> (b, σ);
makeSwappable :: (WorkFun(a,b,σ), WorkFun(a,b,σ)) → Swappable (a,b);
instantiate :: (Stream (), Stream a, Swappable(a,b)) → Stream b;
```

The result of a instantiating a swappable component is pictured below. A controller intercepts all incoming data, as well as “switch!” messages on the control stream. When a switch occurs, the controller stops sending data to the active component ($A$ or $A'$), and instead sends it an “off” message. Upon receiving that message, the active component bundles its state, and sends it back with an “ack(state)” message. The controller buffers all incoming data between sending the “off” message and receiving the “ack” message. Once it receives the “ack”, the controller may send an “on(state)” message to the inactive component, followed by resuming transfer of incoming data to that component (starting with any that has been buffered).

Because of asynchronous communication, output-elements from $A$ may be languishing on $A$’s outgoing channel when $A'$ is activated. For this reason a reordering operator is necessary on the outgoing side. Accordingly, outgoing messages are numbered, and the counter is appended to the state that is handed between $A$ and $A'$.



If the $A/A'$ operator falls at the boundary between the node and the server, then this structure can effectively create a migrating operator. For example, the controller and $A$ could be assigned to the embedded node; $A'$ and downstream operators could be assigned to the server. For example, in our marmot application, $A$ and $A'$ could represent two instantiations of the direction-of-arrival (DOA) component. The controller would be on the sensor node, and would decide whether DOA should be computed locally or sent to the server. Unfortunately, these swappable components do not compose in the way most desirable for migrating computations. If two swappable components are instantiated in sequence, they will be separated in the middle by a controller. This controller can be statically assigned only to one machine. Even if both migratable operators move to the *other* machine, the data transfer between them will still be routed through the first machine! A more composable form of swappable components is a topic for future work.

# Chapter 5

# WaveScript Implementation

The WaveScript language is part of the ongoing WaveScope project, which delivers a complete stream-processing system for high data-rate sensor networks. It includes many components that fall outside the scope of this thesis, including: the networking layer, scheduling engines, and control and visualization software. Instead, in this chapter we focus on the language implementation: compilation and code generation. We save compiler optimizations for Chapter 6, and then save the details of the program partitioning algorithm for Chapter 7.

## 5.1 A Straw-man Implementation

A naive way to implement WaveScript is to use a general-purpose language with support for threads and communication channels, such as Concurrent ML or Java. In that setting, each node of the dataflow graph would be represented by a thread, and the connections between the nodes by channels. Each thread would block until necessary inputs are received on input channels, perform the node's computation, and forward results on output channels.

While such an approach could leverage parallelism, the overhead of running a distinct thread for each node, and of synchronizing through communication channels, would be prohibitive for many parts of the computation. Because the compiler would not have direct access to the structure of the (fixed) stream graph, the job would fall to the runtime to handle all scheduling of threads. For example, if the OS-scheduler is permitted to schedule threads it will load balance them across CPUs, but knowing nothing of the communication structure, will not try to minimize edge crossings across CPU boundaries[1]. This approach is used in some streaming databases, to the detriment of performance [26].

## 5.2 The WaveScript Approach

WaveScript instead exposes the stream graph to the compiler, allowing a range of stream optimizations described in Section 6. The scheduler uses profiling data to assign operators to threads, with one thread per CPU. Each thread, when a data element becomes available, performs a depth-first traversal of the operators on that thread, thereby "following" the data through. This depth-first algorithm is modified to accommodate communication with operators on other threads. Outgoing messages are placed on thread-safe queues[2] whenever the traversal reaches an edge that crosses onto another CPU. Also, the traversal is stopped periodically to check for incoming messages from other threads, while respecting that individual operators themselves are not reentrant. The original WaveScope runtime system was described in [26], though new versions of the system include a reimplementation of the runtime along with the newer ANSI C backend (described in this chapter).

The overall structure of the WaveScript compiler is depicted in Figure 5-1. The *interpret&reify* evaluator, described in the next section, is the critical component that transforms the WaveScript program into a stream graph. Subsequently, it is optimized, partitioned, lowered into a monomorphic, first-order intermediate language, and sent through one of WaveScript's three backends.

### 5.2.1 Execution Model

WaveScript targets applications in which the structure of the stream graph remains fixed during execution. We leverage this property to evaluate all code that manipulates stream

[1]Of course, this scenario would also be costly due to thread context-switching.

[2]Queue implementation is a area worthy of study in its own right. We use the same lock-free algorithm as adopted by Java's concurrency library, described by Michael and Scott in [44]. This implementation provides unbounded queues, but must allocate on each enqueue. In the future, we hope to systematically experiment with alternatives.

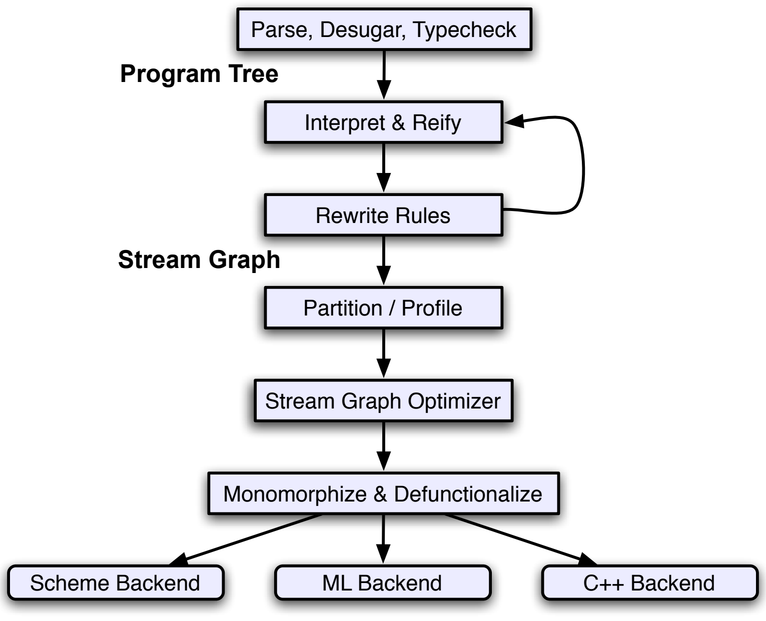


Figure 5-1: Compilation Work-flow and Compiler Architecture

graphs at compile time. For the supported class of programs, however, this multi-phase evaluation is semantics preserving (with respect to a single-phase evaluation at runtime).

A WaveScript evaluator is a function that takes a program, together with a live data stream, and produces an output stream.

$$\texttt{Eval :: } \textit{program} \;\rightarrow\; \textit{inputstream} \;\rightarrow\; \textit{outputstream}$$

Our evaluation method is tantamount to specializing the evaluator (partially evaluating) given only the first argument (the program). The end result is a stream dataflow graph where each node is an `iterate` or a `merge`. (In a subsequent step of the compiler, we perform inlining *within* those `iterate`'s until they are monomorphic and first-order.)

WaveScript's evaluation technique is key to enabling higher-order programming in the performance-critical, resource limited embedded domains—some features (closures, polymorphism, first-class streams) are available to the programmer but evaluated at compile-time. We find that this provides much of the benefit, in terms of modularity and library design, without the runtime costs. Further, this method simplifies backend code generation, as our low-level backends (described in Section 5.4) may omit these features. The main benefit of the evaluation method, however, is that it enables stream graph optimizations. Thus the performance increases described in Section 8.2 can be directly attributed to *interpret&reify*.

Our method is in contrast with conventional *metaprogramming*, where multiple explicit stages of evaluation are orchestrated by the programmer. For example, in MetaML [65], one writes ML code that generates additional ML code in an arbitrary number of stages. This staging imposes a syntactic overhead for quotation and antiquotation to separate code in different stages. Further, it imposes a cognitive burden on the programmer—extra complexity in the program syntax, types, and execution. For the streaming domain in particular, WaveScript provides a much smoother experience for the programmer than a more general metaprogramming framework.

**Interpret & Reify:** Now we explain our method for reducing WaveScript programs to stream graphs. During compile time, we feed the WaveScript source through a simple, call-by-value interpreter. [3] The interpreter's value representation is extended to include *streams* as *values*. The result of interpretation is a stream value. A stream value contains (1) the name of a built-in stream-operator it represents (e.g. `iterate`, `merge`, or a *source* or *network* operator), (2) input stream values to the operator where applicable, and (3) in the case of `iterate`, a *closure* for the kernel function.

The dependency links between stream values form the *stream graph*. All that remains is to *reify* the kernel functions from closures back into code. Fortunately this problem is much studied [60]. Closures become $\lambda$-expressions once again. Variables in the closure's environment are recursively reified as let-bindings surrounding the $\lambda$-expression. The algorithm uses memoization (through a hash table) to avoid duplicating bindings that occur free in multiple closures. These shared bindings become top-level constants.

Let's consider a small example. Within the compiler, the kernel function argument to an `iterate` is always represented (both before and after interpret & reify) by a let-binding for the mutable references that make up its state, surrounding a $\lambda$-expression containing the code for the kernel function. The abstract syntax looks approximately like the following.

