

The Case for a Signal-Oriented Data Stream Management System

Position Paper

Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan,
Hari Balakrishnan, Samuel Madden
MIT CSAIL

Email: wavescope@nms.csail.mit.edu

ABSTRACT

Sensors capable of sensing phenomena at high data rates—on the order of tens to hundreds of thousands of samples per second—are useful in many industrial, civil engineering, scientific, networking, and medical applications. In these applications, high-rate streams of data produced by sensors must be processed and analyzed using a combination of *both* event-stream and signal-processing operations. This paper motivates the need for a data management and continuous query processing architecture that integrates these two different desired classes of functions into a single, unified software system. The key goals of such a system include: the ability to treat a sequence of samples that constitute a “signal segment” as a basic data type; ease of writing arbitrary event-stream and signal-processing functions; the ability to process several million samples per second on conventional PC hardware; and the ability to distribute application code across both PCs and sensor nodes.

1. INTRODUCTION

There is a need for data management and continuous query processing systems that integrate high data rate event-stream and signal-processing operations into a single system. This need is evident in a large number of signal-oriented streaming applications, including preventive maintenance of industrial equipment; detection of fractures and ruptures in pipelines, airplane wings, or buildings; *in situ* animal behavior studies using acoustic sensing; network traffic analysis; and medical applications such as anomaly detection in electrocardiogram signals.

These target applications use a variety of embedded sensors, each sampling at fine resolution and producing data at rates as high as hundreds of thousands of samples per second. In most applications, processing and analyzing these streaming sensor samples requires non-trivial event-stream and signal-oriented analysis. In many cases, signal processing is application-specific, and hence requires some amount of user defined code. Current general-purpose data management systems fail to provide the right features for these applications: stream processing engines (SPEs) such as Aurora [8], STREAM [23], and TelegraphCQ [9] as well as more recent commercial offerings (*e.g.*, StreamBase, Coral8) handle event processing over streaming data, but don’t provide a convenient way to write user-defined custom code to handle signal processing operations. In particular, they suffer from an “impedance mismatch”, where data must be converted back and forth from its representation in the streaming database to an external language like Java or C++, or even to a separate system

like MATLAB. Signal processing operations in these external languages are usually coded in terms of operations on arrays, whereas most SPEs represent data streams as sequences of tuples. These sequences need be packed into and unpacked from arrays and be passed back and forth. The conversion overheads imposed by this mismatch also limit the performance of existing SPEs when performing signal processing operations, constraining the applicability of these existing systems to lower rate domains.

Another option for building signal processing applications is to use a graphical modeling package such as Simulink or LabVIEW. These systems, which offer high-level programming languages (sometimes data-flow oriented), lack stream processing capabilities, database-like optimization features, distributed execution, and the ability to integrate naturally with relational data stored on disk. It is both inconvenient and inefficient to just use one of these signal processing systems as a front- or back-end to a conventional stream processor because many applications require alternating sequences of event stream and signal processing operations.

This paper describes the motivation and high-level architecture of a combined event-stream and signal-processing system that we are building as part of the WaveScope project. The project’s components include:

- A programming language, *WaveScript*, that allows users to express signal processing programs as declarative queries over streams of data.
- A high-performance execution engine that runs on multiprocessor PCs.
- A distributed execution engine that executes programs written in *WaveScript* over both conventional PCs across a network and between PCs and embedded sensor nodes.

WaveScript includes several noteworthy features. Its data model introduces a new basic data type, the *signal segment*. A signal segment is a sequence of isochronous (*i.e.*, sampled regularly in time) data values (samples) from a signal that can be manipulated as a batch. *WaveScript* natively supports a set of operations over signal segments. These include various transforms and spectral analyses, filtering, resampling, and decimation operations. Another important feature of *WaveScope* is that users express both queries and user-defined functions (UDFs) in the same high-level language (*WaveScript*). This approach avoids the cumbersome “back and forth” of converting data between relational and signal-processing operations. The *WaveScript* compiler produces a low-level, asynchronous data-flow graph similar to query plans in traditional streaming systems. The runtime engine efficiently executes the query plan over multiprocessor PCs or across networked nodes, using both compiler optimizations and domain-specific rule-based optimizations.

This position paper is primarily concerned with the main features of the *WaveScript* language and outlines some elements of the execution engines and the optimization framework. It attempts to make the case for a signal-oriented streaming system, but does not describe design details, report on an implementation, or provide any experimental results.

This article is published under a Creative Commons License Agreement (<http://creativecommons.org/licenses/by/2.5/>).

You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007, Asilomar, California, USA.

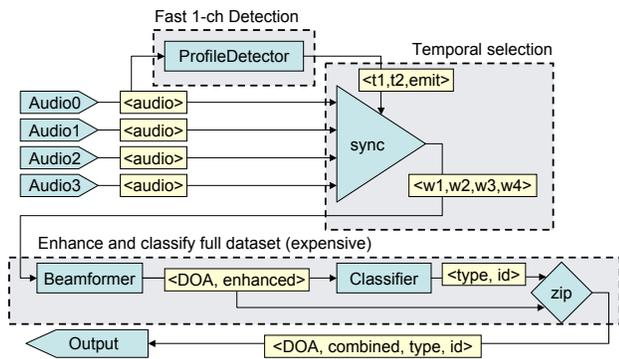


Figure 1: Marmot call detection, direction-of-arrival estimation, and classification workflow.

