Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2008-005
CBCL-270

January 31, 2008

WaveScript: A Case-Study in Applying a Distributed Stream-Processing Language

Ryan Newton, Lewis Girod, Michael Craig, Sam Madden, and Greg Morrisett

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

WaveScript: A Case-Study in Applying a Distributed Stream-Processing Language

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

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. 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.

1. Introduction

This paper presents the design and implementation of the *WaveScript* programming language. WaveScript is designed to support 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.

1.1 Application: Locating Yellow-Bellied Marmots

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 in August, 2007.

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 (11), 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. Several of the subcomponents of the system we built had previous implementations in MATLAB or C. These provide a natural point of comparison for our approach.

The marmot localization application uses an eight-node network of Acoustic ENSBox nodes (11), based on the XScale PXA 255 processor. 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 the purposes of this application, WaveScript provided three key features:

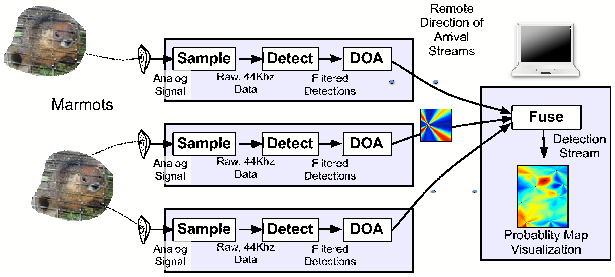
1. **Embedded Operation:** A compiled WaveScript program yields an efficient, compact binary which is well suited to the low-power, limited-CPU nodes used in the marmot-detection application. WaveScript also includes features such as the ability to integrate with drivers that capture sensor data, interfaces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

[Copyright notice will appear here once 'preprint' option is removed.]

2. **Distribution:** WaveScript is a *distributed* language in that the compiler can execute a single program across many nodes in a network or on multiple processors in a single node. Nodes may use substantially different hardware infrastructures, and data may arrive from multiple, distributed data sources.

   Ultimately, distribution of programs is possible because WaveScript, like other languages for stream-processing, encodes computations as distinct stream operators 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. **Asynchronicity:** 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 purpose in designing WaveScript was to bring high-level, declarative, type-safe programming to the data-intensive, embedded domain. The common wisdom is that this type of programming is inappropriate in this context. We argue that by designing a language and compiler *specifically* for the asynchronous streaming domain, that we can overturn this common wisdom. For the applications we built, this appears to be true.

Efficiency in WaveScript is provided by three key techniques. First, the compiler employs a *novel evaluation strategy* that uses a combination of interpretation, reification, and compilation to partially evaluate programs into stream dataflow graphs, enabling abstraction and modularity with *zero* performance overhead.

Second, WaveScript uses a *profile-driven optimization* strategy to enable many static optimizations and static parallelization of stream dataflow graphs. Specifically, WaveScript uses representative sample data for streams to estimate rates and computation times of each operator, and then applies a number of well-understood techniques from the synchronous dataflow world.

Third, WaveScript uses *domain-specific rewrite rules* to improve performance and to enable abstract library routines with simple interfaces to be intuitively composed while still providing good performance.

In the rest of this paper, we give an overview of the WaveScript language and the design of its compiler. We explain the implementation of the above features, and highlight how applying the optimizations is made easier by WaveScript's declarative nature and its high-level operations for manipulating streams and other datatypes. Finally, we evaluate the performance of our application's components as compared with earlier, handwritten C versions written by different authors, and we demonstrate how our optimizations benefit this application in particular.

2. Related Work

Stream processing (SP) has been studied for decades. In a typical SP model, a graph of operators communicate by passing data tokens. Operators "fire" under certain conditions to process input tokens, perform computation, and produce output tokens. 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