```
iterate (let st=ref(3) in λx.emit(x+!st)) S
```

[3] Originally, we used an evaluator that applied reduction rules, including $\beta$- and $\delta$-reduction, until fixation. Unfortunately, in practice, to support a full-blown language (letrec, foreign functions, etc.) it became complex, monolithic, and inscrutable over time, as well as running around 100 times slower than our current interpret/reify approach.

When interpreted, the let-form evaluates to a closure. During reification, mutable state visible from the closure (`st`) is reified into binding code no differently than any other state. However, it is a compile-time error for mutable state to be visible to more than one kernel function. For the simple example above, interpret & reify will generate the same code as it was input. More generally, this pass will eliminate all stream transformers (such as `zip` from Figure 3-4) leaving only `iterate`s, `merge`s, and network/source operators—in other words, a stream graph.

## 5.3 WaveScript Backend Code Generation

WaveScript's compiler front-end uses multiple backend compilers to generate native code. Before the backend compilers are invoked, the program has been profiled, partitioned into per-node subgraphs, optimized, and converted to a first-order, monomorphic form. The WaveScript compiler does not generate native code, but source code in another language. Some would call it a *source-to-source* compiler for this reason. By eliminating some of the trickier language features at compile-time (polymorphism, recursion, higher-order functions), WaveScript avoids the usual need for a compiler to go all the way to native-code so as to optimize the implementation of these difficult features. As a software artifact, one of the advantages of WaveScript is that it is easily retargetable and doesn't require much of potential backend targets. This is a necessary feature for supporting a range of embedded devices.

Ultimately WaveScript targets two families of computer architectures: commodity multicore processors, and embedded processors. Over time, the set of backend code generators has expanded to support specific hardware platforms and operating systems. There were two major generations of the system, corresponding to the first three WaveScript backends (Chez Scheme, MLton, and C++/XStream) and the more recently added backends (ANSI C, TinyOS, JavaME). Both generations of the system are described here for completeness.

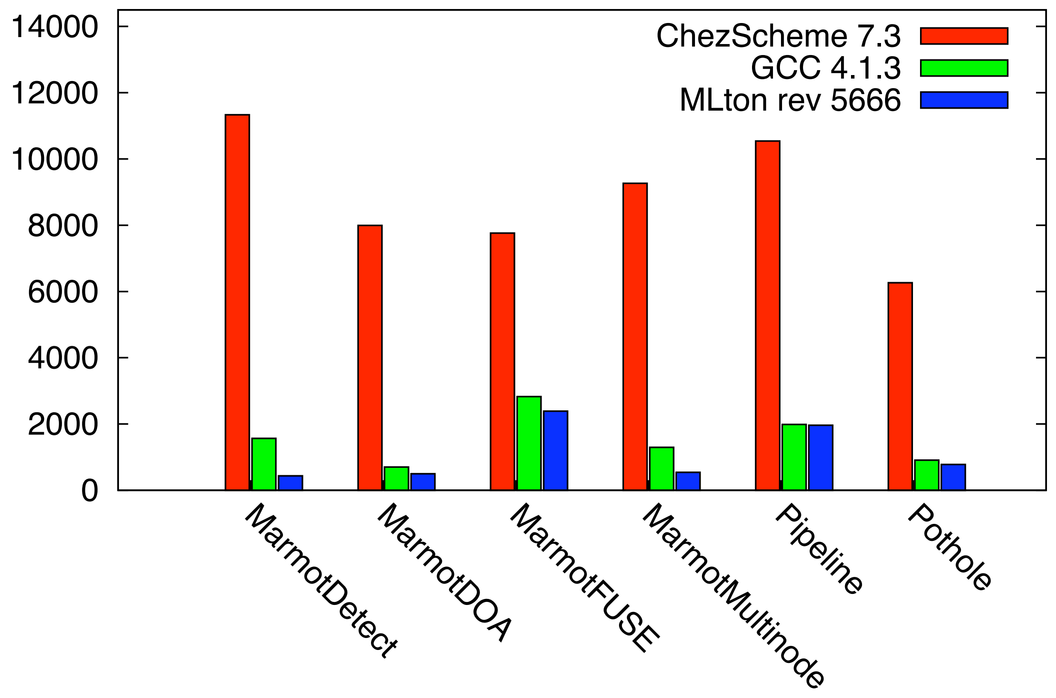


Figure 5-2: Execution times (in milliseconds) of legacy WaveScript backends on application benchmarks. Single-threaded benchmark on AMD Barcelona, optimizations enabled. Benchmarks include three stages of the marmot application (detection, DOA, and fusing DOAs), as well as a complete multinode simulation—eight nodes simulated on one server, as when processing data traces offline. Also included are our pipeline leak detection, and road surface anomaly (pothole) detection applications.

## 5.4 Legacy Backends

The first generation of the WaveScript system used three compilers as backends for generating native code: Chez Scheme [22], MLton [67], and G++/GCC. These backends each provide a different combination of compile-time, debugging, performance, and parallelism. The backends' relative performance on a benchmark suite is shown in Figure 5-2.

**Scheme backend:** The WaveScript compiler itself is implemented in the Scheme programming language. Accordingly, the first, and simplest backend is simply an embedding of WaveScript into Scheme using macros that make the abstract syntax directly executable. This backend is still used for development and debugging. Furthermore, it enables faster compile times than the other backends. And when run in a special mode, it will enable direct evaluation of WaveScript source immediately after type checking (without evaluating to a stream-graph). This provides the lowest-latency execution of WaveScript source, which is relevant to one of our applications that involves large numbers of short-lived WaveScript "queries" submitted over a web-site. It also keeps us honest with respect to our claim that our reification of a stream graph yields exactly the same behavior as direct execution.

**MLton backend:** MLton is an aggressive, whole-program optimizing compiler for Standard ML. Generating ML code from the kernel functions in a stream graph is straightforward because of the similarities between the languages' type systems. This provided us with an easy to implement single-threaded solution that exhibits surprisingly good performance [67], while also ensuring type- and memory-safe execution. In fact, it is with our MLton backend that we beat the handwritten C version of the acoustic localization application. MLton in itself is an excellent option for building embedded software, if GC pauses can be tolerated. However, using MLton directly would forego WaveScript's stream graph optimization.

**C++ backend :** Originally, we had intended for our C++ backend to be the best-performing of the three backends, as it includes a low-level runtime specifically tailored for our streaming domain. However, in the MLton backend actually outperforms our C++ backend, due to three primary factors:

1. The C++ backend leverages the flexible WaveScope scheduling engine for executing stream graphs. The engine supports several different scheduling policies and combinations thereof. The cost of this flexibility is that transferring control between operators is at least a virtual method invocation, and may involve a queue. The MLton and Scheme backends support only single-threaded depth-first traversal, where control transfers between operators are direct function calls.

2. MLton incorporates years of work on high-level program optimizations that GCC cannot reproduce (the abstractions are lost in the C code), and which we do not have time to reproduce within the WaveScript compiler.

3. Our prototype uses a naive reference counting scheme (with cycles prevented by the

type system) that is less efficient than MLton's tracing collector. (Although it does reduce pauses relative to MLton's collector.) In the future we believe that we can implement a substantially more efficient domain-specific collector by combining deferred reference counting with the fact that our stream operators do not share mutable state.

As we show in Section 8, in spite of its limitations, our current prototype C++ runtime is the best of these choices when parallelism is available. This is important in several of our applications where large quantities of offline data need to be processed quickly on multi-core/multiprocessor servers, such as when evaluating our algorithms on over a *terabyte* of accumulated marmot audio data. The MLton runtime and garbage collector do not support concurrent threads, and it would be a daunting task to add this functionality. We could, however, attempt process-level parallelism using MLton, but because MLton does not directly support inter-process shared memory, this would require additional copying of data.

## 5.5 Recent Backends

Unfortunately, none of the legacy backends above could plausibly be adapted to program extremely resource constrained devices such as Telos motes. When we decided to target NesC/TinyOS, our first step was to replace the WaveScript C++ backend with a low-level ANSI C backend. This C backend is used to execute the server-side portion of a partitioned program, as well as the node-side portion on Unix-like embedded platforms that run C, such as the iPhone and the Gumstix.

### 5.5.1 Code Generation: ANSI C

The original backend depended on features such as Boost smart pointers and templates (for generic printing and so on). There were sufficient C++ dependencies that we decided to reimplement the code generator rather than modify the existing one[4]. In particular, the C++

[4]As a minor point, an additional motivation was to rewrite the generator in an object-oriented fashion so as to subclass the TinyOS and Java code generation from the basic generator.

backend depended on the separate XStream scheduling engine. Removing that dependency and handling scheduling directly within the generated code adds complexity but allows the compiler greater control and generally improves performance.

By default, our C code-generator produces single threaded code in which each operator becomes a function definition. Passing data to a downstream operator is accomplished by calling its function. As with the C++ backend, the newer C backend also produces threaded code. It uses one thread per stream operator and relies on earlier phases of the compiler (fusion/fision, Chapter 6) to groom the stream graph to contain an appropriate number of operators.

As we improved our compiler and ANSI C code generator, the performance of our C backend surpassed the MLton compiler. Currently the ANSI C backend is WaveScript's flagship, and except for some as-yet-unsupported features we turn to it first in both single- and multi-threaded scenarios.

### 5.5.2 ANSI C Backend: Stream-Optimized Garbage Collection

One of the ways that the C backend improved over its predecessor was through superior garbage collection. The C++ backend used simple reference counting, whereas the C backend provides—in addition to simple reference counting and conservative collection via the Boehm collector [11]—an efficient form of deferred reference counting. We call this collector *stream optimized* because it takes advantage of the structure of streaming programs (independent, atomically executed stream operators) to improve collection times.

Two restrictions on the source language make this possible. First, the language only allows programs that generate dataflow-graphs to be written. Second, by forbidding recursive datatypes, WaveScript avoids cycles, and therefore avoids the necessity of a backup tracing collector to detect the cycles that a reference counting collector will miss. Without recursive datatypes, long pauses for freeing objects also become less likely (because large linked structures are less common) even without implementing more sophisticated incremental deallocation strategies. (And, indeed, the lack of cycles benefits the older C++ backend as well.)

Traditional deferred reference counting [20] ignores pointers from the stack (local variables), which typically account for a majority of reference count updates. The downside of this approach is that heap objects may not be immediately discarded upon their reference count going to zero and may therefore contribute to larger space consumption. Instead, these pointers are stored in a "zero count table" (ZCT). Periodically, the stack is traced, and once those pointers are taken into account, then objects in the ZCT that still have no references can be deallocated.

We have additional information about the structure of a WaveScript program's execution: it consist of independent, atomically executed stream operators. Because the WaveScript semantics do not allow shared mutable state between operators, each operator logically has its own "heap". Therefore it is a legitimate strategy to keep disjoint physical heaps, in which case collections may proceed independently from one another. This is ideal for parallel garbage collection. But with a deferred reference counting approach there is an additional benefit. When an operator is finished executing, there are no stack frames to trace. Therefore the ZCT can immediately be cleared. Thus we keep the benefits of deferred reference counting while avoiding the costs[5].

The problem with the disjoint heap model is that it precludes opportunities for sharing immutable objects between operators. It does not, however, require that objects always be copied when they are transmitted between operators. It is rare in practice for an operator that allocates a new object, to both keep it *and* forward it along. Thus it is sensible to simply transfer ownership of objects to between operators, as long as the source operator relinquished all pointers to it. This optimization requires, however, a shared memory implementation.

[5]However, deferred reference counting may result in substantially larger memory footprints, as objects are collected later. Also, while stream operators are typically kept small (fine grained), they are of course permitted to perform arbitrary allocation and computation. In this case a backup collector must be in place to perform a collection before the end of an operator's execution. A conservative collector can serve as a simple backup.

### 5.5.3 Code Generation: TinyOS 2.0

Supporting TinyOS 2.0 posed a number of challenges, including the extreme resource constraints of motes, the fact that TinyOS requires tasks to be relatively short-lived, and the requirement that all IO be performed with split-phase asynchronous calls.

Our implementation does not support the entirety of the WaveScript language on TinyOS. Notably, we don't support dynamic memory management in code running on motes. We may support memory management in the future, for example using TinyAlloc together with reference counting, but it remains to be seen whether this style of programming can be made effective for extremely resource constrained devices. We note that to preserve the existing failure modes of WaveScript (dropped stream items on designated edges, but no abortion of execution/corruption of operator state), the compiler would need to move dynamic allocations ahead of any operator-state mutation or else checkpoint the state.

#### Scheduling by Cooperative Multitasking

The most difficult issue in mapping a high-level language onto TinyOS is handling the TinyOS concurrency model. All code executes in either *task* or *interrupt* context, with only a single, non-preemptive task running at a time. WaveScript simply maps each operator onto a task. Each data element that arrives on a source operator, for example a sensor sample or an array of samples, will result in a depth-first traversal of the operator graph. This graph traversal is not re-entrant. Instead, the runtime buffers data at the source operators until the current graph traversal finishes.

This simple design raises several issues. First, task granularity should fall within a restricted range. Tasks with very short executions incur unnecessary overhead, and tasks that run too long degrade system performance by starving important system tasks (for example, sending network messages). Second, the best method for transferring data items between operators is no longer obvious. In the basic C backend, we simply issue a function call to the downstream operator, wait for it to complete, and then continue computation. If we attempt this method in TinyOS, we are forced to perform an entire traversal of the graph in a single very long task execution. The alternative, to execute an operator in its entirety before executing any downstream operators, would require a queue to buffer all output elements of the current operator.

We now describe our solution to each of these problems:

**Merging Tasks:** WaveScript provided an existing framework for merging operators for optimization purposes. By simply controlling the parameters involved, we can prevent WaveScript from generating tasks smaller than a certain granularity. Their small execution times (gleaned from profiling data) will cause them to be merged with adjacent upstream or downstream operators.

**Splitting Tasks:** Splitting existing large tasks into smaller pieces is much more difficult than merging but is critically important for system health. Splitting involves inserting *yield points* to create a cooperative multitasking system. Once the yield points are selected, it is possible for WaveScript to split operators at those points by saving the current state (local variables) of the execution. In the TinyOS context, a task yields by storing its state and reposting itself. This process is made easier by the fact that WaveScript operators do not share mutable state or hold locks. Of course, storing the state of a stalled operator execution requires additional memory. But because graphs are acyclic, and only a single traversal of the graph is active at once, only one stored copy of each operator's state is needed.

The benefits of cooperative multithreading in embedded systems and sensor networks are well appreciated. But in general, it is difficult to automatically insert yield points in arbitrary, Turing-complete programs. Not knowing the bounds on loops or the amount of time spent within them makes it difficult to insert yield points. For example, the loop could yield on every iteration, or every N iterations, where N is either fixed heuristically or based on a estimate of the loops execution time derived from program analysis.

WaveScript instead uses profile data to determine yield points. If an operator consistently takes approximately $T$ time to execute, where $T$ is greater than a fixed limit $L$, the system must insert $\lceil T/L \rceil - 1$ yield points. The goal is to evenly divide the execution. By mapping points in an execution trace back onto code locations (using the profiling method from Section 6.1) WaveScript determines where to insert yield points. Optimizing the precise placement of yield points is an open question.

**Moving Data Elements:** Once tasks are split and merged to the correct granularity, there still remains the issue of inter-operator communication. Here, we again leverage the cooperative multithreading method. A WaveScript operator sends data to its output stream using the *emit* keyword. We insert a yield point after each emit command. The running task schedules the downstream operator (by posting a task). It then stores its state, re-posts itself, and yields. This enables us to process stream elements as soon as they are produced, just as in the basic C backend, without a monolithic task.

This same method could be used to split WaveScript work functions at blocking function call sites which (in TinyOS) become *asynchronous*. For example, reading a sensor or flushing the printf port requires an asynchronous, split-phase call in TinyOS. However, such calls introduce unpredictable delays in a graph traversal that interfere with our automatic partitioning and scheduling. Thus, we presently assume that no split-phase calls occur within WaveScript work functions. (Split-phase calls to acquire data are made by the system *before* beginning a graph traversal.)

### Messaging

Aside from scheduling, messaging is the most important TinyOS feature abstracted by WaveScript. The user writes programs in terms of an abstract stream model where individual stream elements may be of arbitrary size. Stream transport in TinyOS involves subdividing large stream elements across multiple messages[6], marshaling data from memory into packets, routing packets out of the network, and retransmitting dropped messages. Though the model does not demand perfectly reliable channels (Section 3.7) because stream items may be broken up across multiple messages, message loss is particularly painful.

Fortunately, the TinyOS 2.0 software suite simplifies many of these tasks. The Collection Tree Protocol (CTP) provides a convenient routing tree implementation that we use to transport all streams from the network. CTP also takes care of link-level retransmission. The power management system of TinyOS 2.0 allows the microprocessor to switch into a low-power mode when there are no active tasks—without being explicitly told to do so.

[6]It is nevertheless beneficial for the programmer to keep in mind the size range of TinyOS messages when designing their program.

Also, the MIG tool generates code to unpack C structs from network messages. However, that leaves WaveScript with the job of choosing binary formats for more complex WaveScript datatypes (e.g. tuples containing pointer types such as arrays), and of dividing stream elements across messages, numbering, and reassembling them.

### 5.5.4 Code Generation: JavaME

JavaME is a portable execution environment for cellular phones. Generating code for JavaME is much simpler than for TinyOS, with Java itself providing a high level programming environment that abstracts hardware management. The basic mapping between the languages is the same as in the C backend. Operators become functions, and an entire graph traversal is a chain of function calls. In fact, the JavaME code generator is a simplified version of the C generator. Reference counting is omitted, with the JVM handling garbage collection. It is straightforward to construct Java datatypes to represent WaveScript types (e.g. lists, arrays). However, other minor stumbling blocks unique to Java arise: for example, boxing and unboxing scalar types, or the lack of unsigned integer types. Especially on embedded platforms, the performance of the Java backend relative to C was surprisingly poor, which may in part be due to these difficulties in mapping the types.

# Chapter 6

# Optimization Framework

With the basic structure of the compiler covered, we now focus on the optimization framework. The cornerstone of this framework is the profiling infrastructure, which gathers information on data-rates and execution times that subsequently enable the application of graph optimizations from the synchronous dataflow community. In this section we'll also cover our method for performing algebraic rewrite optimizations, which are not currently driven by profiling information.

## 6.1 Profile & Partition

The WaveScript compiler, implemented in the Scheme language, can profile stream graphs by executing them directly within Scheme during compilation (using sample input traces). This produces platform-independent data rates, but cannot determine execution time on embedded platforms. For this purpose, we employ a separate profiling phase on the device itself, or on a cycle-accurate simulator for its microprocessor.

First, the partitioner determines what operators might possibly run on the embedded platform, discounting those that are pinned to the server, but including movable operators together with those that are pinned to the node. The code generator emits code for this partition, inserting timing statements at the beginning and end of each operator's work function, and at emit statements, which represent yield points or control transfers downstream. The partition is then executed on simulated or real hardware. The profiler generates a visual-

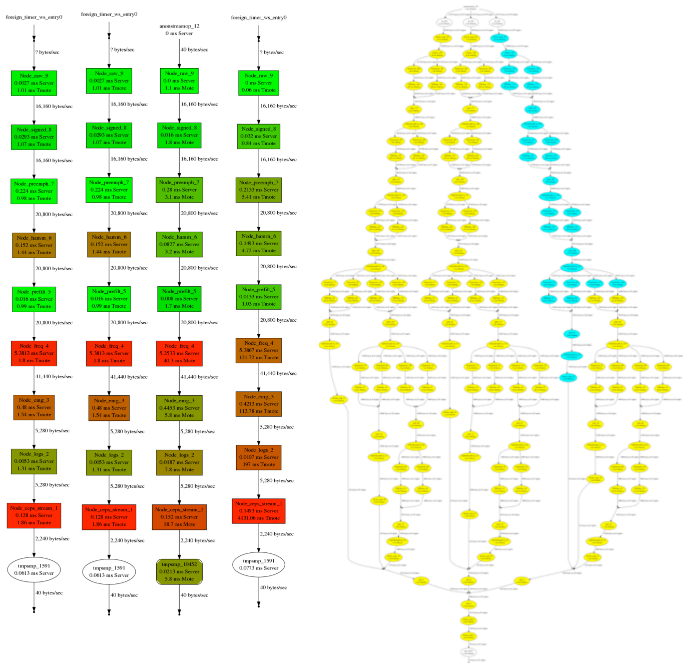


Figure 6-1: Visualizations produced by the compiler. (Labels not intended to be legible; see Figure 8-6 for a clearer rendering of the left-hand graph.) On the left, profiled graphs for a speech detection application, including data rates and CPU consumption, for each of four embedded platforms: TMote, N80, iPhone, and Gumstix (PXA255). On the right, cascades of filters corresponding to three channels of our 22-channel EEG application. Turquoise indicates operators assigned to the embedded node in an optimal partitioning for a particular CPU budget.

ization summarizing the results for the user. We show an example of the visualization in Figure 6-1.

Because our current model only allows communication from node to server, it is unnecessary for the profiler to instantiate the server partition to gather profile data. It is sufficient to execute only the embedded partition and discard its output stream(s). The inserted timing statements print output to a debug channel read by the compiler. For example, we execute instrumented TinyOS programs either on TMote Sky motes or by using the MSPsim simulator[1]. In either case, timestamps are sent through the USB serial port, where they are collected by the compiler.

For most platforms, the above timestamping method is sufficient. That is, the only relevant information for partitioning is how long each operator takes to execute on that platform

[1]We also tried Simics and msp430-gdb for simulation, but MSPsim was the easiest to use. Note that TOSSIM is not appropriate for performance modeling.

(and therefore, given an input data rate, the percent CPU consumed by the operator). For TinyOS, some additional profiling is necessary. To support subdividing tasks into smaller pieces, we must be able to perform a reverse mapping between points in time (during an operator's execution) and points in the operator's code. Ideally, for operator splitting purposes, we would recover a full execution trace, annotating each atomic instruction with a clock cycle. Such information, however, would be prohibitively expensive to collect. We have found it is sufficient to instead simply time stamp the beginning and end of each `for` or `while` loop, and count loop iterations. As most time is spent within loops, and loops generally perform identical computations repeatedly, this enables us to roughly subdivide execution of an operator into a specified number of pieces.

After profiling, control transfers to the partitioner. The movable subgraph of operators has already been determined. Next, the partitioner formulates the partitioning problem in terms of this subgraph, and invokes an external solver (described in Section 7) to identify the optimal partition. When the partitioning is computed, a visualization of the partition is presented to the user. The program graph is repartitioned along the new boundary, and code generation proceeds, including generating communication code for cut edges (e.g., code to marshal and unmarshal data structures).

The stream-graph optimizations described in this chapter are applied independently to each partition, corresponding to the operators that run on a single physical node (possibly with multiple cores/processors). The rewrite optimizations, on the other hand, are applied during metaprogram evaluation and before partitioning.

## 6.2 Stream Graph Optimizations

There are a breadth of well-understood transformations to static and dynamic dataflow graphs that adjust the parallelism within a graph—balancing load, exposing additional parallelism (fission), or decreasing parallelism (fusion) to fit the number of processors in a given machine. The StreamIt authors identify *task*, *data*, and *pipeline* parallelism as the three key dimensions of parallelism in streaming computations [66]. Task parallelism is the naturally occurring parallelism between separate branches of a stream graph. Data parallelism occurs when elements of a stream may be processed in parallel, and must be teased out by replicating (fissioning) operators into multiple workers. Pipeline parallelism is found in separate stages (downstream and upstream) of the stream graph that run concurrently.

We have not taken the time to reproduce all the graph optimizations found in StreamIt and elsewhere. Instead, we have implemented a small set of optimizations in each major category, so as to demonstrate the capability of our optimizer framework—through edge and operator profiling—to effectively implement static graph optimizations normally found in the synchronous dataflow domain. Keep in mind that these optimizations are applied after the graph has been partitioned into per-node (e.g. a VoxNet node or laptop) components. Thus they affect *intra*-node parallelism (e.g., placement onto cores of a processor). Our partitioning methodology separately treats the question of what should go on different physical nodes.

**Operator placement:** For the applications in this paper, sophisticated assignment of operators to CPUs (or migration between them) is unnecessary. We use an extremely simple heuristic, together with profiling data, to statically place operators. We start with the whole query on one CPU, and when we encounter split-joins in the graph, assign the parallel paths to other CPUs in round-robin order, *if* they are deemed "heavyweight". Our current notion of heavyweight is a simple threshold function on the execution time of an operator (as measured by the profiler). This exploits task-parallelism in a simple way but ignores pipeline parallelism.

**Fusion:** We *fuse* linear chains of operators so as to remove overheads associated with distinct stream operators. Any lightweight operators (below a threshold) are fused into either their upstream or downstream node depending on which edge is busier. This particular optimization is only relevant to the C++ backend, as the Scheme and MLton backends bake the operator scheduling policy into the generated code. That is, operators are traversed in a depth first order and `emits` to downstream operators are simply function calls.

**Fission: Stateless Operators:** Any stateless operator can be duplicated an arbitrary number of times to operate concurrently on consecutive elements of the input stream. (A round-robin splitter and joiner are inserted before and after the duplicated operator.) The current WaveScript compiler implements this optimization for `maps`, rather than all stateless operators. A `map` applies a function to every element in a stream, for example, `map(f,s)`. In WaveScript, `map` is in fact a normal library procedure and is turned into an anonymous `iterate` by interpret-and-reify. We recover the additional structure of `maps` subsequently by a simple program analysis that recognizes them. (A `map` is an `iterate` that has no state and one `emit` on every code path.) This relieves the intermediate compiler passes from having to deal with additional primitive stream operators, and it also catches additional `map`-like `iterates` resulting from other program transformations, or from a programmer not using the "map" operator per-se.

Wherever the compiler finds a map over a stream (`map(f,s)`), if the operator is deemed sufficiently *heavyweight*, based on profiling information, it can be replaced with:

```
(s1,s2) = split2(s);
join2(map(f,s1), map(f,s2))
```

Currently we use this simple heuristic: split the operator into as many copies as there are CPUs. Phase ordering can be a problem, as fusion may combine a stateless operator with an adjacent stateful one, destroying the possibility for fission. To fix this we use three steps: (1) fuse stateless, (2) fission, (3) fuse remaining.

**Fission: Array Comprehensions:** Now we look a splitting heavyweight operators that do intensive work over arrays, specifically, that initialize arrays with a non-side-effecting initialization function. Unlike the above fusion and fission examples, which exploit existing *user*-exposed parallelism (separate stream kernels, and stateless kernels), this optimization represents additional, *compiler*-exposed parallelism.

Array comprehensions are a syntactic sugar for constructing arrays. Though the code for it was not shown in Section 3, array comprehensions are used in both the second and third stages of the marmot application (DOA calculation and FuseDOA). The major work of both these processing steps involves searching a parameter space exhaustively, and recording the results in an array or matrix. In the DOA case, it searches through all possible angles of arrival, computing the likelihood of each angle given the raw data. The output is an array of likelihoods. Likewise, the FuseDOA stage fills every position on a grid with the likelihood that the source of an alarm call was in that position.

The following function from the DOA stage would search a range of angles and fill the results of that search into a new array. An array comprehension is introduced with `#[ | ]`.

```
fun DOA(n,m) {
  fun(dat) {
    #[ searchAngle(i,dat) | i = n to m ]
  }
}
```

With this function we can search 360 possible angles of arrival using the following code: `map(DOA(1,360),rawdata)`. There's a clear opportunity for parallelism here. Each call to `searchAngle` can be called concurrently. Of course, that would usually be too fine a granularity. Again, our compiler simply splits the operator based on the number of CPUs available.

```
map(Array.append,
  zip2(map(DOA( 1, 180), rawdata)
       map(DOA(181,360), rawdata)))
```

In the current implementation, we will miss the optimization if the kernel function contains any code other than the array comprehension itself. The optimization is implemented as a simple program transformation that looks to transform any heavyweight `maps` of functions with array comprehensions as their body.

### 6.2.1 Batching via Sigsegs and Fusion

High-rate streams containing small elements are inefficient. Rather than put the burden on the runtime engine to buffer these streams, the WaveScript compiler uses a simple program transformation to turn high-rate streams into lower-rate streams of Sigseg containers. This transformation occurs after interpret-and-reify has executed, and after the stream graph has been profiled.

The transformation is as follows: any edge in the stream-graph with a data-rate over a given threshold is surrounded by a `window` and `dewindow` operator. Then the compiler repeats the profiling phase to reestablish data-rates. The beauty of this transformation is that its applied unconditionally and unintelligently; it leverages on the fusion optimizations to work effectively.

Let's walk through what happens. When a `window/dewindow` pair is inserted around an edge, it makes that edge low-rate, but leaves two high-rate edges to the *left* and to the *right* (entering `window`, exiting `dewindow`). Then, seeing the two high-rate edges, and the fact that the operators generated by `window` and `dewindow` are both *lightweight*, the fusion pass will merge those lightweight operators to the left and right, eliminating the high-rate edges, and leaving only the low-rate edge in the middle.

## 6.3 Extensible Algebraic Rewrites

The high-level stream transformers in WaveScript programs frequently have algebraic properties that we would like to exploit. For example, the windowing operators described in Section 3 support the following laws:

$$\begin{aligned}
dewindow(window(n, s)) &= s \\
window(n, dewindow(s)) &= rewindow(n, 0, s) \\
rewindow(x, y, rewindow(a, b, s)) &= rewindow(x, y, s) \\
rewindow(n, 0, window(m, s)) &= window(n, s)
\end{aligned}$$

In the above equations, it is always desirable to replace the expression on the left-hand side with the one on the right. There are many such equations that hold true of operators in the WaveScript Standard Library. Some improve performance directly, and others may simply pave the way for other optimizations, for instance:

$$map(f, merge(x, y)) = merge(map(f, x), map(f, y))$$

WaveScript allows *rewrite rules* such as these to be inserted in the program text, to be read and applied by the compiler. The mechanism we have implemented is inspired by similar features in the Glasgow Haskell Compiler [32]. However, the domain-specific interpret-and-reify methodology enhances the application of rewrite rules by simplifying to a stream graph before rewrites occur—removing abstractions that could obscure rewrites at the stream level. Extensible rewrite systems have also been employed for database systems [57]. And there has been particularly intensive study of rewrite rules in the context of signal processing [3].

It is important that the set of rules be *extensible* so as to support domain-specific and even application-specific rewrite rules. (Of course, the burden of ensuring correctness is on the rules' writer.) For example, in a signal processing application such as acoustic localization, it is important to recognize that Fourier transforms and inverse Fourier transfers cancel one another. Why is this important? Why would a programmer ever construct an `fft` followed by an `ifft`? They wouldn't—intentionally. But with a highly abstracted set of library stream transformers, it's not always clear what will end up composed together.

In fact, when considered as an integral part of the design, algebraic rewrite rules enable us to write libraries in a simpler and more composable manner. For example, in WaveScript's signal processing library all filters take their input in the time domain, even if they operate in the frequency domain. A lowpass filter first applies an `fft` to its input, then the filter, and finally an `ifft` on its output. This maximizes composability, and does not impact performance. If two of these filters are composed together, the `fft`/`ifft` in the middle will cancel out. Without rewrite rules, we would be forced to complicate the interfaces.

## 6.4 Implementing Rewrites

A classic problem in rewrite systems is the order in which rules are applied. Applying one rewrite rule may preclude applying another. We make no attempt at an optimal solution to this problem. We use a simple approach; we apply rewrites to an abstract syntax tree from root to leaves and from left-to right, repeating the process until no change occurs in the program.

A key issue in our implementation is at what stage in the compiler we apply the rewrite rules. Functions like `rewindow` are defined directly in the language, so if the interpret-