2. KEY FEATURES

To understand the challenges involved in building a signal oriented stream processor, it is helpful to first consider a specific example application. Here we present an example drawn from an acoustic sensor network deployed by scientists to study the behavior of the yellow-bellied marmot (*Marmota flaviventris*) in the wild [24]. The idea is to first use one audio channel to detect the possibility of a nearby marmot. A detection triggers processing on multiple audio channels to detect, orient, localize, and classify the most likely behavior of one or more of these rodents.

Each sensor produces samples of acoustic data at a certain configurable frequency—typical data rates are 44 KHz per sensor, and a typical setup [14] might include eight or ten microphone arrays each consisting of four microphones (for a combined data rate of about 1 MHz). The following questions are of interest:

1. Is there current activity (energy) in the frequency band corresponding to the marmot alarm call?
2. If so, which direction is the call coming from? Use that direction to enhance the signal using beamforming on a four-channel array.
3. Is the call that of a male or female? Is it a juvenile? When possible, identify and distinguish individual marmots.
4. Where is each individual marmot located over time?
5. Are marmots more responsive to alarm calls from juveniles? Are males and females commonly found in mixed or separate groups? Are juveniles predominantly located near males or females? Do subsets of individuals tend to stay together?

Figure 1 shows a block diagram representing the processing steps required by the first two queries shown above. The first query uses continuous spectrum analysis to estimate the energy content in the frequency range of interest, followed by a smoothed noise estimator and threshold detector with hysteresis to capture complete calls; this is labeled “Fast 1-ch Detection” in Figure 1. The second query is implemented by first extracting these time windows of interest from the historical data recorded at a multi-channel acoustic array; this is labeled “Temporal Selection”. Next, the query estimates the direction of arrival (DOA) of the marmot call using an Approximate Maximum Likelihood (AML) beamforming algorithm, enhances the signal by phase-shifting and combining the four separate channels, and finally passes the enhanced signal to a classification algorithm; this is the “Enhance and Classify” box.

¹A medium-sized rodent common in the western US.

Interestingly, the processing needed to answer this sort of query turns out to be similar across domains ranging from industrial monitoring to network traffic analysis. For example, in industrial monitoring, vibration signals are matched against known signatures to determine if a piece of equipment is about to fail. We have distilled the issues raised by a range of applications into five key features that must be provided by a signal-oriented stream processing system:

A data model with explicit support for signals. Most signal processing operations need to run over windows of hundreds or thousands of data samples. To process these windows, existing stream processing systems would either treat groups of samples as uninterpreted blocks of data, or would represent each sample as a separate tuple and impose sliding windows on these tuples as they stream into operators. In contrast, our approach is to define a new fundamental data type, the *signal segment* (*SigSeg*). A *SigSeg* is a first-class object that represents an isochronous window of fixed bandwidth samples in a data-stream. Query processing operators process these *SigSeg* objects, operating directly on the *SigSeg*’s implied time index without incurring the overhead of explicit per-sample timestamps. It is important to note that our data model is not completely isochronous; although readings within a *SigSeg* are assumed to be isochronous, individual tuples (each containing *SigSegs*) may arrive asynchronously.

This approach should yield a substantial performance improvement over traditional approaches. Keeping samples within a *SigSeg* isochronous is one key to providing this performance boost, because there is no space overhead for explicitly storing timestamps, and efficient time-range lookups are possible. By providing a compact data representation that can be passed by reference, we anticipate that using *SigSegs* will also improve cache performance, a critical factor in processing-intensive applications.

Integrated language for expressing queries and UDFs. WaveScript combines relational and signal processing operations in one language, for the reasons mentioned in Section 1.

Efficient runtime execution. The runtime system must minimize in-memory copying and avoid scheduler overhead. First, it is essential to use reference counting on *SigSegs* to eliminate copies as data flows between operators. Second, the scheduling discipline must be designed to allow inter-operator control flow to bypass the scheduler where possible, while still supporting query plans with branching and enabling intra-query parallelism via multi-threading.

Extensible optimization framework. We aim to provide an optimization framework that supports database-like commutativity and merging optimizations, rule-based optimizations similar to extensible query optimizers [16,26] to express signal processing optimizations over query plans, and various compiler optimizations.

Distributed execution. WaveScope targets many applications that must be implemented using distributed sensors. To address this, the query plan must be divided into segments that run on different nodes, and each segment must be compiled and optimized to run on the appropriate target platform. Finally, inter-node links in the query plan must efficiently support both wired and wireless networks.

3. DATA AND PROGRAMMING MODEL

In this section, we summarize the WaveScope data model, and discuss how queries and operators are programmed.

