Design and Evaluation of a Compiler for Embedded Stream Programs

Ryan R. Newton Lewis D. Girod
Michael B. Craig Samuel R. Madden
MIT CSAIL
{newton, girod, mic, madden}@csail.mit.edu

J. Greg Morrisett
Harvard University
greg@eecs.harvard.edu

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, we employ a novel 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, we incorporate an extensible system for rewrite rules to capture algebraic properties in specific domains (such as signal processing).

We have used our language to build and deploy a sensor-network for the acoustic localization of wild animals, in particular, 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, including distributed execution and substantial parallel speedups. 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.

Categories and Subject Descriptors:
D.3.2 Concurrent, distributed, and parallel languages; Applicative (functional) languages; Data-flow languages

General Terms: Design, Languages, Performance

Keywords: stream processing language, sensor networks

1. Introduction

This paper presents the design and implementation of the *WaveScript* programming language. WaveScript supports efficient processing of high-volume, asynchronous data streams. Data stream processing applications—which typically process data in real time as it arrives—arise in a number of domains, from filtering and mining feeds of financial data to extracting relevant features from continuous signals streaming from microphones and video cameras.

We have used WaveScript in several applications requiring high-volume stream-processing, including water pipeline leak detection and road surface anomaly detection using on-board vehicular accelerometers. In this paper, however, we focus on our most mature application: a distributed, embedded application for acoustic localization of wild Yellow-Bellied marmots which was deployed at the Rocky Mountain Biological Laboratory in Colorado during August, 2007.

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. In previous work, we developed the hardware platform for this application (10), performed pilot studies to gather raw data, and developed signal processing algorithms to perform the localization (3). During our recent deployment, we used WaveScript to accomplish the next stage of our research—building a real-time, distributed localization system that biologists can use in the field, while also archiving raw-data for offline analysis.

The marmot localization application uses an eight-node *VoxNet* network, based on the earlier acoustic ENSBox nodes (10), 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 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.

For this application, WaveScript met three key requirements:

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 the marmot-detection application. 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 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.

LCTES '08 6/13/2008, Tucson, AZ.

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 elements of a stream to be grouped (via special windowing operations) into windows—called Signal Segments, or "Sigsegs"—that are synchronous. 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 marmot-detection application.

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 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. Our technique partially evaluates programs into stream dataflow graphs, enabling abstraction and modularity with *zero* performance 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 show in this paper, providing rewrite rules with a library (e.g. DSP operators) can enable the user to compose functionality at a high-level, without sacrificing efficiency.

The rapid prototyping benefits of high-level, polymorphic, garbage-collected languages are widely recognized. WaveScript merely removes certain performance barriers to their application in the embedded domain by exploiting a *domain-specific* programming model. Several subcomponents of the marmot application were previously implemented by different authors (using MATLAB or C). These provide a natural point of comparison for our approach. As we demonstrate in this paper, WaveScript enabled us to rapidly develop the application, writing less code, while *improving* runtime performance versus previous implementations.

In previous work, we argued the need for WaveScript, and evaluated Sigseg implementations and scheduling policies for the single-node runtime engine (11). In this paper, we present the WaveScript compiler for the first time, including the implementation of the above features. Finally, we evaluate the performance the WaveScript marmot application, and the contribution of optimizations to that performance—examining CPU usage for embedded computation and parallelization for desktop/server computation.

2. Related Work

Stream processing (SP) has been studied for decades. In a typical SP model, a graph of operators communicate by passing data messages. Operators "fire" under certain conditions to process inputs, perform computation, and produce output messages. Many SP models have been proposed. For example, operators may execute asynchronously or synchronously, deterministically or nondeterministically, and produce one output or many.

In this section, we briefly review these major tradeoffs, and then discuss how existing stream processing systems are inappropriate for our application domain. We also discuss related systems which inspired our approach.

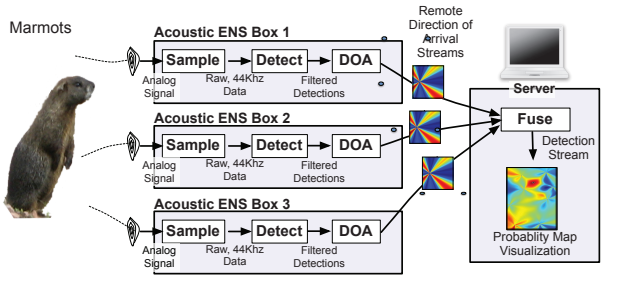

Figure 1. Diagram illustrating the major components of the marmot-detection application.