```
// Before the original interpret/reify pass
When (rewrites-enabled )
   // Turn special functions into primitives:
   runpass( hide-special-libfuns )
   Extend-primitive-table(special-libfuns )
   runpass( interpret-reify )
   // Groom program and do rewrites:
   runpass( inline-let-bindings )
   run-until-fixation( apply-rewrite-rules )
   // Reinsert code for special functions:
   runpass( reveal-special-libfuns )
   Restore-primitive-table()
// Continue normal compilation....
```

Figure 6-2: Pseudo-code for the portion of the compiler that applies rewrite optimizations.

and-reify pass inlines their definitions to produce a stream graph, then rewrite rules will no longer apply. On the other hand, before interpret-and-reify occurs, the code is too abstracted to catch rewrites by simple syntactic pattern matching.

Our solution to this dilemma is depicted in Figure 6-2. We simply apply interpret-and-reify *twice*. The first time, we hide the top-level definitions of any "special" functions whose names occur in rewrite rules (`rewindow`, `fft`, etc), and treat them instead as primitives. Next we eliminate unnecessary variable bindings so that we can pattern match directly against nested compositions of special functions. Finally, we perform the rewrites, reinsert the definitions for special functions, and re-execute interpret-and-reify, which yields a proper stream graph of `iterates` and `merges`.

In summary, we have presented several forms of optimization applied by the WaveScript compiler. Rewrite rules simplify the program at a high level. Profiling and partitioning (which is described in Chapter 7) set the stage for our stream graph optimizations. Each of these optimizations is applicable to the applications we will examine in Chapter 8.

# Chapter 7

# Partitioning Algorithms

In this section, we describe WaveScript's algorithms to search for appropriate partitionings of the stream graph. We consider a directed acyclic graph (DAG) whose vertices are stream operators and whose edges are streams, with edge weights representing bandwidth and vertex weights representing CPU utilization or memory footprint. We only include vertices that can move across the node-server partition; i.e., the movable subgraph between the inner-most pinned operators in either partition. The server is assumed to have infinite computational power compared to the embedded nodes, which is a close approximation of reality.

The partitioning problem is to find a cut of the graph such that vertices on one side of the cut reside on the nodes and vertices on the other side reside on the server. The bandwidth of a given cut is measured as the sum of the bandwidths of the edges in the cut. An example problem is shown in Figure 7-1.

Unfortunately, existing tools for graph partitioning are not a good fit for this problem. Tools like METIS [33] or Zoltan [12,21] are designed for partitioning large scientific codes for parallel simulation. These are heuristic solutions that generally seek to create a fixed number of balanced graph partitions while minimizing cut edges. Newer tools like Zoltan support unbalanced partitions, but with a specified ratios, not allowing unlimited and unspecified capacity to the server partition. Further, they expect a single weight on each edge and each vertex. They cannot support a situation where the cost of a vertex *changes* depending on the partition is it placed in. This is the situation we're faced with: diverse hardware platforms that not only have varying capacities, but for which the *relative* cost of operators varies (for example, due to a missing floating point unit).

We may also consider traditional task scheduling algorithms as a candidate solution to our partitioning problem. These algorithms assign a directed graph of tasks to processors, attempting to minimize the total execution time. The most popular heuristics for this class of problem are variants of *list scheduling*, where tasks are prioritized according to some metric and then added one at a time to the working schedule. But there are three major differences between this classic problem and our own. First, task-scheduling does not directly fit the nondeterministic dataflow model, as no conditional control flow is allowed at the task level—all tasks execute exactly once. Second, task-scheduling is not designed for vastly unequal node capabilities. Finally, schedule length is not the appropriate metric for streaming systems. Schedule length would optimize for *latency*: how fast can the system process one data element. Rather, we wish to optimize for throughput, which is akin to scheduling for a task-graph repeated ad infinitum.

Thus we have developed a different approach. Our technique first preprocesses the graph to reduce the partition search space. Then it constructs a problem formulation based on the desired objective function and calls an external ILP solver. By default, WaveScript currently uses the minimum-cost cut subject to not exceeding the CPU resources of the embedded node or the network capacity of the channel. Cost here is defined as a linear combination of CPU and network usage (which can be a proxy for energy usage). For each platform we set four numbers: the two resource limits, and coefficients $\alpha,\beta$ to trade off the two minimization objectives. The user may also override these quantities to direct the optimization process.

## 7.1 Preprocessing

The graph preprocessing step precedes the actual partitioning step. The goal of the preprocessing step is to eliminate edges that could never be viable cut-points. Consider an operator $u$ that feeds another operator $v$ such that the bandwidth from $v$ is the same or higher than the bandwidth on the output stream from $u$. A partition with a cut-point on the

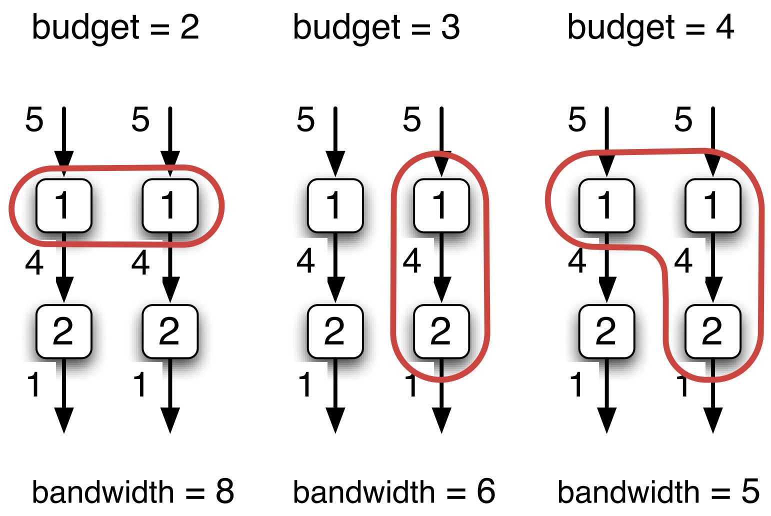


Figure 7-1: Simple motivating example. Vertices are labeled with CPU consumed, edges with bandwidth. The optimal mote partition is selected in red. This partitioning can change unpredictably, for example between a horizontal and vertical partitioning, with only a small change in the CPU budget.

*v*'s output stream can always be improved by moving the cut-point to the stream $u \to v$; the bandwidth does not increase, but the load on the embedded node decreases (*v* moves to the server). Thus, any operator that is data-expanding or data-neutral may be merged with its downstream operator(s) for the purposes of the partitioning algorithm, reducing the search space without eliminating optimal solutions.

One implementation issue that arises is whether preprocessing should be accomplished by actually generating a simplified graph and using that throughout the rest of the compilation, or rather constructing a temporary simplified version, performing the partitioning phase of the compiler, and then mapping the results back to the non-simplified graph. The latter can involve duplicated executions of portions of the compiler.

Presently we instead do the former; we implement preprocessing using WaveScript's existing framework for merging operators in a source-to-source transform. This keeps the compiler simple, and avoids the necessity of repeated compilation steps, and of extra metadata for tracking the relationship between the original operators and the merged operators (for the purpose of partitioning). This approach has a couple drawbacks, however. Individual operator's work functions—and therefore TinyOS tasks—will increase in size due to merging. This is addressed in Section 5.5.3, where we describe a method for subdividing tasks to improve multitasking. Second, for parts of the program that will end up on a multi-core machine, the additional merging of operators in the preprocessing step can reduce the available parallelism.

## 7.2 Optimal Partitionings

It is well-known that optimal graph partitioning is NP-complete [24]. Despite the intrinsic difficulty of the problem, the problem proves tractable for the graphs seen in realistic applications. Our pre-processing heuristic reduces the problem size enough to allow an ILP solver to solve it exactly within a few seconds to minutes.

### 7.2.1 Integer Linear Programming (ILP)

Let $G = (V, E)$ be the directed acyclic graph (DAG) of stream operators. For all $v \in V$, the compute cost is given by $c_v > 0$ and the communication (radio) cost is given by $r_{uv}$ for all edges $(u, v) \in E$. One might think of the compute cost in units of MHz (megahertz of CPU required to process a sample and keep up with the sampling rate), and the bandwidth cost in kilobits/s consumed by the data going over the radio. Adding memory-footprint constraints for both RAM usage (assuming static allocation) and for code storage is straightforward in this formulation. We can also use mean or peak load for each of these costs (we can compute both from our profiles.) Because our applications have predictable rates, we use mean load here, but have also experimented with partitioning for peak loads, which might be more appropriate in applications characterized by bursty rates.

The DAG $G$ contains a set of terminal *source* vertices $S$, and *sink* vertices $T$, that have no inward and outward edges, respectively, and where $S, T \subset V$. As noted above, we construct $G$ from the original operator graph such that these boundary vertices are pinned—all the sources must remain on the embedded node; all sinks on the server; and all of the non-boundary nodes in the graph are movable. Recall that the partitioning problem is to find a single cut of the $G$ that assigns some vertices to the nodes and some to the server. The graph represents the server and just one node, but the vertices that are assigned to the node are executed by all the nodes in the system.

We encode a partitioning using a set of indicator variables $f_v \in \{0,1\}$ for all $v$ in $V$. If $f_v = 1$, then operator $v$ resides on the node; otherwise, it resides on the server. The pinning constraints are:

$$\begin{aligned} &(\forall u \in S)\ f_u = 1 \\ &(\forall v \in T)\ f_v = 0 \\ &(\forall v)\ f_v \in \{0,1\}\ . \end{aligned} \tag{7.1}$$

Next, we constrain the sum of node CPU costs to be less than some total budget $C$.

$$cpu \leq C \quad \texttt{where} \quad cpu = \sum_{v \in V} f_v c_v \tag{7.2}$$

A simple expression for the total cut bandwidth is $\sum_{(u,v) \in E} (f_u - f_v)^2 r_{uv}$. (Because $f_v \in \{0,1\}$, the square evaluates to 1 when the edge $(u,v)$ is cut and to 0 if it is not; $|f_u - f_v|$ gives the same values.) However, we prefer to formulate the integer programming problem as one with a linear rather than quadratic objective function, so that standard ILP techniques can be used to solve the problem.

We can convert the quadratic objective function to a linear one by introducing two new variables per edge, $e_{uv}$ and $e'_{uv}$, which are subject to the following constraints:

$$\begin{aligned} &(\forall (u,v) \in E)\ e_{uv} \geq 0 \\ &(\forall (u,v) \in E)\ e'_{uv} \geq 0 \\ &(\forall (u,v) \in E)\ f_u - f_v + e_{uv} \geq 0 \\ &(\forall (u,v) \in E)\ f_v - f_u + e'_{uv} \geq 0\ . \end{aligned} \tag{7.3}$$

The intuition here is that when the edge $(u,v)$ is not cut (i.e., $u$ and $v$ are in the same partition), we would like $e_{uv}$ and $e'_{uv}$ to both be zero. When $u$ and $v$ are in different partitions, we would like a non-zero cost to be associated with that edge; the constraints above ensure that the cost is at least 1 unit, because $f_u - f_v$ is -1 when $u$ is on the server and $v$ on the embedded node. These observations allow us to formulate the bandwidth of the cut, cap that bandwidth, and define the objective function in terms of both CPU and network load.

$$net < N \quad \texttt{where} \quad net = \left( \sum_{(u,v)\in E} (e_{uv} + e'_{uv}) r_{uv} \right) \tag{7.4}$$

$$\texttt{objective}: \quad \min(\alpha \;\; cpu + \beta \;\; net) \tag{7.5}$$

Any optimal solution of (7.5) subject to (7.1), (7.2), (7.3), and (7.4) will have $e_{uv} + e'_{uv}$ equal to 1 if the edge is cut and to 0 otherwise. Thus, we have shown how to express our partitioning problem as an integer programming problem with a linear objective function, $2|E| + |V|$ variables (only $|V|$ of which are explicitly constrained to be integers), and at most $4|E| + |V| + 1$ equality or inequality constraints.

We could use a standard ILP solver on the formulation described above, but a further improvement is possible if we restrict the data flow to only one direction, from node to server. This restriction is not particularly restrictive in practice because most of the flow is in fact in that direction. On the positive side, the restriction reduces the size of the partitioning problem, which speeds up its solution.

Another set of constraints now apply:

$$(\forall (u, v) \in E) \; f_u - f_v \geq 0 \tag{7.6}$$

With this constraint, the network load quantity simplifies:

$$net = \left( \sum_{(u,v)\in E} (f_u - f_v) r_{uv} \right) . \tag{7.7}$$

This formulation eliminates the $e_{uv}$ and $e'_{uv}$ variables, simplifying the optimization problem. We now have only $|V|$ variables and at most $|E| + |V| + 1$ constraints. We have chosen this restricted formulation for our current, prototype implementation, primarily because the per-platform code generators don't yet support arbitrary back-and-forth communication between node and server. We use an off-the-shelf integer programming solver, `lp_solve`[1],

[1] `lp_solve` was developed by Michel Berkelaar, Kjell Eikland, and Peter Notebaert. It is available from `http://lpsolve.sourceforge.net`. It uses branch-and-bound to solve integer-constrained problems, like ours, and the Simplex algorithm to solve linear programming problems.

to minimize (7.7) subject to (7.1) and (7.2).

We note that the restriction of unidirectional data flow does preclude cases when sinks are pinned to embedded nodes (e.g., actuators or feedback in the signal processing). It also prevents a good partition when a high-bandwidth stream is merged with a heavily-processed stream. In the latter case, the merging must be done on the node due to the high-bandwidth stream, but the expensive processing of the other stream should be performed on the server. In our applications so far, we have found our restriction to be a good compromise between provable optimality and speed of finding a partition.

## 7.3 Data Rate as a Free Variable

It is possible that the partitioning algorithm will not be able to find a cut that satisfies all of the constraints (i.e., there may be no way to “fit” the program on the embedded notes.) In this situation we wish to find the maximum data rate that *will* support a viable partitioning. The algorithm given above cannot directly treat data rate as a free variable. Even if CPU and network load varied linearly with data rate, the resulting optimization problem would be non-linear. However, it turns out to be inexpensive to perform the search over data-rates as an *outer loop* that on each iteration calls the partitioning algorithm.

This is because in most applications, CPU and network load increase monotonically with input data rate. If there is a viable partition when scaling input data rates by a factor $X$, then any factor $Y < X$ will also have a viable partitioning. Thus WaveScript simply does a binary search over data rates to find the maximum rate at which the partitioning algorithm returns a valid partition. As long as we are not over-saturating the network such that sending fewer packets actually result in more data being successfully received, this maximum sustainable rate will be the best rate to pick to maximize outputs (throughput) of the data flow graph. We will re-examine this assumption in Section 8.3.

## 7.4 Dynamic Programming when the DAG is a Tree

A branch-and-bound integer programming solver like `lp_solve` finds an optimal solution to our partitioning problem, but its running time can be exponential in the size of the problem. Branch-and-bound solvers find a sequence of progressively better solutions and a sequence of progressively tighter lower bounds, until eventually they find an optimal solution. Therefore, one way to address the potentially-exponential running time is to halt the solver when it has found a solution that is provably good but not necessarily optimal. (We note that for our problem, the solution that assigns all the non-sources to the server is feasible, so merely finding a solution is not difficult.)

When $G$ is a tree and when the $c_v$'s are bounded integers, the unidirectional partitioning problem can be solved efficiently (in polynomial time) and optimally. This solution is "approximate" only in the sense that CPU coefficients are rounded to integers. Given that the profiler can only provide execution times up to a certain granularity, discretization of execution times need not lose information. (The scale can be arbitrarily fine.)

Given a subgraph $H \subseteq G$, a *partition plan* $f$ is a unidirectional assignment of $f_v$ for every $v \in H$ (that is, an assignment satisfying (7.6)). The cost of the plan is $c(f) = \sum_{v \in H} f_v c_v$, bandwidth of the plan is $r(f) = \sum_{(u,v) \in H} (f_u - f_v) r_{uv}$, and the external bandwidth of the plan is $x(f) = \sum_{(u,v) \in H, u \in H, v \notin H} f_u r_{uv}$.

Consider a subgraph $H$ consisting of the union of the subtrees rooted at all the children $u_1, \ldots, u_k$ of some vertex $v$. Any plan $f$ for $H$ in which at least one of $u_1, \ldots, u_k$ is assigned to the server can only be extended to $v$ by also assigning $v$ to the server. The cost of the extended plan $f'$ is the cost of $f$, its bandwidth is $r(f') = r(f) + \sum_{i=1}^{k} f_{u_i} r_{u_i v} = r(f) + x(f)$, and its external bandwidth is 0. Any plan $f$ for which all the $u_i$'s are assigned to the node (there is only one such plan) can be extended by assigning $v$ to either the server or the node. If we assign $v$ to the server, the cost is the cost of $f$, the bandwidth is $r(f') = x(f) = r(f) + x(f)$, and the external bandwidth is 0. If we extend the plan by assigning $v$ to the node, the cost of the new plan is $c(f) + c_v$, the bandwidth is 0, and the external bandwidth is $r_{vw}$ where $w$ is the parent of $v$.

The algorithm works by recursively traversing the tree starting from the root. At every

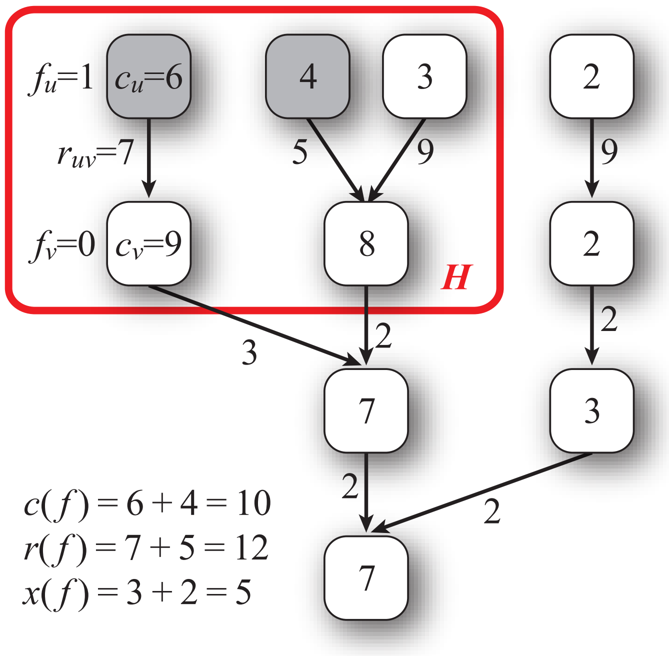


Figure 7-2: The notation for the dynamic programming algorithm.

vertex $v$, we first construct a set of plans for each subtree rooted at a child $u_i$ of $v$. The set contains for each possible cost (there are $c + 1$ such values) a plan $f$ with the minimal possible total bandwidth $r(f) + x(f)$.

Next, the algorithm merges these sets to one set for the union of the subtrees rooted at the children $u_1, \ldots, u_k$ of $v$. We initialize this set to that of $u_1$. We extend the set of plans for union of the subtrees rooted at $u_1, \ldots, u_{i-1}$ to the union of $u_1, \ldots, u_{i-1}, u_i$ as follows. For every pair of plans, one for $u_1, \ldots, u_{i-1}$ and the other for $u$, we compute the cost and bandwidths for the merged plans. If the new set does not contain a plan with this set, we put it in the set. If it does but the existing plan has lower total bandwidth, we discard the new merged plan. If the new merged plan has better total bandwidth than the existing candidate with the same cost, we replace the existing one with the new one.

Finally, we extend the set of plans to include $v$, using the rules given above. This may increase the number of plans in the set by one, but the number of plans remains bounded by $c + 1$. As we consider plans, we always discard ones with total cost greater than $c$, our upper bound. The algorithm is correct because all of the sets that it constructs contain a plan that can be extended to an optimal partitioning. This property holds because for subgraphs $H$ that are unions of subtrees rooted at the children of one vertex, all the plans with the same cost and same total bandwidth are equivalent; if one can be extended to an optimal partitioning then so can the others. The complexity of this algorithm is $O(c^2|V|)$; the bound is somewhat loose but it is beyond the scope of this paper to sharpen it further. For profile-driven CPU-load estimates, $c$ around 1000 gives sufficient accuracy and a reasonable running times.

### 7.4.1 Summary

In this chapter we have examined the algorithm, based on integer linear programming, that WaveScript uses to partition stream programs into embedded node and server components. In Chapter 8 we will evaluate the use of this algorithm to partition and execute our applications on various platforms.

# Chapter 8

# Experimental Results

Evaluating a new programming language is difficult. Until the language has had substantial use for a period of time, it lacks large scale benchmarks. Microbenchmarks, on the other hand, can help when evaluating specific implementation characteristics. But when used for evaluating the efficacy of program optimizations, they risk becoming contrived.

Thus we chose to evaluate WaveScript "in the field" by using it to developing a substantial sensor network application for localizing animals in the wild. First, we compare our implementation to a previous (partial) implementation of the system written in C by different authors. The WaveScript implementation outperforms its C counterpart—with significant results for the sensor network's real-time capabilities. Second, in Section 8.3, we showcase our compiler optimizations in the context of this application, explaining their effect and evaluating their effectiveness.

Subsequently, in Section 8, we turn to evaluating the WaveScript program partitioning facility. While we don't have an in-the-field evaluation of these components, we use two prototype sensing applications—acoustic speaker detection and EEG based siezure detection—to evaluate the system.

Finally, in Section 8.4 we turn to the issue of parallelism, which we evaluate in the context of a fourth application: a computer vision application that highlights the ability of WaveScript to capture a variety of forms of paralleism, including parallel computations over matrices in addition to pipeline and task parallelism between streams, and data parallelism between stream elements.

## 8.1 Marmot: Comparing against handwritten C

A year previous to our own test deployment of the distributed marmot detector, a different group of programmers implemented the same algorithms (in C) under similar conditions in a similar time-frame. This provides a natural point of comparison for our own WaveScript implementation. Because the WaveScript implementation surpasses the performance of the C implementation, we were able to run both the detector and the direction-of-arrival (DOA) algorithm on the VoxNet nodes (our custom sensing platform) in real-time—something the previous implementation did not accomplish (due to limited CPU).

Table 8.1 shows results for both the continuously running detector, and the occasionally running DOA algorithm (which is invoked when a detection occurs). The detector results are measured in percentage CPU consumed when running continuously on an VoxNet node and processing audio data from four microphone channels at 44.1 KHz (quantified by averaging 20 *top* measurements over a second interval). DOA results are measured in seconds required to process raw data from a single detection. Along with CPU cycles, memory is a scarce resource in embedded applications. The WaveScript version reduced the memory footprint of the marmot application by 50% relative to the original hand-coded version. (However, even the original version used only 20% of VoxNet's 64 MB RAM.) GC performance of MLton was excellent. When running both detection and DOA computations, only 1.2% of time was spent in collection, with maximum pauses of 4ms—more than adequate for our application. Collector overhead is low for this class of streaming application, because they primarily manipulate arrays of scalars, and hence allocate large quantities of memory but introduce relatively few pointers.  Table 8.2 lists the size of the source-code for the Detector and DOA components, discounting blank lines and comments. Both versions of the application depend on thousands of lines of library code and other utilities. Lacking a clear boundary to the application, we chose to count only the code used in the implementation of the core algorithms, resulting in modest numbers.

The ability to run the DOA algorithm directly on VoxNet results in a large reduction in data sent over the network—800 bytes for direction-of-arrival probabilities vs. at least 32KB for the raw data corresponding to a detection. The reduced time in network trans-

Table 8.1: Performance of WaveScript marmot application vs. handwritten C implementation. Units are percentage CPU usage, number of seconds, or speedup factor.

| | C | WaveScript | Speedup |
|---|---|---|---|
| VoxNet DOA | 3.00s | 2.18s | 1.38 |
| VoxNet Detect | 87.9% | 56.5% | 1.56 |

mission offsets the time spent running DOA on the VoxNet node (which is much slower than a laptop), and can result in *lower* overall response latencies. The extra processing capacity freed up by our implementation was also used for other services, such as continuously archiving all raw data to the internal flash storage, a practical necessity that was not possible in previous attempts.

Our original goal was to demonstrate the ease of programming applications in a high-level domain-specific language. In fact, we were quite surprised by our performance advantage. We implemented the same algorithm in roughly the same way as previous authors. Neither of the implementations evaluated here represent intensively hand-optimized code. A significant fraction of the application was developed in the field during a ten-day trip to Colorado. Indeed, because of the need for on-the-fly development, programmer effort is the bottleneck in many sensor network applications. This is in contrast with many embedded, or high-performance scientific computing applications, where performance is often worth any price. Therefore, languages that are both high-level *and* allow good performance are especially desirable.

After the deployment we investigated the cause of the performance advantage. We found no significant difference in the efficiency of the hottest spots in the application (for example, the tight loop that searches through possible angles of arrival). However, the C implementation was significantly less efficient at handling data, being constrained by the layered abstraction of the EmStar framework. It spent 26% percent of execution time and 89% percentage of memory in data acquisition (vs. 11% and 48% for WaveScript). In short, the application code we wrote in WaveScript was *as* efficient as hand-coded C, but by leveraging the WaveScript platform's vertical integration of data acquisition, management of signal data, and communication, overall system performance improved.

Table 8.2: Non-whitespace, non-comment lines of code for WaveScript and C versions of the core localization algorithm.

| | LOC/WaveScript | LOC/C |
|---|---|---|
| Detector | 92 | 252 |
| DOA | 124 | 239 |

## 8.2 Effects of Optimization on Marmot Application

Here we relate the optimizations described in Section 6 to our marmot application case study. One thing to bear in mind is that there are *multiple* relevant modes of operation for this application. A given stage of processing may execute on the VoxNet node, the laptop base station, or offline on a large server. Both on the laptop and offline, utilizing multiple processor cores is important.

**Rewrite-rules:** As discussed in Section 6.3, many of our signal processing operations take their input in the time domain, but convert to the frequency domain to perform processing. An example of this can be seen in the `bandpass` library routine called from the `marmotScores` function in Section 3 (part of the detector phase). Notice that the `marmotScores` function is another example; it also converts to the frequency domain to perform the PSD. The rewrite-rules will eliminate all redundant conversions to and from the frequency domain, with a 4.39× speedup for the detector phase in the MLton backend and 2.96× speedup in the C++ backend.

**Fusion and Batching:** The fusion optimizations described in Section 6.2 are relevant to the C++ backend, which has a higher per-operator overhead. Fusion is most advantageous when many lightweight operators are involved, or when small data elements are passed at a high rate. Because the marmot application involves a relatively small number of operators, and because they pass data in Sigsegs, the benefits of fusion optimization are modest. (For the same reason, the batching optimizations performed by the compiler, while invaluable in many cases, provide no benefit to the marmot application.)

The detector phase of the application speeds up by 7%, and the DOA phase by 2.7%. The FuseDOA phase does not benefit.

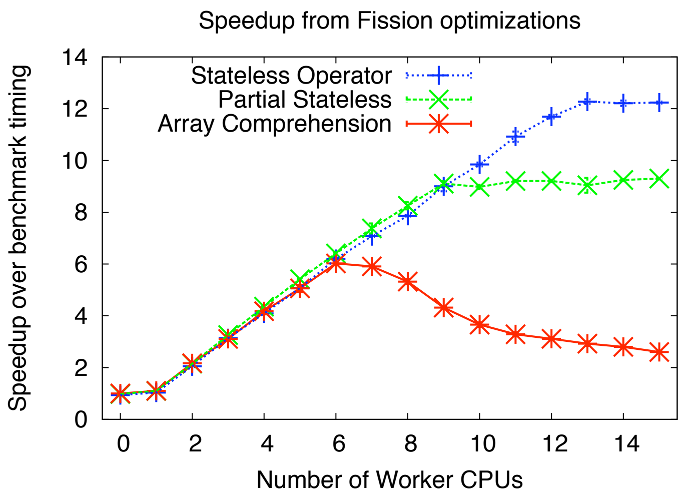


Figure 8-1: Parallel speedups achieved by applying fission optimizations to the DOA phase of the marmot application.

**Fission and Parallelization:** Offline processing has intrinsic parallelism because it applies the first and second phases of the application (detector and DOA) to many data streams in parallel (simulating multiple nodes). To squeeze parallelism out of the individual marmot phases, we rely on our fission optimizations from Section 6.2.

To evaluate our fission optimizations, we applied each of them to the DOA phase of the marmot application and measured their performance on a commodity Linux server. Our test platform is a $4 \times 4$ motherboard with 4 quad-core AMD Barcelona processors and 8 GB of RAM, running Linux 2.6.23. In our parallel tests, we control the number of CPUs actually used in software. We used the Hoard memory allocator to avoid false sharing of cache lines.

Fission can be applied to the DOA phase in two ways: by duplicating stateless operators, and by using array comprehension to parallelize a loop. Figure 8-1 shows the parallel speedup gained by applying each of these optimizations to the DOA phase of the marmot application. In this graph, both flavors of fission optimization are presented to show speedup relative to a single-threaded version. Each data point shows the mean and 95% confidence intervals computed from 5 trials at that number of worker CPUs. The point at '0' worker CPUs is single-threaded; the point at '1' worker CPU places the workload operator on a different CPU from the rest of the workflow (e.g., the I/O, split, join, etc).

The greatest gain, a speedup of 12× is achieved from parallelizing stateless operators. In our application, the entire DOA phase of the workflow is stateless, meaning that the whole phase can be duplicated to achieve parallelism. As described in Section 6.2, a `map` operator or a sequence of `map` operators is replaced by a `split`→`join` sequence that delivers tuples in round robin order to a set of duplicated worker operators, and subsequently joins them in the correct order. Running this on our 16 core test machine, we see near-linear speedup up to 13 cores, where performance levels off. This level is the point at which the serial components of the plan become the bottleneck, and are unable to provide additional work to the pool of threads.