3.1 Data Model

WaveScope models data as a stream of tuples. Each tuple within a stream is drawn from the same schema, and each field in the schema

has a type. Field types are either primitive types (*e.g.*, integer, float, character, string), arrays or sets, tagged unions (variant records), or signal segments (SigSegs).² A SigSeg represents a window into a signal (time series) of fixed bandwidth values that are regularly spaced in time (isochronous). Hence, a typical signal in WaveScope will be represented by a stream of tuples, where each tuple contains a SigSeg object that represents a fixed sized window on that signal. A SigSeg object is conceptually similar to an array, in that it provides methods to get values of elements in the portion of the signal it contains and determine its overall length. However, SigSegs also contain a *timebase* that specifies the measurement times of values in the SigSeg and provides a set of methods for comparing and mapping between signals sampled from sensors at different rates. Although values within a SigSeg are isochronous, a data stream itself may be asynchronous, in the sense that the arrival times of tuples will not be spaced completely regularly in time (this is particularly likely to be true of streams in the middle of a WaveScope plan, after filtration and transformation operations have been applied.)

Streams of tuples in WaveScope follow pass-by-value (copying) semantics between operators, including tuples containing SigSegs. Pass-by-value can be implemented in several ways. For example, in the case of SigSegs, the implementation will likely include performance optimizations to reduce the cost imposed by copy semantics, *e.g.*, using reference-counting and copy-on-write, which are discussed in more detail in Section 4. The implementation of the data model has important performance implications, but should not affect application semantics.

Regardless of the particular implementation, the WaveScope data model treats SigSegs as first-class entities that are transmitted in streams, and may be stored, processed, and forwarded at will. This is unlike other streaming systems that impose windows on individual tuples as a part of the execution of individual operators, but do not allow the windows themselves to be manipulated. By making SigSegs first-class entities, windowing can be done once for a whole chain of operators, and logical windows can be stored and passed around dynamically, rather than being defined by the query plan. Packing blocks of readings together in SigSegs is natural for many signal processing operations that operate on fixed sized windows, and is much more efficient for high data rate operations as operators are invoked on blocks of hundreds or thousands of samples at once.

In this paper we focus on one dimensional signals (*e.g.*, streams of audio and vibration data), but in general the WaveScope data model also supports SigSegs which refer to multidimensional signals (*e.g.*, streams of images).

3.2 Programming Model

A compiled WaveScript program is a data-flow graph of stream operators. The WaveScript source, however, is a script that generates this graph. Within this script, the user may define and invoke reusable stream-graph constructor functions. We call these *subquery constructors*. These, like macros, evaluate at compile time to produce clusters of connected stream operators. In addition, the user writes *POD functions* (plain-old-data functions), which in our context refer to functions that neither produce nor consume streams. These functions may be used from *within* the bodies of stream operators as they process data, and may be either inlined (like subquery constructor functions) or remain separately compiled.

²The WaveScript source language also allows user defined functions to process nested tuples, as well as polymorphic tuples (where not all field types are determined). These features, however, disappear during the compilation process.

A single, integrated language for queries, reusable subquery-constructors, and POD functions yields a number of advantages. First, by using a single data model, WaveScript avoids the complexities of mediating between the query itself and user-defined functions residing in an external language (such as C). Further, it enables *type-safe* construction of queries. In contrast, a language like SQL is frequently embedded into other languages, but the programmer that embeds the query is given no compile-time guarantee about the well-formedness of that SQL query.

We illustrate WaveScript with a code snippet from the marmot detection query. ProfileDetect is a subquery-constructor that connects together a number of more basic stream-processing operators in a reusable way. It instantiates a series of stream-processing operators which search a data stream for windows matching a given frequency profile (see Figure 1).

```
fun profileDetect(S, scorefun, <winsize,step>, threshsettings) {
  // Window input stream, ensuring that we will hit each event
  wins = rewindow(S, winsize, step);
  // Take a hanning window and convert to frequency domain.
  scores : Stream< float >
  scores = iterate(w in hanning(wins)) {
    // Compute frequency decomposition
    freq = fft(w);
    // Score each frequency-domain window
    emit (scorefun(freq));
  };
  // Associate each original window with its score.
  withscores : Stream<float, SigSeg<int16>>
  withscores = zip2(scores, wins);
  // Find time-ranges where scores are above threshold.
  // threshFilter returns <bool, starttime, endtime> tuples.
  return threshFilter(withscores, threshsettings);
}
```

Note that WaveScript is statically typed, but types are inferred from variable usages using standard techniques [22].³ In the above program, *wins*, *scores*, and *withscores* are all streams of tuples. The type of *withscores*, for example, is `Stream<float, SigSeg<int16>>`. Notice that we allow a tuple to contain SigSegs, and set-valued types.

ProfileDetect first creates a new windowed view of the samples in the input stream using the `rewindow` operator. Here, the input stream *S* is divided into windows of size *winsize* that advance by *step*. These values are application-specific: *winsize* determines the resolution of the frequency decomposition, while *step* determines how sparsely the stream is sampled. For example, for events shorter than one window, *step* must be chosen such that adjacent windows overlap, whereas longer events can be detected with sparser sampling of the channel. The detector then computes the frequency decomposition of each individual window, and passes this frequency map to a custom scoring function to generate a match score. It accomplishes this using the `iterate` construct (discussed below) to execute code against each individual window in the stream. Finally, the detector zips together the original data with the computed match score.