2.1 Tradeoffs in Existing Stream Processing Systems

A major divide in SP systems is between those that are *synchronous* and those that are *asynchronous*. A synchronous SP model is generally thought of by analogy with synchronous hardware—a *clock* is driving operators to fire at the same time. On the other hand, in this paper the most relevant aspect of synchronicity, and the sense in which we'll be using the term, is whether or not data streams are synchronized *with each other*.

Most literature on streaming databases has focused on an asynchronous streaming model (6; 4), while most compiler literature has focused on synchronous models (12; 5; 7). WaveScript, like a streaming database, targets applications that involve multiple feeds of data arriving on network interfaces at unknown rates. This essentially forced us to adopt an asynchronous SP model and its associated overheads, including explicit annotation of data records with timestamps and the use of queueing and dynamic scheduling to handle unpredictable data rates. In contrast, in purely synchronous systems, because data rates are known and static, timestamps are not needed, and it is possible for the compiler to statically determine how much buffering is needed, avoiding queuing and scheduling overheads.

Hence, synchronous systems typically outperform asynchronous ones. Our approach, as mentioned in the introduction, is able to overcome performance limitations of the asynchronous model by recognizing that many high data rate asynchronous applications—including our audio processing system— do in fact have some regularity in their data rates some of the time. Thus, in WaveScript, data streams are decomposed into windows (called *Sigsegs*) containing data elements that are regularly sampled in time (or *isochronous*). For example, a single Sigseg might represent a few seconds of audio sampled at 44 kHz. Asynchronous streams of Sigsegs are then the primary streaming data type in WaveScript. Data rates and timing between Sigsegs on these streams are inherently unpredictable, since a stream may represent the union of several streams arriving from different nodes on the network, or may have had filters applied to it to remove some Sigsegs that are not of interest to the application.

In summary, no existing system provides exactly the mix of features and performance that WaveScript requires. Stream processing database systems, like Aurora (6) and Stanford Stream (4), are purely asynchronous and designed to process at most a few thousand data items per second, which is insufficient for real-time processing of multiple channels of audio data arriving at hundreds of kilosamples per second. Existing high-performance stream programming languages are synchronous and therefore inappropriate for our domain. Finally, general purpose languages, though clearly expressive enough to build our application, are not amenable to many of the optimizations that a stream compiler can apply, and

lack support for automatic parallelization and distribution of programs, thus complicating the job of the programmer.

2.2 StreamIt and FRP

Although we chose to implement our own SP system, rather than reuse an existing one, WaveScript draws inspiration from two existing projects in particular. These are *functional reactive programming* (FRP) and *StreamIt* (12). FRP embeds asynchronous events and continuously valued signals into Haskell (9). FRP programs may use event handlers to reconfigure the collection of signal transformers dynamically. This provides an extremely rich context for manipulating and composing signals. FRP has been applied to animation, robotics, computer vision, and mathematical visualization. Unfortunately, both because it is implemented atop Haskell, and because signals are dynamic and reconfigurable, FRP does not deliver competitive performance for data-intensive applications.

StreamIt is a C-like stream programming language with static memory allocation that targets the synchronous dataflow domain. StreamIt provides a syntax for constructing stream graphs using loops, first-order recursive functions, and a second-class representation of streams. This provides a high degree of control in constructing graphs, but is not as expressive as FRP.

Recent releases of StreamIt include support for streams with unknown rates. Using this facility, it may have been possible to implement our acoustic localization application with StreamIt. We would, however, have had to extend the StreamIt implementation in a number of ways. First, StreamIt has primarily been developed for architectures other than our own (e.g., RAW). Also, it would need a foreign function interface for accessing sensor hardware and networking, as well as the accompanying infrastructure for distributed execution. Finally, in addition to these engineering hurdles, we chose not to employ StreamIt because it is a goal of our research to target the data-intensive streaming domain with a flexible language that includes dynamic allocation, garbage collection, and first-class streams. Dynamic allocation and garbage collection allow for higher level programming and better encapsulated libraries. Further, when working with dynamic workloads (e.g. unpredictable numbers of marmot calls) portioning resources dynamically can make more efficient use of finite memory.

3. The WaveScript Language

In this section, we introduce the language and provide code examples drawn from our marmot-detection application. We highlight the major features of the language, leaving implementation issues for Section 4. The details of the language are documented in the user manual (1).

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 (12; 5; 16), a WaveScript program is structured as a set of communicating stream operators.

In WaveScript, however, 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 take and produce streams. Those stream transformers in turn do the work of assembling the stream *operators* that make up the nodes in an executable stream graph.