Array comprehension parallelization yields a lesser, but still significant maximum speedup of 6×. This case is more complex because fission by array comprehension applies to only a portion of the DOA phase. The DOA computation consists of a preparation phase that computes some intermediate results, followed by a work phase that exhaustively tests hypothetical angle values. This structure limits the maximum possible speedup from this optimization. As a control, the "Partial Stateless" curve designates the speedup achieved by restricting the stateless operator fission to the phase duplicated by the array comprehension. From the graph we see that the parallel benefit is maximized when distributing the work loop to 6 worker cores; beyond that point the additional overhead of transferring between cores (e.g., queueing and copying overhead) diminishes the benefit. The appropriate number of cores to use is a function of the size of the work loop and the expected copying and queueing overhead.

Optimizing for latency is often important for real time responses and for building feedback systems. Although the stateless operator optimization achieves higher throughput through pipelining, it will never reduce the latency of an individual tuple passing through the system. However, array comprehension *can* substantially reduce the latency of a particular tuple by splitting up a loop among several cores and processing these smaller chunks in parallel. In our experiments we found that the array comprehension optimizations reduced the average latency of tuples in our test application from 30 ms to 24 ms, a 20% reduction.

## 8.3 Evaluation: Partitioning

In this section we evaluate the WaveScript system on two applications, seizure onset detection and speech detection, that were mentioned briefly in Chapter 3.1 and will be described in more detail below. We focus on two key questions:

1. Can WaveScript efficiently select the best partitioning for a real application, across a range of hardware devices and data rates?
2. In an overload situation, can WaveScript effectively predict the effects of load-shedding and recommend a "good" partitioning?

### 8.3.1 EEG Application: Seizure Onset Detection

We used WaveScript to implement a patient-specific seizure onset detection algorithm [62]. The application was previously implemented in C++, but by porting it to WaveScript we enabled its embedded/distributed operation, while reducing the amount of code by a factor of four without loss of performance. This application will later be used to evaluate the program-partitioning capabilities of WaveScript.

The algorithm is designed to be used in a system for detecting seizures outside a clinical environment. In this application, a user would wear a monitoring cap that typically consists of 16 to 22 channels. Data from the cap is processed by a low-power portable device.

The algorithm we employ [63] samples data from 22 channels at 256 samples per second. Each sample is 16-bits wide. For each channel, we divide the stream into 2 second windows. When a seizure occurs, oscillatory waves below 20 Hz appear in the EEG signal. To extract these patterns, the algorithm looks for energy in certain frequency bands.

To extract the energy information, we first filter each channel by using a polyphase wavelet decomposition. We use a repeated filtering structure to perform the decomposition. The filtering structure first extracts the odd and even portions of the signal, passes each signal through a 4-tap FIR filter, then adds the two signals together. Depending on the values of the coefficients in the filter, we either perform a low-pass or high-pass filtering operation. This structure is cascaded through 7-levels, with the high frequency signals from the last three levels used to compute the energy in those signals. Note that at each level, the amount of data is halved.

As a final step, all features from all channels, 66 in total, are combined into a single vector which is input into a patient-specific support vector machine (SVM). The SVM detects whether or not each window contains epileptiform activity. After three consecutive positive windows have been detected, a seizure is declared.

There are multiple places where WaveScript can partition this algorithm. If the entire application fits on the embedded node, then the data stream is reduced to only a feature vector—an enormous data reduction. But data is also reduced by each stage of processing on each channel, offering many intermediate points which are profitable to consider.

## 8.3.2 Partitioning the EEG Application

Our EEG application provides an opportunity to explore the scaling capability of our partitioning method. In particular, we look at our worst case scenario—partitioning all 22-channels (1412 operators). As the CPU budget increases, the optimal strategy for bandwidth reduction is to move more channels to the nodes. On our lower-power platforms, not all the channels can be processed on one node. The graph in Figure 8-5(a) shows partitioning results only for the *first* of 22 channels, where we vary the data rate on the X axis and measure the number of operators that “fit” on different platforms. We ran `lp_solve` to derive a 22-channel partitioning 2100 times, linearly varying the data rate to cover everything from “nothing fits” to “everything fits easily”. As we reduce data rate (moving to the left), more operators can fit within the CPU bounds on the node (moving up). The sloping lines show that every stage of processing yields data reductions.

The distribution of resulting execution times are depicted as two CDFs in Figure 8-2, where the x axis shows execution time in seconds, on a log scale. The top curve in Figure 8-2 shows that even for this large graph, `lp_solve` always found the optimal solution in under 90 seconds. The typical case was much better: 95 percent of the executions reached optimality in under 10 seconds. While this shows that an optimal solution is typically discovered in a reasonable length of time, that solution is not necessarily *known* to be optimal. If the solver is used to prove optimality, both worst and typical case runtimes become

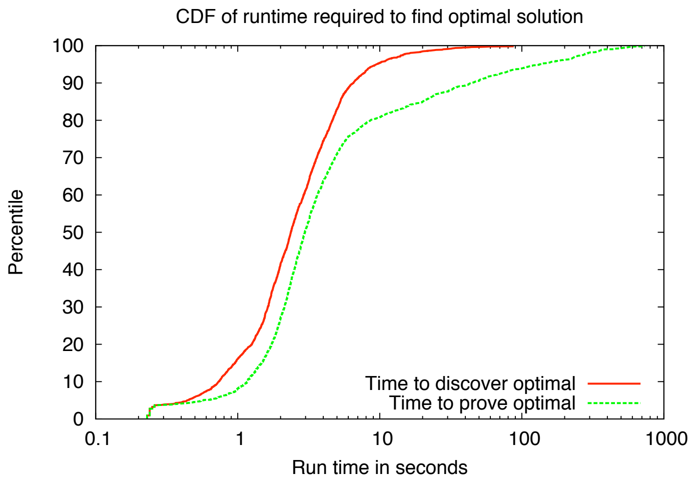


Figure 8-2: CDF of the time required for `lp_solve` to reach an optimal partitioning for the full EEG application (1412 operators), invoked 2100 times with data rates. The higher curve shows the execution time at which an optimal solution was found, while the lower curve shows the execution time required to prove that the solution is optimal. Execution times are from a 3.2 GHz Intel Xeon.

much longer, as shown by the lower CDF curve (yet still under 12 minutes). To address this, we can use an approximate lower bound to establish a termination condition based on estimating how close we are to the optimal solution.

### 8.3.3 Speech Detection Application

We also used WaveScript to build a speech detection application that uses sampled audio to detect the presence of a person who is speaking near a sensor. The ultimate goal of such an application would be to perform speaker *identification* using a distributed network of microphones. For example, such a system could potentially be used to locate missing children in a museum by their voice, or to implement various security applications.

However, in our current work we are only concerned with speech *detection*, a precursor to the problem of speaker identification. In particular, our goal is to reduce the volume of data required to achieve speaker identification, by eliminating segments of data that probably do not contain speech and by summarizing the speech data through feature extraction.

Our implementation of speech detection and data reduction is based on Mel Frequency Cepstral Coefficients (MFCC), following the approach of prior work in the area [19]. Re-

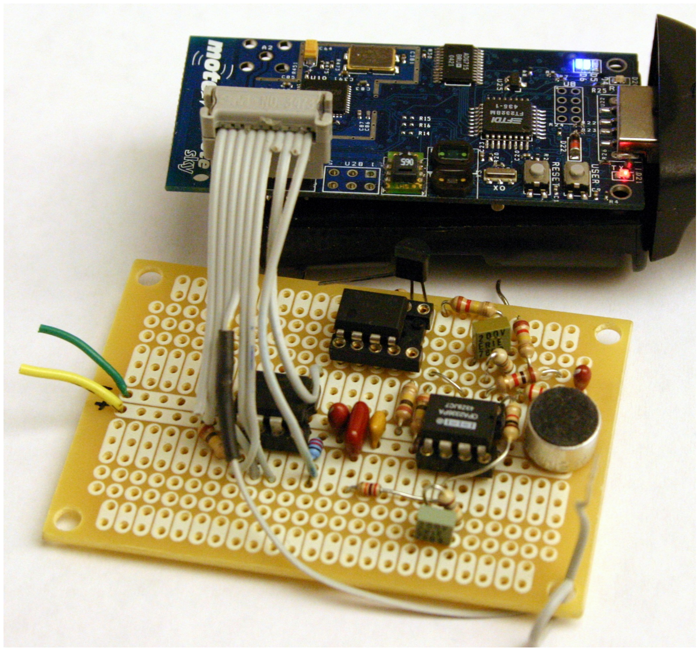


Figure 8-3: Custom audio board attached to a TMote Sky.

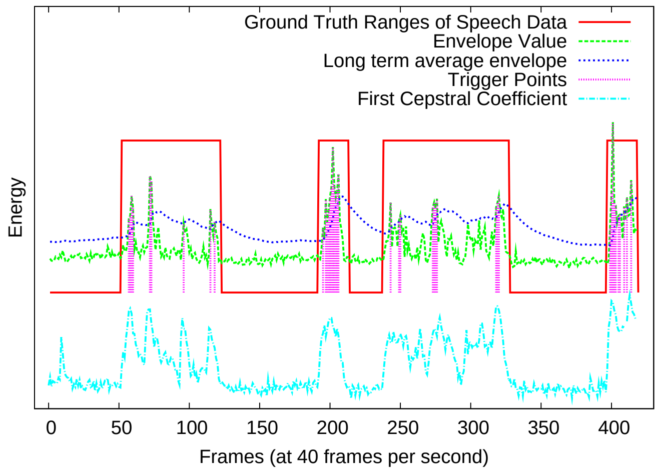


Figure 8-4: Performance of the envelope detector, in comparison to the ground truth presence of speech and the first cepstral coefficient.

cent work by Martin, *et al.* has shown that clustering analysis of MFCCs can be used to implement robust speech detection [43]. Another article by Saastamoinen, *et al.* describes an implementation of speaker identification on smartphones, based on applying learning algorithms to MFCC feature sets [58]. Based on this prior work, we chose to exercise our system using an implementation of MFCC feature extraction.

### Mel Frequency Cepstral Coefficients

Mel Frequency Cepstral Coefficients (MFCC) are the most commonly used features in speech recognition algorithms. The MFCC feature stream represents a significant data reduction relative to the raw data stream.

To compute MFCCs, we first compute the spectrum of the signal, and then summarize it using a bank of overlapping filters that approximates the resolution of human aural perception. By discarding some of the data that is less relevant to human perception, the output of the filter bank represents a 4X data reduction relative to the original raw data. We then convert this reduced-resolution spectrum from a linear to a log spectrum. Using a log spectrum makes it easier to separate convolutional components such as the excitation applied to the vocal tract and the impulse response of a reverberant environment, because transforms that are multiplicative in a linear spectrum are additive in a log spectrum.

Finally, we compute the MFCCs as the first 13 coefficients of the Discrete Cosine Transform (DCT) of this reduced log-spectrum. By analyzing the spectrum of a spectrum, the distribution of frequencies can be characterized at a variety of scales [19, 16]. The MFCC feature stream could then be input to a speaker identification algorithm [58], but we did not implement this in our current system.

### Tradeoffs in MFCC Extraction

One high level goal of WaveScript is to show that a complex application written in a single high level language can be efficiently and easily distributed across a network of devices and support many different platforms. As such, the MFCC application presents an interesting challenge because for sensors with very limited resources there appears to be no perfect solution; rather, using WaveScript the application designer can explore different trade-offs in application performance.

These trade-offs arise because this algorithm squeezes a resource-limited device between two insoluble problems: not only is the network capacity insufficient to forward all the raw data back to a central point, but the CPU resources are also insufficient to extract the MFCCs in real time. If the application has any partitioning that fits the resource constraints, then the goal of WaveScript is to select the best partition, for example, lowest cost in terms of energy. If the application does not *fit* at its ideal data rate, ultimately, some data will be dropped on some target platforms. The objective in this case is to find a partitioning that minimizes this loss and therefore maximizes the *throughput*: the amount of input data successfully processed rather than dropped at the node or in the network.

Of course, the designer may also use the profiling information to observe hot spots and change the algorithm. For example, in the MFCC application, initial profiling results led us to conclude that if we were forced to drop data, it may be beneficial to introduce a prefilter to help select *which* data is dropped. To experiment with this technique, we implemented a simple Finite Impulse Response (FIR) pre-emphasis filter to emphasize the high end of the spectrum, and an envelope detector to trigger on high-amplitude. We then use an Exponentially Weighted Moving Average (EWMA) to determine a long-term average envelope and then trigger the full computation pipeline only when the value of the envelope exceeds the average by more than a constant threshold factor $T_E$. Figure 8-4 shows the performance of this pre-filter stage, with $T_E = 1.5$, for a segment of audio data that has been hand-labeled to identify speech sounds.

### Implementing Audio Capture

Some platforms, such as the iPhone and embedded-Linux platforms (such as the Gumstix), provide a complete and reliable hardware and software audio capture mechanism. On other platforms, including both TMotes and J2ME phones, capturing audio is more challenging.

On TMotes, we used a custom-built audio board to acquire audio. The board uses an electret microphone, four opamp stages, a programmable-gain amplifier (PGA), and a 2.5 V voltage reference. One opamp stage is used to buffer the voltage reference, a second to buffer a 1.25 V voltage divider, and two are used as inverting amplifiers. The buffered 1.25 V is used to bias both the inverting amplifiers and the PGA; the inverting amplifiers can use an unbuffered divider, but the PGA configuration is non-inverting, so it requires a low-impedance reference. The referenced and buffered 2.5 V is used to power the electret microphone (which contains a simple amplifier). We have found that when the microphone was powered directly by the analog supply of the TMote, the audio board performed well when the mote was only acquiring audio, but was very noisy when the mote was communicating. The communication causes a slight modulation of the supply voltage, which gets amplified into significant noise. Using a separately regulated supply for the microphone removed this noise. The gain of the PGA is controlled by SPI bus interface, which allows the program running on the TMote to adjust the audio gain. The anti-aliasing filter is a simple RC filter; to better reject aliasing, the TMote samples at a high rate and applies a digital low-pass filter (filtering and decimating a 32 Ks/s stream down to 8 Ks/s works well). The amplified and filtered audio signal is presented to an ADC pin of the TMote's microcontroller, which has 12 bits of resolution. We use TinyOS 2.0 `ReadStream<uint16_t>` interface to the ADC, which uses double buffering to deliver arrays of samples to the application.

Phones naturally have built-in microphones and microphone amplifiers, but we have nonetheless encountered a number of problems using them as audio sensors. Many J2ME phones support the Mobile Media API (JSR-135), which may allow a program to record audio, video, and take photographs. Support for JSR-135 does not automatically imply support for audio or video recording or for taking snapshots. Even when audio recording is supported, using it is subject to the J2ME security policy which is installed by the phone's vendor (the manufacturer or a cellular carrier). In the phones that we have used (Nokia N80, N95, E61i), recording audio requires the user's permission every time an unsigned program starts. The JSR-135 API does not stream audio (or video); it records audio to a file or to an array, and most phones cannot record continuously. Even if the application stops recording and immediately starts another recording, there would be a gap in the audio. The audio or video is delivered as a media file; even if the file contains uncompressed audio samples, its headers must first be stripped by the program before the samples can be accessed.

We ran into a bug on the Nokia N80: after recording audio segments for about 20 minutes, the JVM would crash. Other Nokia phones with the same operating system (Symbian S60 3rd Edition) exhibited the same bug. We worked around this bug using a simple Python script that runs on the phone and accepts requests to record audio or take a photograph through a TCP connection, returning the captured data also via TCP. The J2ME program acquires audio by sending a request to this Python script, which can record indefinitely without crashing.

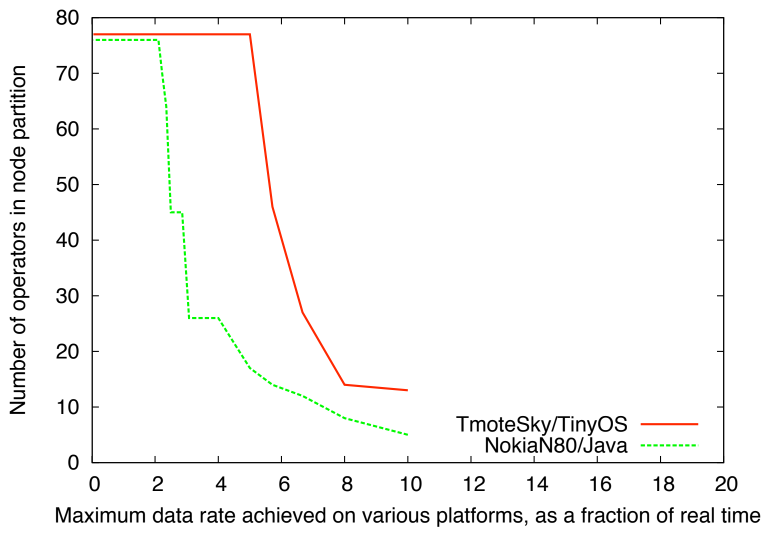
The J2ME partition of the WaveScript program uses TCP to stream partially processed results to the server. When the J2ME connects, the phone asks the user to choose an IP access point; we normally use a WiFi connection, but the user can also choose a cellular IP connection. With any of these communication methods, dependence on user interaction presents a practical barrier to using phones in an autonomous sensor network. Yet these software limitations are incidental rather than fundamental, and should not pose a long-term problem.

### 8.3.4 Partitioning the Speech Detection Application

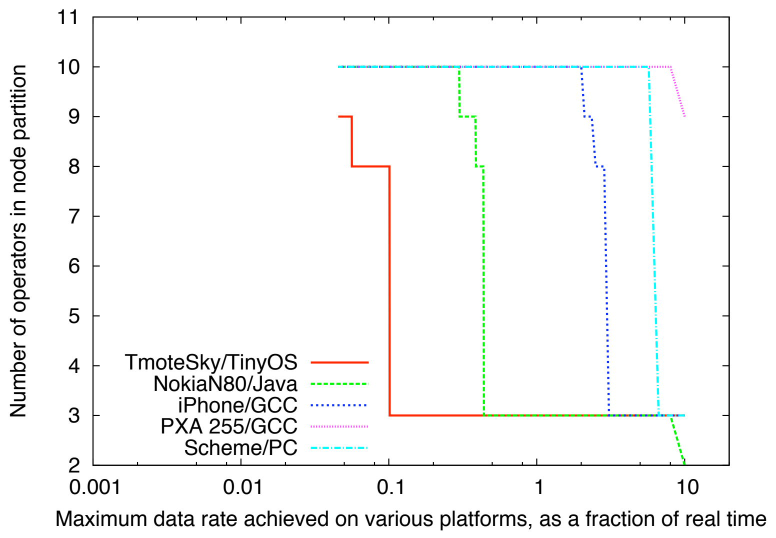
The speech detection application is a linear pipeline of only a dozen operators. Thus the optimization process for picking a cut point should be trivial—a brute force testing of all cut points will suffice. Nevertheless, this application's simplicity makes it easy to visualize and study, and the fact that the data rate it needs to process all data is unsustainable for TinyOS devices provides an opportunity to examine the other side of WaveScript's usage: what to do when the application doesn't fit.

In applying WaveScript to the development process for our speech detection application, we were able to quickly assess the performance on several different platforms. The results of the profiling stage are presented in Figure 6-1 (left) and Figure 8-6. The four columns in Figure 6-1 (left) show the profiling results for our four test platforms, in a graphical form color coded by the running time of the operator (red = longer, green = shorter).

Figure 8-6 is a more detailed visualization of the performance tradeoffs, showing only



(a)



(b)

Figure 8-5: Optimal partitioning for varying data rates (as a fraction of real time). X axis shows the selected data rate; Y axis shows number of operators in computed node partition. The EEG application (a) consists of identical processing pipelines for separate channels (only one channel pictured). Speech detection application are shown in (b)—note log scale on X axis.

the profiling results for TMote Sky (a TinyOS platform). In this figure, the *X* axis represents the linear pipeline of operators, and the *Y* axis represent profiling results. For each operator the vertical impulse represents the number of microseconds of CPU time consumed by that operator per frame (left scale), while the line represents the number of bytes per second output by that operator. It is easy to visualize the tradeoff between CPU cost and data rate. Each point on the X-axis represents a potential graph cut, where the sum of the red bars to the left provides the processing time per frame.

Thus, we see that the MFCC dataflow has multiple data-reducing steps. The algorithm must natively process 40 frames per second in real time, or one frame every 25 ms. The initial frame is 400 bytes; after applying the filter bank the frame data is reduced to 128 bytes, using 250 ms of processing time; after applying the DCT, the frame data is further reduced to 52 bytes, but using a total of 2 s of processing time. This structure means that although no split point can fit the application on the TMote at the full rate, we can achieve different CPU/bandwidth trade-offs by selecting different split points. Selecting a defunct partitioning can result in retrieving no data, and the best "working" partition provides 20 times more data than the worst. Figure 8-5(b) shows the output of the partitioner for five different platforms, as a function of different target data rates. This shows the predicted performance of our speech detection application on different platforms.

As expected, the TMote is the worst performing platform, with the Nokia N80 performing only about twice as fast—surprisingly poor performance given that the N80 has a 32-bit processor running at 55X the clock rate of the TMote. This is due to the poor performance of the JVM implementation. The 412 MHz iPhone platform using GCC performed 3X worse than the 400 MHz Gumstix-based Linux platform; we believe that this is due to the frequency scaling of the processing kicking in to conserve power.

We can also visualize the relative performance of different operators across different platforms. For each platform processing the complete operator graph, Figure 8-7 shows the fraction of time consumed by each operator. If the time required for each operator scaled linearly with the overall speed of the platform, all three lines would be identical. However, the plot clearly shows that the different capabilities of the platforms result in very different relative operator costs. For example, on the TMote, floating point operations, which are used heavily in the `cepstrals` operator, are particularly slow. This shows that a model that assumes the relative costs of operators are the same on all platforms would mis-estimate costs by over an order of magnitude.

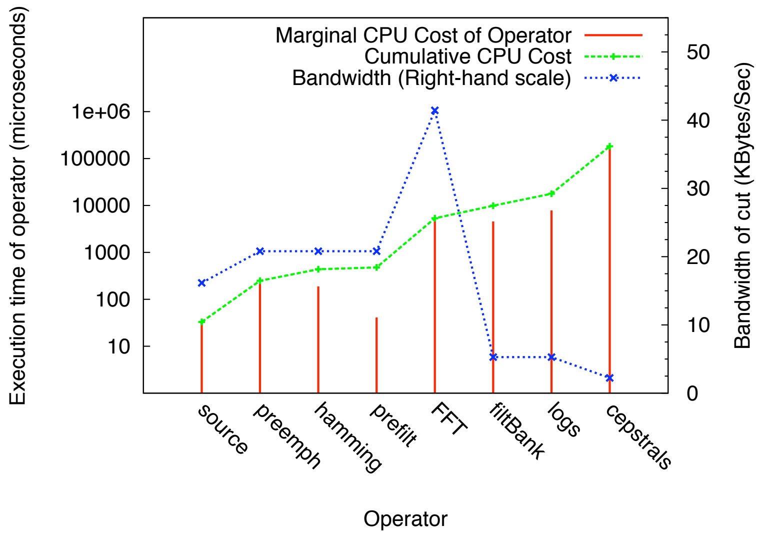


Figure 8-6: Data is reduced by processing, lowering bandwidth requirements, but increasing CPU requirements.

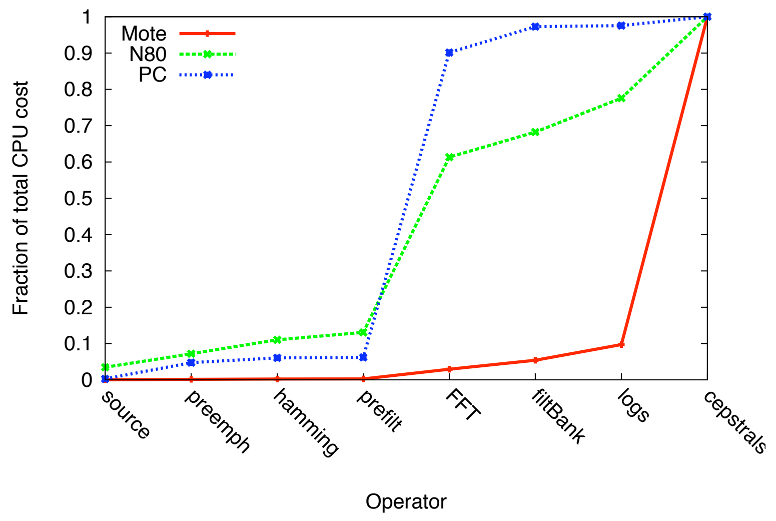


Figure 8-7: Normalized cumulative CPU usage for different platforms. Relative execution costs of operators vary greatly on the tested systems.

### 8.3.5 WaveScript Deployment

To validate the quality of the partitions selected by WaveScript, we deployed the speech detection application on a testbed of 20 TMote Sky nodes. We also used this deployment to validate the specific performance predictions that WaveScript makes using profiling data (e.g., if a combination of operators were predicted to use 15% CPU, did they?).

**Network Profiling**

The first step in deploying using WaveScript is to profile the network topology. It is important to note that simply changing the network size changes the available per-node bandwidth and thus requires re-profiling of the network and re-partitioning of the application. We run a portable WaveScript program that measures the goodput from each node in the network. This tool sends packets from all nodes at an identical rate, which gradually increases. For our 20 node testbed the resulting network profile is typical for TMote Sky devices: each node has a baseline packet drop rate that stays steady over a range of sending rates, and then at some drops off dramatically as the network becomes excessively congested. Our profiling tool takes as input a target reception rate (e.g. 90%), and returns a maximum send rate (in msgs/sec and bytes/sec) that the network can maintain.For the range of sending rates within this upper bound the assumption mentioned in 7.3 holds—attempting to send more data does not result in additional bytes of data received. Thus we are free to maximize the data rate within the upper bound provided by the network profiling tool, and thereby maximize total application throughput. This enables us to use binary search to find the the maximum sustainable data rate when we are in an overload situation.

To empirically verify that our computed partitions are optimal, we established a ground truth by exhaustively running the speech detection application at every cut point on our testbed. Figures 8-8 and 8-9 show the results for six relevant cutpoints, both for a single node network (testing an individual radio channel) and for the full 20 node TMote network. WaveScript counts missed input events and dropped network messages on a per-node basis. The relevant performance metric is the percentage of sample data that was fully processed to produce output. This is roughly the product of the fraction of data processed at sensor inputs, and the fraction of network messages that were successfully received.

Figure 8-8 shows the input event loss and network loss for the single TMote case, and shows the resulting goodput. On a single mote, we can see the data rate is so high at early cutpoints that it drives the network reception rate to zero. Alternatively, at the later cutpoints too much computation is done on the nodes and therefore the CPU is busy for long periods and misses the vast majority of input events. In the middle, even a underpowered

TMote can process 10% of sample windows. This is equivalent to polling for human speech four times a second—a reasonably useful configuration.

Figure 8-9 compares the goodput achieved with a single TMote and basestation to the case of a network of 20 TMotes. For the case of a single TMote, peak throughput rate occurs at the 4th cut point (filterbank), while for the whole TMote network in aggregate, peak throughput occurs at the 6th and final cut point (cepstral). As expected, the throughput line for the single mote tracks the whole line closely until cut point six. For a high-data rate application with no in-network aggregation, a many node network is limited by the same bottleneck as a network of only one node: the single link at the root of the routing tree. At the final cut point, the problem becomes compute bound. In this case the aggregate compute power of the 20 TMote network makes it more potent than the single node.

We also ran the same test on an a Meraki Mini based on a low-end MIPS processor. While the Meraki has relatively little CPU power—only around 15 times that of the TMote—it has a WiFi radio interface with at least 10x higher bandwidth. Thus for the Meraki the optimal partitioning falls at cut point 1: send the raw data directly back to the server.

Having determined the optimal partitioning in our real deployment, we can now compare it to the recommendation of our partitioning algorithm. Doing this is slightly complex as the algorithm does not model message loss; instead, it keeps bandwidth usage under the user-supplied upper bound (using binary search to find the highest rate at which partitioning is possible), and minimizes the objective function. In the real network, lost packets may cause the actual delivered bandwidth to be somewhat less than expected by the profiler. Fortunately, the cut-point that maximizes throughput should be the same irrespective of loss as CPU and network load scale linearly with data rate.

In this case, binary search found that the highest data rate for which a partition was possible (respecting network and CPU limits) was at 3 input events per second (with each event corresponding to a window of 200 audio samples). The optimal partitioning at that data rate[1] was in fact cut point 4, right after filterbank, as in the empirical data. Like-

[1]In this case with $\alpha = 0, \beta = 1$, although the linear combination in the objective function is not particularly when we are maximizing data rate we are saturating either CPU or bandwidth

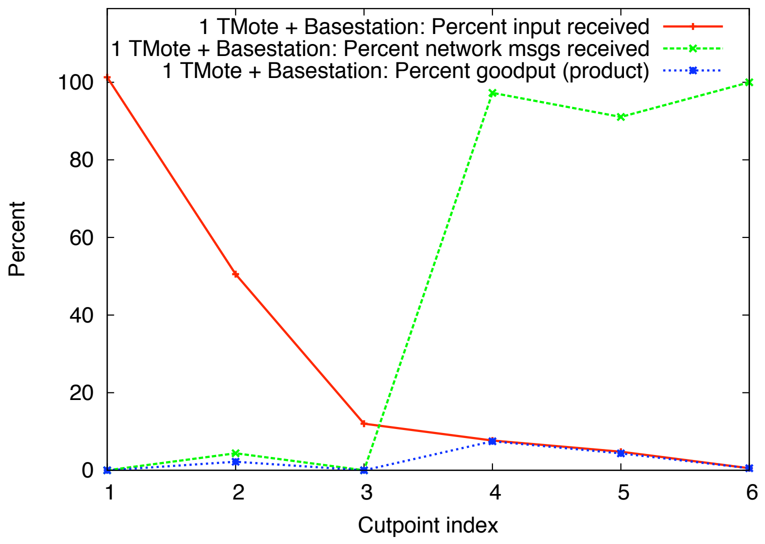


Figure 8-8: Loss rate measurements for a single TMote and a basestation, over different partitionings. The curves show the percentage of input data sampled, the percentage of network messages received, and the product of these which represents the percent goodput.

wise, the computed partitions for the 20 node TMote network and single node Meraki test matched their empirical peaks, which gives us some confidence in the validity of the model.

In the future, we would like to further refine the precision of our CPU and network cost predictions. To use our ILP formulation we necessarily assume that both costs are additive—two operators using 10% CPU will together use 20%, and don't account for operating system overheads or processor involvement in network communication. For example, on the Gumstix ARM-linux platform the entire speaker detection application was predicted to use 11.5% CPU based on profiling data. When measured, the application used 15% CPU. Ideally we would like to take an automated approach to determining these scaling factors.

## 8.4 Parallelism: Background Subtraction Case Study

This application is part of a computer vision system used to detect and identify birds in the wild. It is part of an effort by the UCLA Vision Group to achieve better classifiers for birds in one of their James Reserve sensor network deployments. For our purposes, it

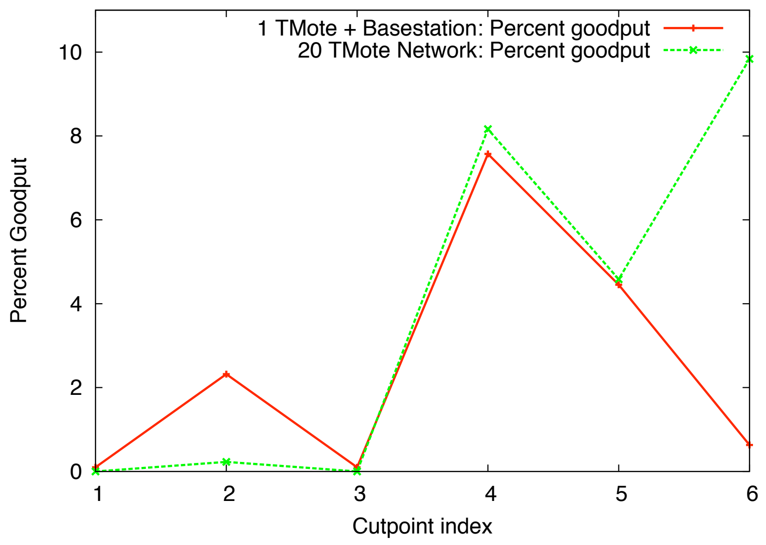
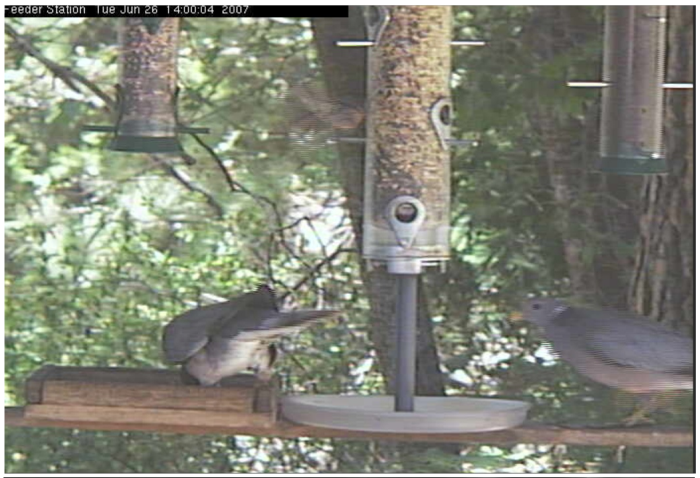


Figure 8-9: Goodput rates for a single TMote and for a network of 20 TMotes, over different partitionings when running on our TMote testbed.

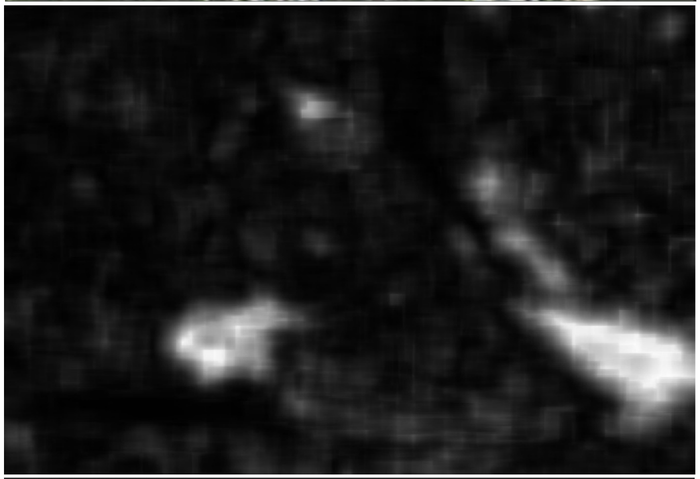
serves to illustrate the ability of metaprogramming to extend stream-based parallelism to also encompass stateful, parallel processing of matrices.

In this case study, we focus on the background subtraction component of the vision system. This phase is a precursor to actual classification of birds; classification algorithms perform better if the noisy background (leaves blowing, lighting shifting) is first removed. The background subtraction approach is based on building a model for the colors seen in the neighborhood around each pixel and subsequently computing the Bhattacharyya distance between each new pixel value and the historical model for that pixel's neighborhood. Therefore, the algorithm must maintain state for each pixel in the image (the model) and traverse each new input image, comparing each new pixel's value against the model for that pixel, and finally read the new values for all surrounding pixels, so as to update the model. An example result can be seen in Figure 8.4.

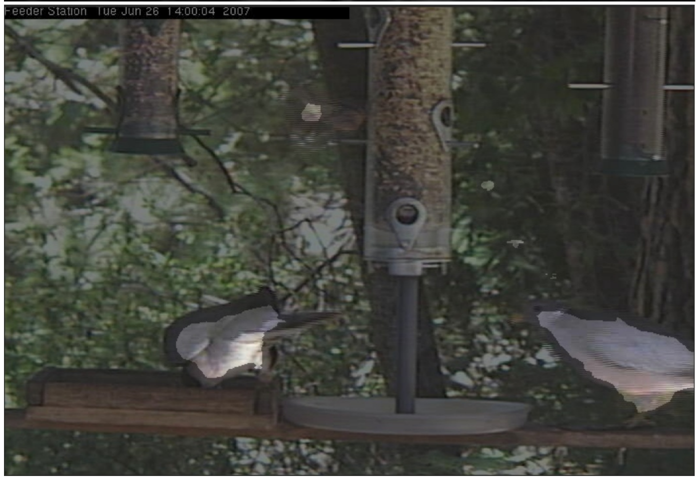
This algorithm is clearly data parallel. In fact, a separate process can compute each pixel's Bhattacharyya distance (and update each pixel's model) independently. But the data-access pattern is non-trivial. To update each pixel's model, each process must read an entire patch of pixels from the image around it. Thus *tiling* the matrix and assigning tiles to worker threads is complicated by the fact that such tiles must overlap so each pixel may



A:



B:



C:

Figure 8-10: Example of background subtraction: (A) Original Image (B) Difference from background model (C) Image with positive detections highlighted

reach its neighbors. For these reasons it is not straightforward to implement this algorithm in most stream processing languages. For example, stream processing languages tend to require that the input to each stream operator be a linear sequence of data. Exposing parallelism and exploiting locality in the background subtraction then requires an appropriate encoding of the matrix (for example, using Morton-order matrices), but this in turn creates complicated indexing. It is reasonable to say that the stream-processing paradigm is not the best fit for parallel matrix computations.

The original C++ source code for the background subtraction phase was shared with me by Teresa Ko from UCLA. Given the original source code, I proceeded in a four step process.

1. Port code verbatim to WaveScript.
2. Factor duplicated code; use higher-order functions.
3. Remove unnecessary floating point.
4. Parameterize design; expose parallelism.

**Port Verbatim:** Because WaveScript has imperative constructs and a similar concrete syntax, it is straightforward to do a verbatim translation of C or C++ code. This does not in any way extract parallelism (it results in a dataflow graph with one operator). But it is the best way to establish correspondence with the output of the original program and then proceed by correctness preserving refactorings.

The original source code possessed substantial code duplication, as seen in the verbatim port below. The code pictured is the core of the `populateBg` function, with the following signature.