Zip2 is a simple operator that merges two streams pair-wise, synchronously. Since data streams in general are asynchronous, zipping only works properly when it is known that there will be a one-to-one correspondence between the input streams being zipped. Often a more sophisticated strategy is needed to merge streams. Zip2 turns out to be good enough in this case because all its inputs are derived from the same source stream.

Note that operations like `hanning`, `fft` and `rewindow` are library functions—common signal processing operators included

³Type annotations may optionally be included for clarity. For example, the marmot-detection code in the appendix contains type declarations for each top-level function.

Class	Examples
POD Functions (built-in, linked from C, or user-defined)	arithmetic, SigSeg operations, timebase operations, FFT/IFFT, profileDetect, classify
Subquery-Constructors	beamForm, sync, zip
Fundamental Stream Operators	iterate, union

Table 1: Classes of Programming Constructs in WaveScript.

with WaveScope. We are developing an extensive library of such functions, but do not detail them here.

Table 1 lists the different classes of programming constructs that are available in WaveScript. Subquery-constructors may be application-defined (like `profileDetect`) or may be defined within the WaveScript library. In either case they may be augmented with optimization rules as discussed in Section 5. Built-in POD functions are low-level primitive operations (or externally linked C functions) that cannot or should not be implemented in WaveScript. The basic stream operators (`iterate`, `union`) are special programming constructs for manipulating streams and are the fundamental operators recognized by the runtime engine (as discussed in Section 4.1).

Main query body: Next, we will take a look at the body of a WaveScript program for detecting and classifying marmots based on their audio calls (shown graphically in Figure 1). The first thing to do is configure the data sources. This can occur in the query file, or separately in an included catalog file:

```
Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);
```

These statements declare that C0-C3 are audio streams sampled at 48 KHz from audio channels 0 through 3 and are to be windowed into windows of 1024 samples each. With the variables C0-C3 bound, we can now write a basic marmot detector using `profileDetect` and a few other subquery constructors defined in WaveScript.

```
control = profileDetect(Ch0, marmotScore, <64,192>,
<16.0, 0.999, 40, 2400, 48000>);
// Use the control stream to extract actual data windows.
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);
beam<doa,enhanced> = beamform(datawindows, arrayGeometry);
marmots = classify(beam.enhanced, marmotClassifier);

return zip2(beam, marmots);
```

The above query first uses `profileDetect` as a real-time prefilter on one of the four audio channels. The result is a `<bool,time,time>` control stream, which is used to “snapshot” certain time ranges and discard others. `Sync4` accomplishes this by aligning the four streams in time, and extracting the data contained within the time-ranges specified by the control stream—in this case representing the time-ranges suspected by the profile-detector as containing marmot calls. The inputs to `Sync4` are of type `Stream<SigSeg<int16>>` for the data streams and `Stream<bool,time,time>` for the control stream. Note that `int16` could be replaced by any type: the audio channels happen to carry `int16` samples, but `sync4` is generic.

Next, the synchronized windows of data from all four audio channels are processed by a beamforming algorithm. This is the second processing phase which makes the query multi-pass. The algorithm computes a direction-of-arrival (DOA) probability distribution, and enhances the input signal by combining phase-shifted versions of the four channels according to the most likely direction of arrival. The `beam` function returns a stream of two-tuples. We use a special binding syntax, “`beam<doa, enhanced> = ...`”, to give temporary names to the fields of these tuples. That is, `beam.doa` projects a stream of direction-of-arrivals, and `beam.enhanced` contains the

```
fun sync2 (ctrl_strm, S1, S2) {
  S3 = union3(S1, S2, ctrl_strm);
  S4 = iterate (tagged in S3) {
    state { acc1 = NullSeg;
           acc2 = NullSeg; }
    switch(tagged) {
      case Input1 v : acc1 := append(acc1, v);
      case Input2 v : acc2 := append(acc2, v);
      case Input3 <flag,t1,t2> :
        if (flag)
          then emit <subseg(acc1,t1,t2),
                    subseg(acc2,t1,t2)>;
        acc1 := subseg(acc1, t2, acc1.end);
        acc2 := subseg(acc2, t2, acc2.end);
    }
  }
  return S4;
}
```

Figure 2: A 2-input temporal synchronizer with error-handling omitted.

enhanced versions of the raw input data. Finally, this enhanced signal is fed into an algorithm that classifies the calls by type (male, female or juvenile), and, when possible, identifies individuals.

Defining custom operators: The queries we have seen thus far primarily wire together existing operators, rather than writing custom ones. Although we seek to provide a broad array of useful operators in the WaveScope libraries, it is likely that most applications will require custom operators that in turn may invoke user-defined POD functions.