Figure 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. 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 much more convenient than writing directly in C, where streams would be implicit.

```
// Node-local streams, run on every node:
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);
}
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 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. 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 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 2 and 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.

3.1 Core Operators: iterate and merge

WaveScript is an extensible 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 shown in Figure 3. 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.

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 (α), and the current state of the operator (σ). It produces as output zero or more elements on the output stream (List β) as well as a new state (σ). **Iterate** must also take an additional argument specifying the initial state (σ).

3.2 Defining Synchronization

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**.

One example, shown in Figure 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 sample only particular time ranges of data. In Figure 2, **syncSelect** extracts windows containing event detections from all four channels of microphone data on each node.

In Figure 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 4, **Zip** is defined by iterating over the output of **mergeTag**, maintaining buffers of past stream elements, and producing output

```
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 4. *Zip*—the simplest synchronization function.

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.

3.3 Windowing and Sigsegs

The `marmotScores` function in Figure 3 consumes a stream of Sigsegs. In addition to capturing locally isochronous 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 τ)` 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 (11).

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.

3.4 Distributed Programs

A WaveScript program represents a graph of stream operators that is ultimately partitioned into subgraphs and executed on multiple platforms. In the current implementation, this partitioning is user-controlled. The code in Figure 2 defines streams that reside on the node as well as those on the server. Function definitions outside of these blocks may be used by either. The crossings between these partitions (named streams declared in one and used in the other), become the points at which to cut the graph. Note that with the application in Figure 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. In addition to the code in Figure 2, the programmer also must specify an inventory of the nodes—type of hardware, and so on—in a configuration file. 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 (using TCP sockets in our implementation).

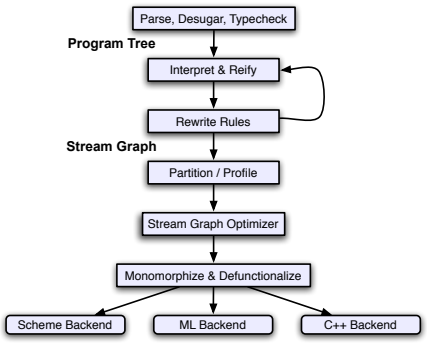

Figure 5. Compilation Work-flow and Compiler Architecture

4. WaveScript Implementation

The WaveScript language is part of the ongoing WaveScope project, which delivers a complete stream-processing (SP) system for high data-rate sensor networks. As such, it includes many components that fall outside the scope of this paper, including: the networking layer, scheduling engines, and control and visualization software. What we will describe in this section are the key points in the implementation of the compiler, with a brief treatment of issues pertaining to runtime execution.

4.1 A Straw-man Implementation

A naive way to implement WaveScript is to use a 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. (This approach is used in some streaming databases, to the detriment of performance (11).)

4.2 The WaveScript Approach

WaveScript instead exposes the stream graph to the compiler, allowing a range of stream optimizations described in Section 5. 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 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 WaveScope runtime system was described in (11).

The overall structure of the WaveScript compiler is depicted in Figure 5. 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.

4.3 Semantics and Evaluation 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 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.

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

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 C++ backend (described in Section 4.4) omits these features. The main benefit of the evaluation method, however, is that it enables stream graph optimizations. Thus the performance increases described in Section 6.2 can be directly attributed to *interpret&reify*.

Our method is in contrast with *metaprogramming*, where multiple explicit stages of evaluation are orchestrated by the programmer. For example, in MetaML (17), 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. [1] 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 (15). Closures become λ-expressions once again. Variables in the closure's environment are recursively reified as let-bindings surrounding the λ-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 λ-expression containing

[1] Originally, we used an evaluator that applied reduction rules, including β- and δ-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.

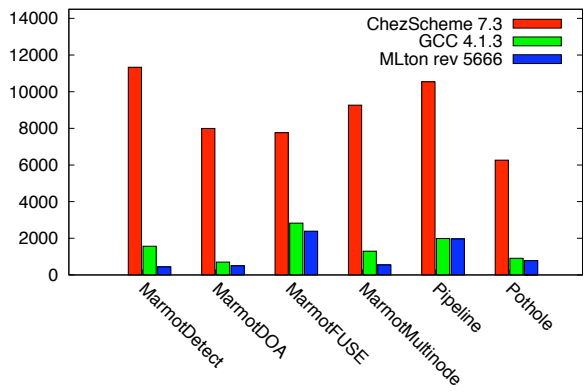

Figure 6. Execution times (in milliseconds) of current WaveScript backends on application benchmarks. Single-threaded benchmark on AMD Barcelona, optimizations enabled. Benchmarks include three stages of the marmot application (detection, DOA, and fusing DOAs), as well as a complete multinode simulation—eight 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.