```
populateBg :: (Array4D Float, Image) -> ();
type RawImage = Array Color;
type Image = (RawImage * Int * Int); // With width/height
type Array4D t = Array (Array (Array (Array t)));
```

This function builds the initial background model for each pixel. It is called repeatedly with a series of `Image`s to ready the background model before processing (classifying) new frames. In this version of the code, no significant type abstraction has yet been applied. The “model” for each pixel is a three-dimensional histogram in color space (the argument to `populateBg` is four dimensions to include a 3D histogram for each pixel in the image).

The background subtraction algorithm as a whole consists of 1300 lines of code containing three functions very much like `populateBg`. A second function updates the model for each new frame, and a third compares each new image to the existing model to compute Bhattacharyya distances for each pixel. Each of these functions traverses both the input image and the stored histograms (e.g. `matrix_foreachi` below). The code is rife with `for`-loops and indexing expressions and can be difficult to follow.

**Code Sample, First Version – Verbatim Port:**

```
for r = 0 to rows-1 {
  // clear temp patch
  fill(tempHist, 0);
  // create the left most pixel's histogram from scratch
  c :: Int = 0;
  roEnd = r - offset + SizePatch;  // end of patch
  coEnd = c - offset + SizePatch;  // end of patch
  for ro = r-offset to roEnd-1 { // cover the row
    roi = if ro < 0 then -ro-1 else
          if ro >= rows then 2 * rows-1-ro else ro;
    for co = c-offset to coEnd-1 { // cover the col
      coi = if co < 0 then -co-1 else
            if co >= cols then 2 * cols-1-co else co;
      // get the pixel location
      i = (roi * cols + coi) * 3;
      // figure out which bin
      binB = Int! (Inexact! image[i  ] * inv_sizeBins1);
      binG = Int! (Inexact! image[i+1] * inv_sizeBins2);
      binR = Int! (Inexact! image[i+2] * inv_sizeBins3);
      // add to temporary histogram
      tempHist[binB][binG][binR] += sampleWeight;
    }
  };
  // copy temp histogram to left most patch
  for cb = 0 to NumBins1-1 {
   for cg = 0 to NumBins2-1 {
     for cr = 0 to NumBins3-1 {
       bgHist[k][cb][cg][cr] += tempHist[cb][cg][cr];
  }}};
  // increment pixel index
  k += 1;
  // compute the top row of histograms
  for c = 1 to cols-1 {
      // subtract left col
      co = c - offset - 1;
      coi = if co < 0 then -co - 1 else
            if co >= cols then 2 * cols - 1 - co else co;