In WaveScope, the user employs the `iterate` construct to construct a one-input, one-output operator that invokes a user-provided block of code on its input stream. For example, the following snippet constructs a generic aggregation operator parameterized by the `init`, `aggr`, and `out` function arguments.

```
fun build_aggr(S, init, aggr, out) {
  S2 = iterate (x in S) {
    state { acc = init(); }
    acc := aggr(acc, x);
    emit out(acc);
  }
  return S2;
}
```

Within the `iterate` construct, `state{}` is used to declare persistent (static) state, and an `emit` produces a value on the output stream (S2). In any given iteration, `emit` may be called zero or more times. The code inside a `iterate` construct is restricted to enable efficient compilation to a low-level imperative language (such as C). But WaveScript is expressive enough to enable us to write nearly all of our library operators directly in the language. It allows conditionals, for-loops, arrays, and handles SigSegs and timebases. In fact, the `sync4` primitive shown in the marmot-query above is written directly in WaveScript. In figure 3.2 is a definition for `sync2` (the two-input version of `sync4`) with error-handling omitted. `sync2` takes three inputs: one control stream of the same type as `sync4`, and two streams of SigSegs,

The `union` family of operators are the only primitive operators allowed to take multiple input streams. `Union3` takes three input streams and produces a value on its output stream whenever a value arrives on *any* input. The output value must be tagged for downstream operators to distinguish which input stream created the value. To accomplish this tagging in a type-safe way, WaveScope allows `tagged union` types (also called “variant records”, similar to type-safe enums) [22]. In particular, the variable `tagged` has three variants: `Input1`, `Input2`, and `Input3` corresponding to which channel the data arrived on. The `switch/case` construct uses pattern matching to dissect variants and their fields. The `Input1` and `Input2` cases carry SigSeg values from S1 and S2 respectively. These SigSegs are appended to their respective accumulators.⁴ The

⁴In the full version of `sync2`, exception handling would be required to

state of the `iterate` operator at any point in time is simply two `SigSegs` (`acc1` and `acc2`). `Sync2` accumulates both signals until it receives a message from the control stream. The control stream provides a time-range and a boolean flag indicating whether to discard or snapshot that range. The `subseg` function is used to crop the accumulators; it produces a new `SigSeg` that represents a sub-range of the input `SigSeg`.

By using a `union` operator together with a state-carrying `iterate` construct, it is possible to implement arbitrary synchronization policies (e.g., different buffering policies and temporal sensitivities). We have designed WaveScope in this way because there are a plethora of viable application specific synchronization policies. The WaveScript library, however, will include a suite of generally useful synchronization operators, such as `sync2`.

4. SYSTEM ARCHITECTURE

Queries in WaveScope are WaveScript programs that make use of a library of built-in signal processing functions. A WaveScope query, initially of the form we discussed in Section 3.2, must go through a number of stages to reach deployment and execution.

- **Preprocessor:** Eliminates syntactic sugar and type-checks the WaveScript source.
- **Expander:** Inlines all subquery-constructors and many POD functions, erasing abstraction boundaries and leaving a dataflow graph of basic operators—the query plan.
- **Optimizer:** Applies a number of inter- and intra-operator optimizations described in Section 5. (Mixed with expander phase.)
- **Compiler:** Generates the query plan, which will be in a low level imperative language (e.g., C) for efficiency. The query plan wires together functions compiled from each `iterate` operator in the original WaveScript query, and links against the WaveScope runtime library.
- **Runtime:** The runtime library for single-node execution consists principally of three modules: a **scheduler**, a **memory manager**, and a **timebase manager** (Section 4.2).

The final step in compilation entails compiling the query plan to machine code and executing the query. Since WaveScope will also support distributed query execution and network deployment (e.g., on a sensor network or PC cluster), there is an additional phase where query plans are disseminated to nodes in the network. (Section 4.3).

4.1 Compiled Query Plans

After the expander has produced an initial query plan, the optimizer performs multiple passes over the plan (Section 5) and generates an optimized query plan. The final query plan is an imperative program, corresponding to an Aurora-style directed graph that represents the flow of streaming data through the program. The only operators that survive to this point are `iterate`, `union`, and special source operators for each data source in the query.

- **iterate:** An `iterate` operator is the basic workhorse of a WaveScope query. Each `iterate` construct in a WaveScript program is compiled to an imperative procedure in the query plan, optionally with a piece of persistent state. This function is repeatedly invoked on each of the tuples in the operator's

handle the case where the `SigSegs` cannot be appended because they are not adjacent, or are not in the same timebase.

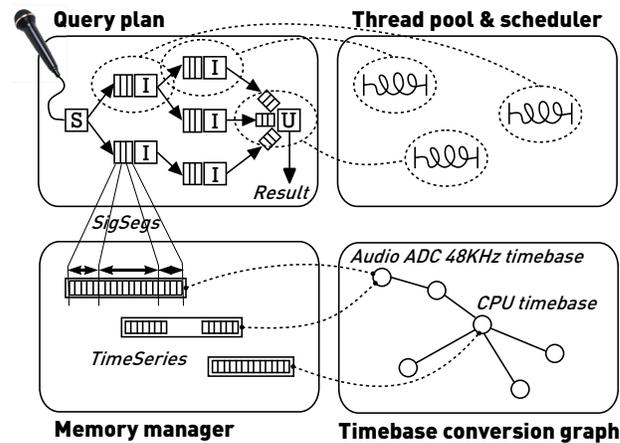


Figure 3: An example of a WaveScope query running in an execution engine of a node. Node types: S: source, I: iterate, U: union. A segment of one of the timeseries has been garbage-collected. Two of the timeseries share the same timebase.

input stream. The code inside the function may call an `emit` construct to output tuples, which will be routed to all of the operator's successors in the query plan.