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
```

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 4) leaving only **iterate**s, **merge**s, and network/source operators—in other words, a stream graph.

4.4 WaveScript backends

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. Our current backends generate code for Chez Scheme (8), MLton (18), and GCC. For the purpose of targeting new architectures, we may extend WaveScript to generate code for other languages as well, including, perhaps, other stream-processing languages, or languages that would enable compilation on 8-bit "mote" platforms (such as NesC/TinyOS).

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 6.

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 (18), 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 our current implementation 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 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 6, in spite of its limitations, our current prototype C++ runtime is the best choice 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.

5. 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.

To use the profiling features, representative sample data is provided along with the input program. In our marmot application, the sample audio data provided includes both periods of time with and without marmot alarm calls. The current implementation uses the Scheme embedding of WaveScript to execute part or all of the stream graph on the sample data.

The profiler measures the number of elements passed on streams, their sizes, and the execution times of stream operators.

The relative execution times of operators (in Scheme) are taken to be representative of the other backends as well. This method is expedient, and provides the best support for incremental or repeated profiling of the stream graph, but if a more precise notion of relative times is called for, we may need to perform profiling in other backends in the future (at the cost of much longer compile times).

5.1 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 (12). 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 fissioning operators. 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. We do not yet try to automatically optimize inter-node parallelism.

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.

5.1.1 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.

5.2 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 (13). 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 (14). And there has been particularly intensive study of rewrite rules in the context of signal processing (2).

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.

5.3 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.

```
// 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 7. Pseudo-code for the portion of the compiler that applies rewrite optimizations.

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-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 7. 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**.

6. Evaluation

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, we showcase our compiler optimizations in the context of this application, explaining their effect and evaluating their effectiveness.

6.1 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 timeframe. 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 in real-time—something the previous implementation did not accomplish (due to limited CPU).

Table 1 shows results for both the continuously running detector, and the occasionally running DOA algorithm (which is invoked

Table 1. Performance of WaveScript marmot application vs. hand-written 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

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 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 transmission 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 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

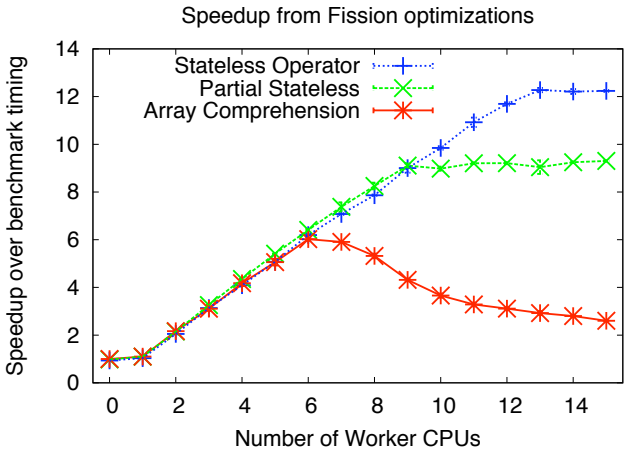

Figure 8. Parallel speedups achieved by applying fission optimizations to the DOA phase of the marmot application.

6.2 Effects of Optimization on Marmot Application

Here we relate the optimizations described in Section 5 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 5.2, 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 5.1 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 benefits not at all.

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 5.1.

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 4×4 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 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 $12\times$ 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 5.1, 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 $6\times$. 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.

7. Conclusion

We 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 $1.38\times$ 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.

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.

References

[1] Wavescript users manual, http://regiment.us/wsman/.

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

[3] A. M. Ali, T. Collier, L. Girod, K. Yao, C. Taylor, and D. T. Blumstein. An empirical study of acoustic source localization. In *IPSN '07: Proceedings of the sixth international conference on Information processing in sensor networks*, New York, NY, USA, 2007. ACM Press.

[4] A. Arasu et al. Stream: the stanford stream data manager (demonstration description). In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 665–665, New York, NY, USA, 2003. ACM.

[5] I. Buck et al. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[6] 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.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of prog. lang.*, pages 178–188, New York, NY, USA, 1987. ACM.

[8] R. K. 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.

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

[10] L. Girod, M. Lukac, V. Trifa, and D. Estrin. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *ACM SenSys*, Boulder, CO, Nov 2006.

[11] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. Xstream: A signal-oriented data stream management system. In *ICDE*, 2008.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proc. of the 12th intl. conf. on Arch. support for prog. lang. and op. sys.*, pages 151–162, New York, NY, USA, 2006. ACM.

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

[14] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. pages 39–48, 1992.

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

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

[17] 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.

[18] S. 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.