      for ro = r - offset to r-offset + SizePatch - 1 {
        roi = if ro < 0 then -ro-1 else
              if ro >= rows then 2 * rows - 1 - ro else ro;
        i = (roi * cols + coi) * 3;

        binB = Int! (Inexact! image[i+0] * inv_sizeBins1);
        binG = Int! (Inexact! image[i+1] * inv_sizeBins2);
        binR = Int! (Inexact! image[i+2] * inv_sizeBins3);

        tempHist[binB][binG][binR] -= sampleWeight;
        if (tempHist[binB][binG][binR] < 0) then {
          tempHist[binB][binG][binR] := 0;
        };
      };
      // add right col
      co = c - offset + SizePatch - 1;
      coi = if co < 0 then -co-1 else
            if co >= cols then 2 * cols-1-co else co;
      for ro = r-offset to r-offset + SizePatch - 1 {
        roi = if ro < 0 then -ro-1 else
              if ro >= rows then 2 * rows-1-ro else ro;

        i = (roi * cols + coi) * 3;

        binB = Int! (Inexact! image[i  ] * inv_sizeBins1);
        binG = Int! (Inexact! image[i+1] * inv_sizeBins2);
        binR = Int! (Inexact! image[i+2] * inv_sizeBins3);
        tempHist[binB][binG][binR] += sampleWeight;
      };
      // copy over
      for cb = 0 to NumBins1-1 {
      for cg = 0 to NumBins2-1 {
      for cr = 0 to NumBins3-1 {
        bgHist[k][cb][cg][cr] += tempHist[cb][cg][cr];
      }}};
      // increment pixel index
      k += 1;
  }
}
```

**Refactor Code:** The next step, before trying to expose parallelism, was to simply clean up the code. Some of this consisted of factoring out simple first-order functions to capture repeated index calculations (a refactoring applied just as easily to the original C++). Other refactorings involved using higher order functions, for example, to encapsulate traversals over the image matrices and thereby remove `for`-loops and indexing expressions. After refactoring the code was reduced to 400 lines; the `populateBg` function is shown below.

```
// Builds background model histograms for each pixel.
// Additions to the histograms are scaled according the assumption that
// it will receive NumBgFrames # of images.
//   tempHist: temporary space
//   bgHist  : background histograms
//   image   : frame of video stream
populateBg :: (PixelHist, Array PixelHist, Image) -> ();
fun populateBg(tempHist, bgHist, (image,cols,rows)) {
  using Array; using Mutable;
  assert_eq("Image must be the right size:",length(image), rows * cols * 3);

  // Patches are centered around the pixel.
  // [p.x p.y]-[halfPatch halfPatch] gives the upperleft corner of the patch.

  // Histograms are for each pixel (patch).
  // First create a histogram of the left most pixel in a row.
  // Then the next pixel's histogram in the row is calculated incrementally by:
  //   1. removing pixels in the left most col of the previous patch from the histogram and
  //   2. adding pixels in the right most col of the current pixel's patch to the histogram

  // This makes a strong assumption about the order the matrix is traversed.
  // It's not representation-independent in that sense.
  matrix_foreachi(bgHist, rows,cols,
    fun(r,c, bgHist_rc) {
      if c==0 then  initPatch(r,c, rows,cols, tempHist, image, sampleWeight1)
      else          shift_patch(r,c, rows,cols, tempHist, image, sampleWeight1);
      add_into3D(bgHist_rc, tempHist);    // copy temp histogram to left most patch
    })
}
```

Note that this code uses the same representations and performs the same operations as the original. We will see later how we further abstract the representation and expose parallelism.

**Reduce Floating Point:** One of our goals in porting this application was to run it on camera-equipeed cell phones, as well as on multicores and partitioned across phones and desktops. The phones in question use ARM processors without floating-point units. Thus, the penultimate step was to reduce the use of floating point calculations (for example, in calculating indices into the color histograms). WaveScript offered no special support for this. Its numeric operations are polymorphic, but WaveScript doesn't yet include built-in support for fixed point-arithmetic; rather, it is provided as a library[2]

**Expose Parallelism, Parameterize Design:** Finally, the most interesting part of this case study was using WaveScript to parallelize the application. Fortunately, the refactoring for parallelization (abstracting data types, matrix transforms) was also beneficial to code clarity. The essential change was to move away from code handling monolithic matrices (arrays) to expressing the transform locally on image tiles, and then finally at the level of the individual pixel (with the stipulation that a pixel transform must also access its local neighborhood). The end result was a reusable library for parallel matrix computations (see `parmatrix.ws`).

The first step is to implement transforms on local tiles. From the perspective of the client code, this is the same as doing the transform on the original matrix (only smaller).

[2]In the future we would like to add tool support for porting computations to fixed point, monitoring overflows and quantifying the loss of accuracy.

However, the library code must handle splitting matrices into (overlapping) tiles and disseminating those tiles to workers. The resulting dataflow graph structure will appear as in Figure 8.4[3]. The results of each independent worker are joined again, combined into a single matrix and passed downstream to the remaining computation. In Figure **??** we see the interface for building *tile kernels* via the function `tagged_tile_kernel`. Calling `tagged_tile_kernel`$(x, y, w, transform, init)$ will construct a stream transformer that splits matrices on its input stream into $x \times y$ tiles, with an overlap of $w$. First the *init* function is called (at metaprogram evaluation) to initializes the mutable state for each worker. Then at runtime each tile is processed by the $transform$ function, producing a new tile, potentially of a different type.

The type signature for `tagged_tile_kernel` is listed in Figure **??**. The extra "tagged" part is an additional complication introduced by the control-structure of the application. Typical of stream-processing applications, there is a tension between dividing an application into finer-grained stream kernels and avoiding awkward data movement between kernels (e.g. packing many configuration parameters into the stream). Because there is no shared state between kernels, all data must be passed through streams. In this particular example, it is frequently desirable to pass an extra piece of information (tag) from the original (centralized) stream of matrices down to each individual tile- or pixel-worker, which is exactly what the interface `tagged_tile_kernel` allows. For the background subtraction application, an earlier phase of processing determines what mode the computation is in (populating background model, or estimating foreground) and attaches that mode flag as a boolean on the stream of images.

The next step in building the `parmatrix.ws` library was to wrap the tile-level operator to expose only a pixel-level transform. This has a similar interface, shown below. Note that the resulting stream transformer on matrices has the same type as with the tile-level version.

[3]Whether the matrix tiles are physically copied depends on whether WaveScript is running in single-heap mode or with independent heaps. In single-heap mode it is possible for the workers to share one copy of the input image, except for two problems. (1) It is not clear that sharing one copy of the data is desirable given caching issues. (2) Unless unsafe optimizations are enabled the matrix tiles must be copied because WaveScript doesn't currently perform enough analysis to determine that the workers do not mutate the input image. One solution to (2) would be to add immutable vectors (a la Standard ML) to the language in addition to arrays, which would allow more sharing.

```
/*  This function creates X*Y workers, each of which handles a region
 *  of the input image.  The tile_transform is applied to each tile,
 *  and also may maintain state between invocation, so long as that
 *  state is encapsulated in one mutable value and passed as an
 *  argument to the transform.
 *
 *  This assumes the overlap between tilees is the same in both
 *  dimensions.
 */
tagged_tile_kernel ::
     (Int, Int, Int,
      ((tag, st, Tile px) -> Tile px2),
      (Tile px) -> st)
    ->
      Stream (tag * Matrix px) -> Stream (Matrix px2);

/* A tile is a piece of a matrix together with metadata to tell us
 * where it came from.  Fields are:
 *    (1) a matrix slice
 *    (2) tile origin on original matrix
 *    (3) original image dimensions
 */
type Tile t = (Matrix t * (Int * Int) * (Int * Int));
```

Figure 8-11: Code for tile level transform.

```
tagged_pixel_kernel_with_neighborhood  ::
   (Int, Int, Int,
    // Work function at each pixel:
    (tag, st, Nbrhood px) -> px2,
    // Per-pixel state initialization function
    (Int, Int) -> st)
  ->
   Stream (tag * Matrix px) -> Stream (Matrix px2);

type Nbrhood a = (Int,Int) -> a;
```

The `Nbrhood` type is used to represent the method for accessing the pixels in a local vicinity. It is a function mapping $(x, y)$ locations onto pixel values. Calling it with $(0, 0)$ gives the value of the center pixel—the one being transformed and updated. Now we are able to express the background subtraction application as a simple pixel-level transformation. The core of the implementation is show below. Given a boolean tag on the input stream (`bgEstimateMode`), and a state argument that contains the histogram for just that pixel (`bghist`), the kernel decides whether top populate the background (`populatePixHist`) or to estimate the foreground (`estimateFgPix`).

```
 tagged_pixel_kernel_with_neighborhood(workersX, workersY, overlap,
   // This is the work function at each pixel.
   fun(bgEstimateMode, bghist, nbrhood) {
     if bgEstimateMode  then {
       populatePixHist(bghist, sampleWeight1, nbrhood);
     } else {
       estimateFgPix(bghist, nbrhood);
     }
   },
```

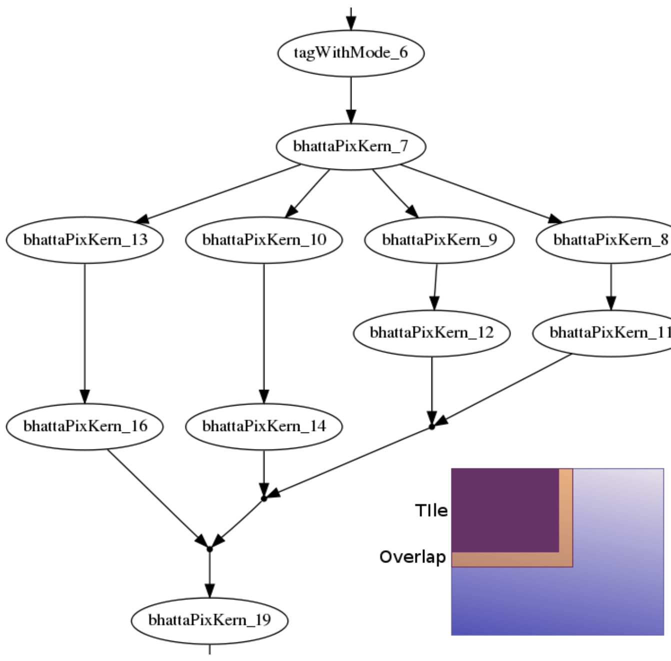


Figure 8-12: An example dataflow graph resulting from using a tile/pixel transform with *rows* = *cols* = 2. Even though the pixels are partitioned into four disjoint tiles (dark purple), an extra margin must be included around each tile (orange) to ensure each dark-purple pixel may access its local neighborhood.