- **union:** `Union` is the only operator in compiled query plans that takes multiple inputs and is the primitive upon which stream synchronization operations, such as `sync` and `zip` are built. Our previous example code for `sync2` illustrates its use.
- **source:** A source operator interfaces with I/O devices to produce signal data that feed to operators in the query plan. For example, the `AudioSource` in the marmot detection application (Section 3.2) continuously samples audio data from a sound card, and invokes the memory manager (Section 4.2.2) to window contiguous chunks of audio into `SigSegs`. These `SigSegs` are wrapped within tuples and emitted to operators that perform the actual processing. In the implementation, we envision that source operators will usually live in their own threads to avoid blocking on I/O.

Although the runtime engine only needs to handle the small collection of primitive operators described above (simplifying implementation), the WaveScript compiler and optimizer will recognize the special properties of many signal processing functions (e.g., FFT, IFFT, Convolve). This enables several high-level query optimizations, as discussed in Section 5.

4.2 Single-Node Runtime

Once deployed on a physical node, the WaveScope program runs within the execution engine. We provide an example of an audio processing query running within an engine of a single node in Figure 3. The figure demonstrates the three main modules of the engine: the scheduler, the memory manager, and the timebase manager. Below, we describe the functions of each of these subsystems in more detail.

4.2.1 Scheduler

The WaveScope scheduler chooses which operators in the query to run next, and provides the tuple passing mechanisms by which operators communicate. The scheduler maintains a thread pool, and assigns idle threads to available operators in the query graph. An operator is available if its input queues are non-empty and it is not already being executed by another thread.

A good scheduler should possess the following desirable properties:

Compact memory footprint to prevent thrashing, especially on embedded platforms with little RAM. Compactness includes reducing the overhead of allocating and deallocating memory, which can place a heavy burden on the operating system when WaveScope operates on high-rate streams.

Cache locality, which affects performance and leads to several important design tradeoffs. For example, executing an operator until its input queues are drained offers good instruction cache locality (operator code stays in cache) and data cache locality (operator’s internal state stays in cache). However, processing many input tuples in one sweep may lead to producing many output tuples, which could dramatically grow the size of the output queues. On the other hand, executing each operator one tuple at a time may incur more CPU cache misses, but could help lower the memory footprint.

Fairness helps avoid starving parts of the query plan. Devoting disproportionate time to the execution of any particular operator may result in a buildup in its output queues. Fairness is also important for multi-input operators (such as *Sync* and *Join*), where the skew in arrival rates of input streams may cause accumulation of data in internal buffers, and lead to delays in materializing the output tuples.

Because of possible differences in the sampling rates of sources and the disparity in the costs of computing different input streams to such operators, achieving fairness is an interesting scheduling problem which may require run-time profiling of each operator in the query plan. Since profiling of operators may be relatively expensive, our design may consider profiling the system and adjusting the schedule occasionally when rate skew is detected.

Scalability with the number of processors or CPU cores. A good scheduler should minimize the amount of thread synchronization and ensure affinity between operators and CPU cores to maintain cache residency of data and code.

One possible design we intend to investigate divides a query plan into “slices,” constraining the operators in each slice to only execute on a particular CPU core. The scheduler may run one thread per core, and schedule the order of execution within each slice separately, which avoids synchronization on centralized data structures. Determining a partitioning of the query plan that maximizes steady-state throughput is an interesting problem that we plan to investigate.

High-throughput tuple passing. The choice of the scheduler can also help to entirely eliminate queuing in the system. If the scheduler design gives up some flexibility in choosing the next operator to execute, and instead traverses the query plan *along* the operator connections (for example, in depth-first order), it can use direct operator-to-operator function calls, passing tuples by reference. Of course, assuring fairness in such schemes may be more difficult because of extra constraints on the order of execution.

4.2.2 Memory Management

The task of the memory manager is to provide a simple and efficient way for operators to create, access and garbage collect all signal data flowing through the query plan.

All signal data in a WaveScope query originates either from source operators in the query plan (*e.g.*, sensors or network sockets), or from intermediate operators that create new signals as output (*e.g.*, FFT). Conceptually, these operators invoke a memory manager API call to continuously append signal samples to a time series in the in-memory signal store. This API call, which we term

`create_sigsegs`, batches chunks of isochronous samples into SigSegs, which are returned to the application and in turn passed to subsequent operators in the query plan.

Operators in the query plan pass SigSegs between each other. Semantically, copying a SigSeg is equivalent to making a copy of the underlying signal data. However, to scale to high data rates and reduce in-memory copying overhead, we plan to use an implementation where SigSegs are passed by reference with copy-on-write semantics. In this implementation, the signal store automatically garbage collects signal data using standard reference counting.