```
    // This initializes the per-pixel state; create a histogram per pixel:
    fun(i,j) Array3D:make(NumBins1, NumBins2, NumBins3, 0))
```

**Conclusion:** The end result of this case-study was a cleaned up implementation that also exposes parallelism. (And a reusable parallel matrix library!) Parallel speedup results up to 16 cores are shown in Figure 8.4. These results are generated given a single, fixed 4×4 tiling of the matrix. A more appropriate approach is to have the metaprogram set the tiling parameters based on the number of CPUs on the target machine. But this is complicated by the need to factor the target number of threads into separate $x$ and $y$ components such that $xy = numthreads$. In general, this is fine, but for simplicity the measurements in Figure 8.4 are taken using a fixed grid size and allowing the operating system scheduler to figure out how to schedule the threads.

Allowing the metaprogram to read the number of CPUs and determine a grid size is another example of the utility of the metaprogram as a separate phase that precedes the WaveScript compiler's own dataflow-graph scheduling and optimization. (The first was in

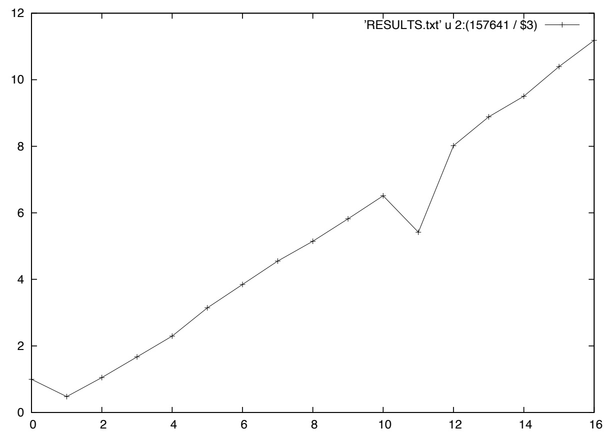


Figure 8-13: Parallel speedup for Bhattacharyya distance based background subtraction on 16-core AMD Barcelona machine. Datapoint 0 represents the single-threaded case with thread-support disabled. All runs are on a Linux 2.6 kernel, with the number of cores artificially limited using the `/sys/devices/system/cpu` interface.

Section 4.2, where Streamit-style scheduling was performed during metaprogram evaluation.) Still, it would be ideal to expose more of this tuning problem to the compiler itself. In the future perhaps a means will be found to reify the compiler's own profile-driven optimization and auto-tuning process in a way that it can be invoked by the programmer to determine the grid size.

## 8.5 Conclusion

This thesis described WaveScript, a type-safe, garbage collected, asynchronous stream processing language. We deployed WaveScript in an embedded acoustic wildlife tracking application, and evaluated its performance relative to a hand-coded C implementation of the same application. We observed a 1.38× speedup—which enabled a substantial increase in in-the-field functionality by allowing more complex programs to run on our embedded nodes—using half as much code. We also used this application to study the effectiveness of our optimizations, showing that the throughput of our program is substantially improved through domain-specific transformations and that our parallelizing compiler can yield near-linear speedups.

Additionally, we evaluated WaveScript's program partitioning capabilities. With WaveScript, users can run the same program on a range of sensor platforms, including TinyOS motes and various smartphones. WaveScript uses profiling to determine how each operator in the dataflow graph will perform using sample data. Its partitioning algorithm models the problem as an integer linear program that minimizes a linear combination of network bandwidth and CPU load, and uses program structure to solve the problem efficiently in practice. Our results on acoustic and EEG sensing applications show that the system computes optimal partitionings rapidly, enables effective exploration of application design space, and achieves good performance in practice.

In conclusion, we believe that WaveScript is well suited for both server-side and embedded applications, offering good performance and simple programming in both cases. For the embedded case, its potential to bring high-level programming to low-level domains is particularly exciting.

# Appendix A

# Related Work

WaveScript's design touches several areas of computer science. Related work falls into these broad categories: first, work on functional reactive programming and metaprogramming; second, stream processing languages and database query systems; third, work on languages and programming tools for sensor networks; and fourth, work on program partitioning.

## Functional Reactive and Meta-programming

Functional reactive programming (FRP) [23] is form of functional programming that deals with *values* that represent continuously varying *signals*. It was introduced in the context of the purely functional language Haskell. It differs from previous work on stream processing, first by addressing the subject in a purely functional manner, and second by modeling *continuously* varying signals rather than streams of discrete events. FRP is also *reactive* in that it enables the program to handle events that are triggered when a signal meets certain conditions. FRP was an important initial inspiration for Regiment, though the present WaveScript design has little in common with it.

*Metaprograms* are simply programs that generate programs. Metaprogramming, or "multi-stage" programming, is extremely common in the form of C++ templates, the C preprocessor, or even PHP scripts that generate websites. Also, macros in the Lisp and Scheme families of languages are examples of multi-stage programming [34]. Macros and preprocessors, however, are unstructured and undisciplined metaprogramming. Recently, there has been substantial work on metaprogramming in the context of strongly typed languages of the ML family. MetaML [65], for example, has explicit quotation and anti-quotation mechanisms for constructing run-time representations of pieces of ML code that, when executed, produce a value of a specific type. WaveScript, on the other hand, is a two-stage language in which the user's source code executes at compile time to generate a graph of stream operators. WaveScript differs from MetaML primarily in that it lacks explicit quotation setting apart the stages of computation. For example, in MetaML, one writes expressions such as `<1 + ~x>`, where <> are *quotations* constructing a piece of code, and ~ is an *anti-quotation* that injects the code value bound to the variable `x`. WaveScript avoids this by leveraging the asymmetry of its stages—the fact that generated stream graphs follow certain strict requirements differentiating them from source programs.

Metaprogramming is a natural choice for WaveScript because it addresses two requirements: one, the need to programatically construct large and often repetitive dataflow graphs, and two, the need for a static dataflow graph at runtime. Thus, dataflow graph generation instead happens at compile time. It is probably for these same reasons that you see a restricted form of multi-stage execution in some other stream languages, such as StreamIT. Also other there is a precedent in other domain specific languages adopting multi-staged evaluation for similar reasons, for example, the hardware synthesis language Bluespec [51]. Like WaveScript, these languages are *asymmetric* metaprogramming languages in that their different stages consist of different languages (in contrast with MetaML or R6RS Scheme).

## Stream Processing

Stream processing (SP) has been studied for decades. A survey of older work (through 1996) may be found in [64]. Some related literature takes place under other terminology, such as *dataflow* computing [5, 38]. In a typical SP model, a graph of operators (or kernel functions) communicate by passing data messages. Operators "fire" under certain conditions to process inputs, perform computation, and produce output messages. Many specific SP formalisms have been proposed. For example, operators may execute synchronously or not, deterministically or nondeterministically, and produce one output or many. Likewise, dozens of domain specific languages and libraries have been developed.

Ptolemy II [39] is widely-used data-flow system that can be used for modeling these many different dataflow models, and even for combining them. It is frequently used to model or simulate signal processing applications. Unlike WaveScope, it is not focused on efficiency or on providing a high-level, optimizable programming language.

SP has received renewed attention in recent years with the advent of multicore processors. Several stream processing languages arise out of a desire to program a particular parallel architecture. The Cell processor, RAW tiled architecture, and GPUs have all been recent targets for stream programming [66, 13]. (StreamIT [66] is one particular such language that will frequently serve as a point of comparison in this thesis.) There are even commercial chips designed specifically for stream processing such as the Storm processor from Stream Processors Inc. WaveScript, on the other hand, is a top-down project starting from a language design rather than a particular processor. In our work thus far, we have focused on two platforms: embedded sensor nodes (16-bit or 32-bit) and commodity desktop multicores.

## Streaming Database Management Systems

Apart from the architecture and compiler communities, the database community has developed its own take on stream processing. In the contexts of databases, streaming is incorporated by extending an SQL-like query language with the ability to operate over infinite tables (streams), which are turned into finite relations by various windowing operations. Two primary models arose in academic systems, from the Stanford STREAM [7] and Aurora/Borealis [14] projects, which were later adopted by different commercial entities respectively (Oracle/CQL [4] and StreamBase/StreamSQL [1]).

The difference between the models lies primarily in the treatment of time-stamps. Both require that all tuples have time-stamps (providing a partial order on tuples). The StreamSQL model processes every tuple, and CQL processes a batch of input tuples only when the timestamp advances [31]. Both models treat streams as asynchronous events, which differs from the majority of (synchronous) systems arising from signal processing, graphics, and architecture areas. This difference arises from distinct application concerns: applications such as finance or traffic monitoring involve asynchronous events. Partially because of the need to support dynamic scheduling, streaming databases have substantially more overheads than signal-processing systems, to the detriment of performance.

One function of WaveScript is that it serves as a general language that can be used both for both signal processing and streaming database applications—for known and unknown data-rates. When data rates are stable, WaveScript's profile-driven approach can apply the same static-optimization techniques as other streaming languages, otherwise it must fall back on dynamic scheduling as streaming databases.

Relative to streaming databases, WaveScript has the advantage that it is simultaneously the language for manipulating streams, and for writing high performance kernels that process streams. Typically, databases require that such kernels are written as user-defined functions (UDFs) in a separate language such as C++ or Java.

Another difference is that WaveScript supports windows as first-class entities, as opposed to conventional systems wherein windows are second class and tied to particular operators in the query plan. Flexible support for windows enables queries like time-alignment to be naturally expressed and efficiently executed. Sequence databases like SEQ [59] support time-series data as first class objects. However, these systems are targeted at simple queries that analyze trends in data over time, and are less suited to expressing complex signal processing functions. The same is true of Gigascope [18], a streaming sequence database system for network applications that shares our objective of handling high data rates.

## Language Design for SensorNets

Programming complex coordinated behaviors in sensor networks is a difficult task, which has inspired recent research into new programming models for sensor networks. The paucity of resources and operating system services on low-power sensor platforms provide a challenge. Sensornet programming tools research seeks to (1) raise the abstraction level of the code executed in-network (e.g. virtual machines), (2) replace or simulate missing operating system features, (3) capture network communication idioms, and (4) ease the task of partitioning an application between embedded nodes and heavier weight servers. WaveScript itself contributes to (1), (2), and (4), but not (3). Because sensor networks process streams of data, unsurprisingly many proposed sensor network programming abstractions are stream processing systems according to our previous definition.

Maté is a virtual machine designed for motes running the embedded TinyOS operating system. The Maté project focuses on application specific VM extensions, safety guarantees, and energy-efficient dynamic reprogramming. However, the overhead of bytecode interpretation makes intensive computation prohibitive. Maté hosts high-level programming systems through application specific VM extensions. For example, TinySQL is a query processing system built on a specialized Maté VM. Another specialized VM incorporates the abstract regions communication model of Welsh and Mainland [68]. Other high-level languages or language-extensions that have been proposed for programming sensor network applications include: Mottle [72], TinyScript [72], SNACK [28], and the variants of SQL found in Cougar [73] and TinyDB [42].

Tenet [27] addresses the problem of partitioning applications between embedded clients and servers. It proposes a two-tiered architecture with programs decomposed across sensors and a centralized server, much as in WaveScript. The VanGo system [29], which is related to Tenet, proposes a framework for building high data rate signal processing applications in sensor networks, similar to the applications that inspired our work on WaveScript. VanGo, however, is constrained to a linear chain of manually composed filters, does not support automatic partitioning, and runs only TinyOS code.

Marionette [71] and SpatialViews [50] use static partitioning of programs between sensor nodes and a server that is explicitly under the control of the programmer. These systems work by allowing users to invoke pre-defined handlers (written in a low level language like nesC) from a high-level centralized program that runs on a server, but don't offer any kind of automatic partitioning.

COSMOS [6] is a macroprogramming system that allows programmers to specify (possibly recursive) dataflow programs that are automatically placed inside of a network of motes subject to programmer-specifiable placement constraints. It is an example of a system that could benefit from the techniques in WaveScript, since it doesn't attempt to optimize operator placement using profiler output or other statistics.

Abstract Regions [69], Regiment [47], and Hood [70] provide languages and programming abstractions that perform operations over clusters of nodes (or "regions"), rather than single sensors. These abstractions allow data from multiple nodes to be combined and processed, but are targeted more at the coordination of sensors than at stream processing.

TinyDB [41] and Cougar [73] are distributed data flow systems, which typically run the same program on all nodes. These programs are quite constrained and typically are only partitioned in one way, with certain computations pushed into the network via heuristics (e.g., all filters should be applied inside the network; as much aggregation as possible should be inside the network, etc).

## Program Partitioning

WaveScript's profile-driven approach is conceptually different from distributed systems that move operators (computation) dynamically at run-time to optimize performance (e.g., message processing throughput or latency) or minimize resource consumption. In this section we review other systems the employ a compile-time approach to partitioning. Generally speaking, WaveScript differs from these existing systems in its use of a profile-driven approach to automatically derive an optimal partitioning, as well as its support for diverse platforms.

The Pleiades/Kairos systems [35] statically partition a centralized C-like program into a collection of node-level nesC programs that run on motes. Pleiades is primarily concerned with the correct synchronization of shared state between nodes, including consistency, serializability, and deadlocks. WaveScript, in contrast, is concerned with high-rate shared-nothing data processing applications, where all nodes run the same code. Because WaveScript programs are composed of a series of discrete dataflow operators that repeatedly process streaming data, they are amenable to our profile-based approach for cost estimation. Finally, by constraining ourselves to a single cut point, we can generate optimal partitionings quickly, whereas Pleiades uses a heuristic partitioning approach to generate a number of cut points.

Triage [9] is a related system for "microservers" that act as gateways in sensor network applications. Triage's focus is on power conservation on such servers by using a lower-power device to wake a higher-power device based on a profile of expected power consumption and utility of data coming in over the sensor network. However, it does not attempt to automatically partition programs across the two device classes as WaveScript does.

There has been substantial work looking at the problem of migrating operators at runtime, including many stream processing systems [8, 55, 10]. Dynamic partitioning is valuable in environments with variable network bandwidth, unpredictable load, but also comes with serious downsides in terms of runtime overheads. Also, by focusing on static partitioning, WaveScript is able to provide feedback to users at compile time about whether their program will "fit" into their chosen sensor platform and hardware configuration.

There has been some related work in the context of traditional, non-sensor related distributed systems. For example, the Coign [30] system automatically partitions binary applications written using the Microsoft COM framework across several machines, with the goal of minimizing communication bandwidth. Like WaveScript, it uses a profile-driven approach. Unlike WaveScript, Coign does not formulate partitioning as an optimization problem, and only targets Windows PCs. Neubauer and Thiemann [46] present a similar framework for partitioning client-server programs.

Automatic partitioning is also widely-used in high-performance computing, where it is usually applied to some underlying mesh, and in automatic layout of circuits.

# Appendix B

# Prototype Type System for Metaprograms

In Chapter 2 we saw an example of a metaprogram that would not reduce into a valid object program:

```
let f = λ ...
let g = λ ...
let s = iterate (λb . app(if b then f else g, 99)) init strm
in ...
```

I will now introduce a draft of a more restrictive typing regime that will rule out the above example, guaranteeing that terminating MiniWS programs reduce to valid MiniGraph programs—while still permitting a range of desirable higher-order programs. This type system will be described in the context of a mini-language even smaller than MiniWS. At the end of the section we will discuss application to MiniWS.

The goal of this type system is to ensure that every call site within certain regions of the program (`iterates`) will have a corresponding known code location ($\lambda$ expression) that it invokes. This is related to control-flow analysis. In the context of higher-order languages, control-flow analyses seek to identify the set of functions callable at each call site. These analyses are successfully employed in several functional language compilers [15, 2], and are especially effective in the context of whole program compilation.

It has been recognized in the literature that type systems can be equivalent to data- and control-flow analysis [52, 36]. In our present design we seek not to gather information about arbitrary higher-order programs, but to actively reject all programs that do not satisfy our restricted control-flow requirements.

Ultimately, our type system must determine which terms will call unknown code during their evaluation. Terms in operator position which are **not** guaranteed to reduce to $\lambda$-expressions when the program is in its normal form, are considered *unknown*. Conversely, any call to *known* code will successfully be resolved into a $\lambda$-expression (and subsequently $\beta$-reduced). The act of calling unknown code is a kind of computational effect tracked by the type system. This suggests similarities to existing work on effect type systems [54]. In a type-effect system, typing judgements $\Gamma \vdash t : T$ are extended to include additional information, $\varphi$, about the effects performed during the evaluation of $t$, giving us: $\Gamma \vdash t :^{\varphi} T$.

In the type system I present here we decorate types with two kinds of information: *effect* information (does the corresponding term call unknown code locations?), and, for arrow types, *control flow* information (does the corresponding value refer to a knowable code location). Each piece of information is a boolean, adding two bits to each type. We use metavariables $E, F, G \in \{0, 1\}$ for effects, and metavariables $U, V, W \in \{0, 1\}$ for control-flow ($X, Y, Z$ are reserved for normal type variables). We will refer to the new variables as $E/U$ variables (for Effects and Unknown-code), and constraints on them as $E/U$ constraints. Thus we write a type $T$ decorated with $E/U$ variables as: ${}^{E}_{U}T$. Note that $U$ variables are only relevant to function types, for all other types $U = 0$.

Let's look at a few examples. A function type containing an unknown code location would have type ${}_{U}(T_1 \rightarrow T_2)$ where $U = 1$, and an integer resulting from evaluating a term that *calls* unknown code would have type ${}^{E}Int$ where $E = 1$. A function that calls unknown code during its execution would have type $(T_1 \rightarrow^{E} T_2)$ where $E = 1$.

## B.1 First Phase: Type Checking

We formulate our type system as an extension to a typical constraint-based type-inference algorithm [53]. Such an algorithm has two phases. First, it traverses the input program, introducing fresh type variables for relevant program points, and recording constraints on these variables. These constraints are of the form *type* == *type*. The second phase then solves (unifies) the constraints.

In the process of type checking, our algorithm additionally introduces $E/U$ variables and collects $E/U$ constraints. These do not substantially affect the second phase—the new analysis does not affect the unification of "normal" types. But a third phase is added to solve $E/U$ constraints. The new constraints are as follows:

| | |
|---|---|
| $E = F$ | effect variables are equal |
| $U = V$ | unknown-code variables are equal |
| $\langle T \rangle$ | all arrows within a type $T$ have $U = 1$ |
| $E \leftarrow F$ | one effect variable depends on another $(F = 1) \Rightarrow (E = 1)$ |
| $E \leftarrow U$ | effect variable depends on an unknown-code variable |
| $Mask(E, [F_1, F_2, ...])$ | don't consider $F_1, F_2$ when computing $E$ via dependencies |

The use of these constraints will become apparent as we examine the typing rules in Figure B-1. These constraint typing rules involve relations of the form $\Gamma \vdash t : T \mid C$, where $C$ is a set of constraints including both the effect/control-flow constraints above and the usual $type = type$ constraints. For clarity, Figure B-1 omits additional constraints that are required to ensure freshness of variables[1]. Instead, simply assume that each typing rule introduces only unique variables.

We take one further notational liberty. *All* types are implicitly annotated with effect and unknown-code variables: ${}^{E}_{U}T$. In Figure B-1, whenever a type is written without one or the other, such as ${}^{E}T$, it is assumed that the elided variable(s) are present, unique, and unimportant for that particular rule. The implicit $U$ is attached to all references to $T$ in all parts of the rule.

One thing that we see here is that each typing statement contains, in addition to a type-environment $\Gamma$, an environment *mask*, $M$. This is present for typing $\lambda$-abstractions, which will be explained shortly. First, note that literals and variables may be typed with any mask,

[1]These constraints can be systematically added to Figure B-1 by annotating typing statements to track $\chi$ the set of type variables used: $\Gamma \vdash t : T \mid_{\chi} C$. At the expense of verbosity, all typing rules can be systematically rewritten to require that subderivations have disjoint variables, and that all newly introduced type variables are not present in any subderivation.

$$\textsc{LitB}\quad \frac{}{;M \vdash b :\ {}^{E}\texttt{Bool} \mid \{Mask(E, M)\}} \qquad \textsc{LitI}\quad \frac{}{;M \vdash i :\ {}^{E}\texttt{Int} \mid \{Mask(E, M)\}}$$

$$\textsc{Var}\quad \frac{}{\Gamma, v : T; M \vdash v : {}^{E}T \mid \{Mask(E, M)\}}$$

$$\textsc{If}\quad \frac{\begin{array}{c}\Gamma; M \vdash e_1 :\ {}^{E}\texttt{Bool} \mid \mathsf{C}_1 \qquad \Gamma; M \vdash e_2 :\ {}^{F}T_1 \mid C_2 \\ \Gamma; M \vdash e_3 :\ {}^{G}T_2 \mid C_3 \qquad C' = C_1 \cup C_2 \cup C_3 \cup \{{}^{F}T_1 = {}^{G}T_2,\ \langle T_1 \rangle,\ F \leftarrow E\}\end{array}}{\Gamma; M \vdash \texttt{if } \mathsf{e}_1 \texttt{ then } \mathsf{e}_2 \texttt{ else } \mathsf{e}_3 :\ {}^{\mathsf{F}}\mathsf{T}_1 \mid \mathsf{C}'}$$

$$\textsc{App}\quad \frac{\begin{array}{c}\Gamma; M \vdash e_1 :\ {}^{E}_{U}T_1 \mid C_1 \qquad \Gamma; M \vdash e_2 :\ {}^{F}T_2 \mid C_2 \\ C' = C_1 \cup C_2 \cup \{{}^{E}T_1 = ({}^{F}T_2 \to {}^{G}X),\ G \leftarrow U,\ G \leftarrow E,\ G \leftarrow F,\ Mask(G, M)\}\end{array}}{\Gamma; M \vdash \texttt{app}(\mathsf{e}_1,\ \mathsf{e}_2) : {}^{\mathsf{G}}\mathtt{X}}$$

$$\textsc{Abs}\quad \frac{\Gamma, v : T_1; M_1, M_2 \vdash e : T_2 \mid C \qquad M_1 = freeVars(e)}{\Gamma; M_2 \vdash \lambda v.e\ :\ T_1 \to T_2 \mid C}$$

Figure B-1: Typing rules to restrict control flow and ensure metaprogram evaluation will not become stuck.

just as they may be typed in any type-environment $\Gamma$. We typecheck a complete program with an empty mask as well as an empty type environment.

The purpose of these masks is to separate the computation already completed before the execution of a function from the computation performed by the body of that function. When we assign a type to a function, we do not include effects introduced in the computation of its free variables. Thus, the effects are masked from all the types of all the free variables. This hiding of effect annotations is *deep* in that it should propagate through type constructors (other than $(\rightarrow)$).

By associating masks with effect variables, we are recording in the constraints information about the structure of variable-binding and variable-reference in the program. With this information the constraint set is self contained, and the original program is unneeded in the constraint-solving phase. It will be necessary to unify the *type* == *type* constraints before resolving the masks, so as to have complete principle types (except for effects) and thereby determine which effect variables are reachable from which type variables.

There are a few other points to note in these typing judgements. The `If` rule does not introduce any new effect or type variables. (And thus it needn't introduce new *Mask* constraints, either.) It only equates and constrains existing variables. The `Abs` rule is the only one that extends the mask. The $\langle\rangle$ constraint introduced by `If` is the only place that effect/unknown-code variables are directly set to 1. And, like masks, the $\langle\rangle$ operator cannot be resolved until principle types are resolved for all type variables (and thereby the set of reachable $U$ variables is determined).

## B.2 Second phase: Constraint Solving for Regular Types

This phase proceeds by the usual method (described in [53]) of maintaining a stack of type equations, and performing repeated substitutions of type variables for their equivalent types. In the process, we record the resulting types assigned to each type variable, and thereby achieve principle types for all variable bindings in the program.

We modify this process to also accumulate a list of $E/U$ equations. Whenever a type equation $({}^{E}T_1 = {}^{F}T_2)$ is popped from the stack, we record constraints $E = F$ and $U = V$.

These resulting constraints are merged with the $E/U$ from the first type-checking phase and passed on to the third phase.

## B.3 Third phase: $E/U$ Constraint Solving

Solving $E/U$ constraints is somewhat more involved.

1. First, we use all the $(E = F)$ and $U = V$ constraints to generate equivalence classes for all $E/U$ variables. We then rename all references to $E/U$ variables in types and constraints to a representative member from the corresponding equivalence class.

2. Second, we use the principle types computed from phase two to determine which $E$ and $U$ variables are reachable from which type variables. The set of $U$ variables reachable from types with a $\langle T \rangle$ constraint form the "seed". All these $U$ variables are set to 1. The rest of the computation involves propagating these 1's through the various dependency constraints subject to the mask constraints.

3. We then process each distinct $Mask$ in the constraint set. Applying a mask consists of setting all the $E$ variables referred to in the mask to 0. We then form a table mapping each $E/U$ variable to a boolean or to *unknown*, and then recursively apply each $\leftarrow$ constraint to each $E \leftarrow F$ and $E \leftarrow U$ pair until no more changes are possible. This entire table reflects a solution to the system under *one particular* mask. We create a solution for every mask, and produce final values for all $E/U$ based on their corresponding masks.

## B.4 Application to MiniWS

The end result of the type-checking process above is a program with type information that indicates which functions might possibly invoke unknown code. In the case of the MiniWS language, the important point is to check that after type checking, the function argument to the `iterate` operator does *not* invoke unknown code. If that property holds, then the MiniWS program passes and is guaranteed to reduce to a MiniGraph program. Also, the above typing rules would need to be extended in a straightforward to handle the tuples and side-effects present in the MiniWS language.

## B.5 Type Checker Code Listing

```
module Checker_core3 where

-- This prototype handles a functional mini-language.  It can be
-- extended in a straightforward way to handle imperative constructs.

import Control.Monad.State.Lazy
import Data.Map
import qualified Data.Set as Set
import qualified Data.Map as Map

-- Data Type Definitions:
----------------------------------------

data Exp a =
     Lam a Vr (Exp a)
   | Let a Vr (Exp a) (Exp a) -- currently sugar for lambda/app
   | LitI Int
   | LitB Bool
   | Var Vr
   | If (Exp a) (Exp a) (Exp a)
   | App (Exp a) (Exp a)
 deriving Show

data EUConstraint =
     DeepUnknown  -- Taint all arrows that are reachable from this type var.
   | EDep Evr -- This tvar is dirty only if this other effect var is.
   | UDep Uvr     -- This is dirty only if this other var is unknown.
   | Mask [(Evr,Tyvr)] -- Evr should have its constraints solved after masking these others.
 deriving (Show, Ord, Eq)

type Vr = String
type Tyvr = String

data Evr = E String deriving (Show,Ord,Eq)
data Uvr = U String deriving (Show,Ord,Eq)

-- We map each Tvar onto a set of constraints.
-- Thus the domain includes a Tyvr and a Evr:
type EUConstraints = Map.Map Type (Set.Set EUConstraint)

-- These decorate types and indicate whether an arrow type has an
-- unknown code location, or whether a value caries the taint of an
-- unknown application.
type UTag = Uvr
type ETag = Evr

data Type  =
     Int   ETag
   | Bool  ETag
   | Tvar  ETag Tyvr
   | Arrow ETag UTag (Type ) (Type )
 deriving (Show, Ord, Eq)

-- Misc. Helper Functions
----------------------------------------
type GS a = State (Map.Map String Integer) a
```

```
gensym :: String -> GS String
gensym prefix =
    do tbl <- get
       let n = Map.findWithDefault 0 prefix tbl
       put (insert prefix (n+1) tbl)
       return (prefix ++ show n)
runGS m = evalState m Map.empty

add_to_mapset :: (Ord a, Ord b) => a -> b -> Map.Map a (Set.Set b) -> Map.Map a (Set.Set b)
add_to_mapset x y = Map.alter f x
   where
     f Nothing  = Just (Set.singleton y)
     f (Just s) = Just (Set.insert y s)

evar :: GS Evr
uvar :: GS Uvr
evar = do v <- gensym "e"; return (E v)
uvar = do v <- gensym "u"; return (U v)

fresh_tv = do v <- gentvar; e <- evar; return (Tvar e v)
gentvar = gensym "v"

type Tenv = Map.Map Tyvr Type
empty_tenv :: Tenv
empty_tenv = Map.empty
extend_tenv :: Tyvr -> Type -> Tenv -> Tenv
extend_tenv = Map.insert
lookup_tenv :: Tyvr -> Tenv -> Maybe Type
lookup_tenv = Map.lookup

-- -- This extracts the type vars that need to be scrubbed clean of effects.
skim_tenv :: Tenv -> [(Evr, Tyvr)]
skim_tenv tenv = Prelude.map (\ (_, Tvar t v) -> (t,v)) $ toList tenv

deJust (Just x) = x

-- First Phase: Type check
--------------------------------------------------------------------------------

-- Allowing a little syntactic sugar:
preprocess (Let a v e1 e2) = App (Lam a v e2) e1
preprocess (Lam a v e) = Lam a v $ preprocess e
preprocess (If a b c) = If (preprocess a) (preprocess b) (preprocess c)
preprocess (App a b) = App (preprocess a) (preprocess b)
preprocess (LitI i) = LitI i
preprocess (LitB b) = LitB b
preprocess (Var v)  = Var v

-- Type checking returns four values:
--  (1) The type of the input expression.
--  (2) A new expression
--  (3) A list of type equations (constraints), and
--- (4) a set of additional U/T constraints.
type MaskT = [(Evr, Tyvr)]
tcheck :: Tenv -> MaskT -> Exp () ->
          GS (Type, Exp (Tyvr, Tyvr, Tyvr), [(Type,Type)], EUConstraints)
tcheck tenv mask e =
  let
      defmask tvar = Map.singleton tvar defmask'
      defmask' = if mask == [] then Set.empty else Set.singleton (Mask mask)
      maskit s = if mask == [] then s else Set.insert (Mask mask) s
      add_mask (Tvar t v) cs    = adjust maskit (Tvar t v) cs
      add_constraint (Tvar t v) = add_to_mapset (Tvar t v)
  in
  case preprocess e of
    LitI i -> do new <- fresh_tv
                 let Tvar tnt _ = new
                 return (new, LitI i, [(new, Int tnt)], defmask new)
    LitB b -> do new <- fresh_tv
                 let Tvar tnt _ = new
                 return (new, LitB b, [(new, Bool tnt)], defmask new)

    -- Look up in tenv, return that type.  No new constraints.
    Var v -> do let tvar = deJust $ lookup_tenv v tenv
                return (tvar, Var v, [], defmask tvar)

    -- Any lambdas in the result of a conditional become Unknown (a deep operation)
    If pred cons alt ->
      do (Tvar tnt1 v1,e1,c1,ut1) <- tcheck tenv mask pred
         (t2,e2,c2,ut2) <- tcheck tenv mask cons
         (t3,e3,c3,ut3) <- tcheck tenv mask alt
         return (t2, If e1 e2 e3,
                 [(Tvar tnt1 v1, Bool tnt1), (t2, t3)] ++ c1 ++ c2 ++ c3,
                 -- It's ok to tag t2, since it's equated with t3
                 add_constraint t2 (EDep tnt1) $
                 add_constraint t2 DeepUnknown $
                  ut1 `union` ut2 `union` ut3)

    -- There are two different scenarios that can lead to an
    -- effect-tainted result from an application.  First, the operator
    -- may be a tainted value.  Second, the operator may be an unknown
    -- arrow type OR a tainted function value.
    App f x ->
       do (t1,e1,c1,ut1) <- tcheck tenv mask f
          (t2,e2,c2,ut2) <- tcheck tenv mask x
          ret <- fresh_tv; t <- evar
          let Tvar tnt1 _ = t1
          let Tvar tnt2 _ = t2
          u <- uvar
          let arrowt = Arrow t u t2 ret
          return (ret,
                  App e1 e2,
                  -- The operator typechecks with the arrow:
                  (t1, arrowt) :
                  -- The operand typechecks with the argument:
                  --(t2, a) :
                  c1 ++ c2,
                  -- An unknown arrow taints the result:
                  add_constraint ret (UDep u) $
                  -- A effect-tainted arrow taints the result:
                  add_constraint ret (EDep tnt1) $
                  -- A effect-tainted operand taints the result:
                  add_constraint ret (EDep tnt2) $
                  add_mask ret $
                  ut1 `union` ut2)

    Lam () v bod ->
       -- We could skim all the tvars off the surface of the tenv, and snapshot those.
       -- They can be stored as a constraint on the new typevar.
        do arg <- fresh_tv
           let tenv' = extend_tenv v arg tenv
           (ret,e1,c1,ut1) <- tcheck tenv' (skim_tenv tenv ++ mask) bod
           t  <- evar;  u  <- uvar
           let arrowt = Arrow t u arg ret
           result <- fresh_tv
           let Tvar _ retv = ret
           let Tvar _ argv = arg
           let Tvar _ resultv = result
           return (result,
                   Lam (argv,retv,resultv) v e1,
                   -- Equate the result with the arrow type we constructed:
                   (result, arrowt) : c1,
                   add_mask result $
                   add_mask arg   $ -- this one seems unnecessary
                   ut1)

```

```

-- Free type variables within a type, returns a set:
-- Do I need to check for extra free vars within the constraints??
free_tvars (Int t)  = Set.empty
free_tvars (Bool t) = Set.empty
free_tvars (Tvar t v) = Set.singleton v
free_tvars (Arrow t u a b) = Set.unions [free_tvars a, free_tvars b]

ty_effect (Int t) = t
ty_effect (Bool t) = t
ty_effect (Tvar t _) = t
ty_effect (Arrow t _ _ _) = t

-- Second Phase: Unify "Normal" Type Constraints
--------------------------------------------------------------------------------

-- With this unifier we build up a map of principle types, while
-- trying to keep it simple.  The result of unification is a type
-- environment containing principle types, as well as a set of U/T
-- equalities.

unify_types :: [(Type, Type)]
            -> (Tenv, [(ETag, ETag)], [(UTag, UTag)])
unify_types eqs = loop Map.empty [] [] $ eqs
 where
  loop tenv teqs ueqs [] = (tenv, teqs, ueqs)
  loop tenv teqs ueqs (eq:tl) =
   case eq of
     (x,y) | x==y  -> loop tenv teqs ueqs tl
     (Tvar t1 v1, Tvar t2 v2) | v1 == v2 -> error "unimplemenented"
     (Tvar t v, ty) | not (Set.member v $ free_tvars ty) -> shared v t ty
     (ty, Tvar t v) | not (Set.member v $ free_tvars ty) -> shared v t ty
     (Arrow t u a b, Arrow t2 u2 a2 b2) ->
         loop tenv ((t,t2):teqs) ((u,u2):ueqs) $ (a,a2):(b,b2):tl
     (Bool t1, Bool t2) -> loop tenv ((t1,t2):teqs) ueqs tl
     (Int  t1, Int  t2) -> loop tenv ((t1,t2):teqs) ueqs tl
     oth -> error$ "unification failed: "++show oth
   where
     shared v t ty = loop (update_principle v ty tenv)
                          ((t,ty_effect ty):teqs) ueqs (subst_ls v ty tl)

  -- This updates a principle type with new information.  It doesn't
  -- need to accumulate new constraints, it just needs to record the
  -- least-general type.
  update_principle v ty tenv =
     let tenv' = alter (overwriteWith ty) v tenv
         Just ty' = Map.lookup v tenv'
     in subst_map v ty' tenv'

overwriteWith ty Nothing      = Just ty
overwriteWith ty2 (Just ty1) = Just$ merge ty1 ty2

-- This is an inexhaustive case, but unification will fail first:
merge x (Tvar _ _)    = x -- FIXME-- pass the taint on??
merge   (Tvar _ _) y = y
merge (Int t1) (Int t2)   = Int t1
merge (Bool t1) (Bool t2) = Bool t1
merge (Arrow t u a b) (Arrow t2 u2 a2 b2) = Arrow t u (merge a a2) (merge b b2)

subst_map v ty = Map.map (subst_ty v ty)

subst_ls v ty [] = []
subst_ls v ty ((t1,t2):tl) = (subst_ty v ty t1, subst_ty v ty t2) : subst_ls v ty tl

subst_ty v ty oldty = case oldty of
        Int t   -> Int t
        Bool t  -> Bool t
        Tvar t vr -> if v == vr then ty else Tvar t vr
        Arrow t u a b -> Arrow t u (subst_ty v ty a) (subst_ty v ty b)


-- Third Phase: Unify Additional U/T Var Constraints
--------------------------------------------------------------------------------

trans_closure mapset =
  Map.mapWithKey loop mapset
  where
  loop key visited =
      -- Find all that are within two hops.
      let next = Set.unions $ Set.toList $
                  Set.map (\k -> Map.findWithDefault Set.empty k mapset) $
                          visited
          diff = Set.difference next visited
      in
      if Set.null diff
      then Set.delete key visited -- Delete self-edges currently.
      else loop key (Set.union visited next)

flatten_mapset m = concat $
                   Prelude.map (\ (x,set) -> zip (repeat x) (Set.toList set)) $
                   Map.toList m

equiv_classes ls = loop Set.empty ls
  where
  loop set [] = set
  loop set ((x,y):tl) =
    let contx = Set.filter (Set.member x) set
        conty = Set.filter (Set.member y) set
        both  = Set.union contx conty
        other = Set.difference set both
        set'  = Set.insert (Set.insert x $ Set.insert y $
                            Set.unions $ Set.toList both) other
    in loop set' tl

reachable_Uvars ty = loop ty
 where
  loop (Int t)     = Set.empty
  loop (Bool t)    = Set.empty
  loop (Tvar t _) = Set.empty
  loop (Arrow t u a b) = Set.insert u $
                         Set.union (loop a) (loop b)

-- Apply a map to all U/T vars in a type:
subst_ut_types tsubst usubst ty = lp ty
    where
    lt t = case Map.lookup t tsubst of { Just x -> x ; Nothing -> t }
    lu u = case Map.lookup u usubst of { Just x -> x ; Nothing -> u }
    lp (Int t)     = Int$  lt t
    lp (Bool t)    = Bool$ lt t
    lp (Tvar t v) = Tvar (lt t) v
    lp (Arrow t u a b) = Arrow (lt t) (lu u) (lp a) (lp b)

subst_ut_cstrts tsubst usubst set = Set.map f set
    where
    lt t = case Map.lookup t tsubst of { Just x -> x ; Nothing -> t }
    lu u = case Map.lookup u usubst of { Just x -> x ; Nothing -> u }
    f (Mask tyls)  = Mask$ Prelude.map (\ (t,tyvr) -> (lt t, tyvr)) tyls
    f (EDep t)     = EDep$ lt t
    f (UDep u)     = UDep$ lu u
    f DeepUnknown  = DeepUnknown

-- Here we do the full unification, both on types and on taint/unknown constraints.

unify_all cstrnts __uts =
   -- For debugging purposes this temporarily returns a ton of junk:
   (principle_types, reachableUs, utbl, ttbl, crosstbl, alltaints, tequivclasses, uequivclasses)
```

```haskell
  where
   (__principle_types, __teqs, __ueqs) = unify_types cstrnts

   -- Respect equivalence constraints.  Here we smash down the U/T
   -- variables into equivalence classes, and rewrite all Dep/Mask
   -- constraints in terms of these.
   tequivclasses = equiv_classes __teqs
   uequivclasses = equiv_classes __ueqs

   -- Now we must rewrite the principle_types and the U/T constraints:
   -- We use the "first" var in each equivalence class as a proxy for that class.
   subst c = Set.fold (\ cls mp ->
                      --let fst = head$ Set.toList cls in
                      let fst = Set.findMin cls in
                      Set.fold (\ x mp -> Map.insert x fst mp) mp cls)
             Map.empty c
   tsubst = subst tequivclasses
   usubst = subst uequivclasses

   sty = subst_ut_types tsubst usubst
   principle_types = Map.map sty __principle_types
   uts = Map.fromList $
         Prelude.map (\ (ty,set) ->
                      (sty ty, subst_ut_cstrts tsubst usubst set)) $
         Map.toList __uts

   -- Next, we solve all the constraints on taint/unknown variables:
   -- This first involves tracking down "DeepUnknown"-induced dependencies.
   reachableUs = Map.fold Set.union Set.empty $
                 Map.filter (not . Set.null) $
                 Map.mapWithKey (\ (Tvar t v) s ->
                                 if Set.member DeepUnknown s
                                 then reachable_Uvars (Map.findWithDefault (error "missing") v principle_types
                                 else Set.empty)
                                uts

   -- Second, we build graphs with edges representing equalities between U/T variables.
   utbl :: Map.Map Uvr (Set.Set Uvr)
   utbl = Map.empty
   ttbl = loop' flat Map.empty
   -- And dependencies from U->T
   crosstbl = loop'' flat Map.empty
   flat = flatten_mapset uts
   loop' [] m = m
   loop' ((Tvar t1 _, EDep t2) : tl) m = loop' tl (add_to_mapset t2 t1 m)
   loop' (_:tl) m = loop' tl m
   loop'' [] m = m
   loop'' ((Tvar t1 _, UDep u) : tl) m = loop'' tl (add_to_mapset u t1 m)
   loop'' (_:tl) m = loop'' tl m

   -- For a given mask, solve the constraints.
   -- Return sets of "on" U/T vars.
   solve_ut mask = alltaints
     where
     tnts = Prelude.map fst mask
     tnts' = Set.fromList tnts
     -- The masked taint vars cannot be activated or activate others:
     ttbl' = foldl (\ mp tnt ->
                    let m = Map.filter (not . (Set.member tnt)) $ -- Kill any entry that would activate us
                            Map.delete tnt mp
                    in
                    -- Hygiene: filter out entries from the range that are no longer in the domain.
                    Map.map (Set.filter (\x -> Map.member x m)) m
                    )
            ttbl tnts
     crosstbl' = Map.map (\s -> Set.difference s tnts') crosstbl
     ttbl'' = trans_closure ttbl'
     lkp tbl u = Map.findWithDefault Set.empty u tbl

     -- We seed the dependencies with the reachable U's and fill everything in:
     alltaints =
       Set.union crossover $ Set.unions $ Set.toList $
       Set.map (lkp ttbl'') crossover
     -- U vars that activate T vars:
     crossover =
       Set.unions $ Set.toList $
       Set.map (lkp crosstbl') reachableUs

   -- Finally, we compute a solution by solving for each mask.  We memoize the results.
   alltaints = Set.difference ts0 killed
     where
     init = Map.singleton [] ts0
     ts0 = solve_ut []
     -- This accumulates a list of U/T vars that are NOT set because of masking.
     killed = Set.fromList $ loop init (toList uts)
     loop memo [] = []
     loop memo ((Tvar tv _, set):tl) =
         case get_mask set of
           [] -> loop memo tl
           ls -> let memo' = if Map.member ls memo
                                then memo
                                else Map.insert ls (solve_ut ls) memo
                      Just ts = Map.lookup ls memo'
                 in if Set.member tv ts
                    then loop memo' tl
                    else tv : loop memo' tl
     get_mask set =
       case Set.toList $ Set.filter is_mask set of
         [] -> []
         [Mask ls] -> ls
     is_mask (Mask _) = True
     is_mask _        = False
```

# Bibliography

[1] http://www.streambase.com/.

[2] Standard ml of new jersey. http://www.smlnj.org/.

[3] Automatic derivation and implementation of signal processing algorithms. *SIGSAM Bull.*, 35(2):1–19, 2001.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–242, 2006.

[5] Arvind and R.S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on*, 39(3):300–318, Mar 1990.

[6] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *Proc. EuroSys*, 2007.

[7] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[8] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. USENIX NSDI*, San Francisco, CA, March 2004.

[9] Nilanjan Banerjee, Jacob Sorber, Mark D. Corner, Sami Rollins, and Deepak Ganesan. Triage: balancing energy and quality of service in a microserver. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 152–164, New York, NY, USA, 2007. ACM.

[10] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Gün Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Operating Systems Review*, 36(2), 2002.

[11] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[12] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, and William Mitchell. *Zoltan 3.0: Data Management Services for Parallel Applications; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4749W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[14] D. Carney, U. Cetintemel, M. Cherniak, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB*, 2002.

[15] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71, London, UK, 2000. Springer-Verlag.

[16] Ronald A. Cole, Joseph Mariani, Hans Uszkoreit, Annie Zaenen, and Victor Zue. Survey of the state of the art in human language technology, 1995.

[17] Gerardo Costa and Colin Stirling. Weak and strong fairness in ccs. *Inf. Comput.*, 73(3):207–244, 1987.

[18] C. Cranor, T. Johnson, O. Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.

[19] S.B. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. on ASSP*, 28:357–366, 1980.

[20] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, 1976.

[21] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[22] R. Kent Dybvig. The development of chez scheme. In *ICFP '06: Proc. of the 11th ACM SIGPLAN intl. conf on Functional prog.*, pages 1–12, New York, NY, USA, 2006. ACM.

[23] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[24] M. Garey, D. Johnson, , and L. Stockmeyer. Some simplified np complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[25] Lewis Girod, Martin Lukac, Vlad Trifa, and Deborah Estrin. The design and implementation of a self-calibrating acoustic sensing system. In *SenSys*, 2006.

[26] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Xstream: A signal-oriented data stream management system. In *ICDE*, 2008.

[27] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM Press.

[28] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80. ACM Press, 2004.

[29] Ben Greenstein, Christopher Mar, Alex Pesterev, Shahin Farshchi, Eddie Kohler, Jack Judy, and Deborah Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *SenSys*, pages 279–292, 2006.

[30] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Proc. OSDI*, 1999.

[31] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, 2008.

[32] Simon Peyton Jones et al. Playing by the rules: Rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.

[33] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[34] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[35] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. PLDI*, 2007.

[36] Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theor. Comput. Sci.*, 364(3):292–310, 2006.

[37] John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, New York, NY, USA, 1993. ACM Press.

[38] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[39] Edward A. Leet. Overview of the ptolemy project. Technical Report Technical Memorandum No. UCB/ERL M03/25, UC Berkeley, 2003.

[40] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[41] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[42] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[43] A. Martin, D. Charlet, and L. Mauuary. Robust speech/non-speech detection using LDA applied to MFCC. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 237–240, 2001.

[44] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[45] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Science*, 17:348–375, 1978.

[46] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM.

[47] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN*, 2007.

[48] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. the First International Workshop on Data Management for Sensor Networks (DMSN)*, Toronto, Canada, August 2004.

[49] Ryan Newton and Matt Welsh. The regiment macroprogramming system. In *Sixth International Conference on Information Processing in Sensor Networks (IPSN'07)*, April 2007.

[50] Yang Ni, Ulrich Kremer, and Liviu Iftode. Spatial views: space-aware programming for networks of embedded systems. In *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003*, 2003.

[51] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[52] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–378, New York, NY, USA, 1995. ACM.

[53] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.

[54] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT press, 2005.

[55] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. ICDE*, 2006.

[56] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 39–48, New York, NY, USA, 1992. ACM.

[57] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. pages 39–48, 1992.

[58] Juhani Saastamoinen, Evgeny Karpov, Ville Hautamki, and Pasi Frnti. Accuracy of MFCC-Based speaker recognition in series 60 device. *EURASIP Journal on Applied Signal Processing*, 2005(17):2816–2827, 2005.

[59] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database systems. In *VLDB*, 1996.

[60] Peter Sewell et al. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, 2007.

[61] Eugene Shih and John Guttag. Reducing energy consumption of multi-channel mobile medical monitoring algorithms. 2008.

[62] Ali Shoeb et al. Patient-Specific Seizure Onset. *Epilepsy and Behavior*, 5(4):483–498, 2004.

[63] Ali Shoeb et al. Detecting Seizure Onset in the Ambulatory Setting: Demonstrating Feasibility. In *IEEE EMBS 2005*, September 2005.

[64] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[65] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.

[66] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *ICCC*, April 2002.

[67] Stephen Weeks. Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM.

[68] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[69] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[70] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. Mobisys*, 2004.

[71] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proc. IPSN*, 2006.

[72] Bridging The Gap: Programming Sensor Networks with Application Specific Virtual Machines. Uc berkeley tech report ucb//csd-04-1343.

[73] Yong Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.