SigSegs could overlap in range, so maintaining the correct reference counts requires some care. In addition, to ensure that a SigSeg always refers to valid data, WaveScript restricts how they can be created to three interfaces: `create_sigsegs`, which appends new samples to an existing timeseries; `append`, which creates a SigSeg by joining two existing adjacent SigSegs; and `subseg`, which creates a subrange of an existing SigSeg. The latter is useful in several applications (including the acoustic monitoring application described earlier) that identify and lookup an interesting time range of data for further processing.

4.2.3 Timebase Manager

Managing timing information corresponding to signal data is a common problem in signal processing applications. Signal processing operators typically process vectors of samples with sequence numbers, leaving the application developer to determine how to interpret those samples temporally. For example, a decimation filter that halves the input rate takes in $2N$ samples and outputs N samples—but the fact that the rates of the two vectors are different must be tracked separately by the application.

To address this problem, WaveScope introduces the concept of a *timebase*, a dynamic data structure that represents and maintains a mapping between sample sequence numbers and time units. As part of its metadata, a SigSeg specifies a timebase that defines what its sequence numbers mean. Based on input from signal source drivers and other WaveScope components, the timebase manager maintains a *conversion graph* (shown in Figure 3) that denotes which conversions are possible. In this graph, every node is a timebase, and an edge indicates the capability to convert from one timebase to another. The graph may contain cycles as well as redundant paths. Conversions may be composed along any path through the graph; when redundant paths exist, a weighted average of the results from each path may result in higher accuracy [18].

WaveScope will support several types of timebases:

- *Analytic timebases* represent abstract units, *e.g.*, Seconds (since the epoch), Hertz (for frequency domain signals), Meters, and Sequence Numbers—essentially, anything that could represent the independent variable (“x axis”) of a signal or time series. For example, a SigSeg of data sampled at one sample per second would have a timebase of Seconds, since each sequence number would correspond directly to that number of seconds.
- *Derived timebases* represent a linear relationship to an existing “parent” timebase. For example, a SigSeg sampled at 2 samples per second would derive its timebase as $2 \times$ Seconds.
- *Empirical timebases* represent a real free-running clock, *e.g.*, a CPU clock or a sample clock. Conversions associated with empirical timebases are empirically derived by collecting measurements that associate clock values in different timebases. Since clocks are locally linear, conversions to empirical timebases can generally be approximated by

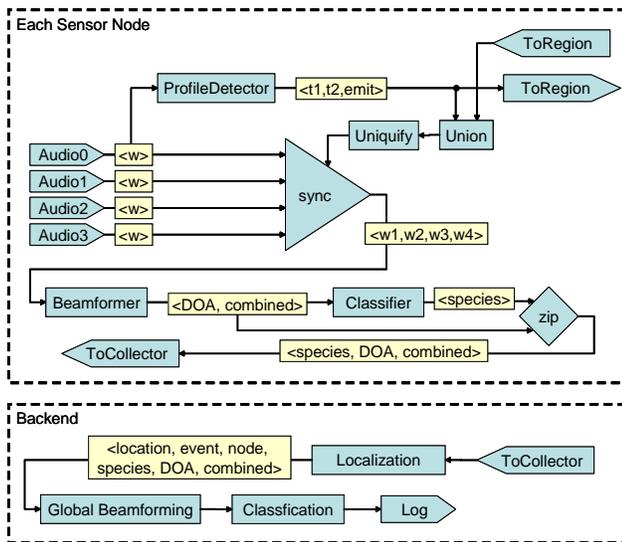


Figure 4: A distributed query for marmot localization.

piece-wise linear functions. For example, the CPU clock and a sound card’s sample clock are both empirical timebases. A relation between them can be derived by periodically observing the CPU time at which a particular sample was taken.

Using the timebase infrastructure, application developers can pass around SigSegs without worrying about keeping track of the temporal meaning of the samples, because this is captured automatically in the timebase graph. For example, a SigSeg that was transformed by a decimation filter can be related to the original signal by simply converting any sequence number in the decimated version to the corresponding sequence number in the original. Since conversion can be composed along any path through the timebase graph, this is even true across multiple composed decimations.

Empirical timebases can also be used to represent node-to-node time conversion, by correlating events observed in terms of both CPU clocks. By adding these node-to-node conversions to the timebase graph, sensor data measured on separate nodes can be related just as data measured on the same node.

4.3 Distributed Query Execution

Support for distributed queries is a key component of many WaveScope applications. For example, consider the marmot detection example from Figure 1. Suppose we now want to run that detection algorithm in embedded sensor nodes in a deployment and combine the results to localize the detected marmots via triangulation.

A workflow diagram for the *localization* step is shown in Figure 4. The components in the upper dotted box run in parallel on each sensor node in the system, detecting and classifying marmot calls. The components in the lower box run only on a centralized server, identifying events that are common across multiple nodes, triangulating the marmots, and re-classifying based on complete information. These computational domains are connected together through two simple network primitives, ToRegion and ToCollector.

ToRegion accepts tuples and multicasts them to all nodes in the specified region, where they are received back into the query plan. The target region might be specified as a particular set of nodes, a region in physical space, or by some other predicate. ToCollector accepts tuples and forwards them along a sink tree to a node identified as the collector. Note that when SigSegs are transmitted

over the network, the signal data they contain must be marshaled and transmitted, rather than only transmitting the metadata. This operation can be expensive, but there will be instances in which it can be optimized. For example, if a SigSeg is forwarded through ToRegion, and all of the nodes in the region will want the data, a multicast distribution tree might be the most efficient way to disseminate the signal data. Alternatively, if only a few of the target nodes will want to process the data, a lazy transmission scheme could be used, in which the metadata is disseminated, and the signal data is “pulled” only by those nodes that need it.

Although these simple primitives appear to be sufficient for this application, we expect that other primitives may arise as we explore new applications. Another common example of a distributed query is the case where each node maintains a local persistent store of data (*e.g.*, data that might be needed in the future but is not currently important enough to send back over the network). If the *sync* operator spills the raw signal data to disk, queries could be installed *post-facto* to perform further analysis of that data. Such a query could be installed by the system operator when something particularly interesting was observed, and the results sent back via ToCollector.

In Figure 4, the query defines the desired physical network mapping by explicitly specifying network links and where different parts of the query are executed. In principle, WaveScope could optimize the selection of these “cuts” based on a variety of metrics including processing and network capacity. We do not envision a completely autonomous cut selection algorithm in the near future; rather, we plan to provide a variety of profiling tools that can inform the user about where best to make the cut.

4.4 Querying Stored Data

In addition to handling streaming data, many WaveScope applications will need to query a pre-existing stored database, or historical data archived on secondary storage (*e.g.*, disk or flash memory). For instance, consider the query mentioned in Section 2 that tracks the positions of individual marmots as a function of time. In order to identify an individual marmot from an audio segment, the application needs to compare the segment with signatures of audio segments from previously seen marmots and determine the closest match, if any. This requires the ability to archive past events (like historical marmot detections) and query this archive at some point in the future.

We plan to provide two special WaveScope library functions that will support archiving and querying stored data declaratively:

- **DiskArchive**, which consumes tuples from its input stream and writes them to a named relational table on disk. The table name is a global, system-wide identifier.
- **DiskSource**, which reads tuples from a named relational table on disk and feeds them upstream. This operator is similar to the other source operators that were discussed earlier, but in addition may support indexing signal data, and pushing down predicates to allow efficient access to relevant regions of history.

Storing and retrieving large segments of signal data (both at a single node, and across a network of nodes) will pose several interesting research questions, including:

- Assuming that the goal is to support both efficient archiving and retrieval, what is the best way to store and represent signal data on disk? For instance, compressing signal data might be one strategy to save space and enable faster lookup by reducing disk bandwidth usage.

- For how long in the past should signal data be retained, and which data should be discarded first? Discard policies are likely to be application specific, but the library could include several policies for applications to choose from. For instance, some viable policies might include allowing applications to prioritize important data, or using progressively lossier encoding for older data (*e.g.*, using wavelets).
- What are the best ways to index signal data stored on disk so that lookup (*e.g.*, matching audio segments against signatures of previously seen marmots) is efficient? Traditional indexing techniques are unlikely to work for signal data [7, 19], and indexing strategies may also need to be application specific (*e.g.*, audio and images have different requirements). Therefore, the above mentioned operators will provide an interface to specify and make use of custom indexing policies.

We leave these, and related questions to future work.

5. OPTIMIZATIONS

A restricted query language and intermediate representation, and support for isochronous signal segments as first-class objects enable a wide range of optimizations in WaveScope. We illustrate some classes of optimizations below.

Query Plan Transformations: Optimizations such as predicate reordering (the compiler can determine that certain stateless `iterate` are merely predicates) and query merging are possible in WaveScope, similar to optimizations in other streaming database systems. For example, in the marmot application, queries for both marmot classification and localization involve a common preprocessing and signature extraction step. The query optimizer can statically analyze and merge portions of the two queries.

Another plan-level optimization is merging adjacent `iterate` operators, which has two benefits. First, fusing adjacent `iterate` operators enables optimizations across *multiple* user functions. The resulting fused code can expose optimizations such as loop fusion or common subexpression elimination. Second, because each `iterate` corresponds to a different operator in the wiring diagram, reducing the number of operators can reduce scheduling overhead (at the cost of decrease in potential parallelism).

Conversely, the optimizer can factor `iterate` operators to separate out processing on the input (or output) channel that does not depend on the operator’s state. This, in turn, offers further optimization possibilities. For example, consider multiple `iterate` operators that first apply an FFT to their input. The compiler can factor these FFTs out into their own, stateless, `iterate` operators. Then, if the original `iterates` are applied to the same stream, query merging can eliminate the redundant FFT computation. This scenario is common in signal processing. For example, a common way to implement speaker identification involves computing the aggregate power (area under the FFT) of overlapping windows in an audio signal in *both* preprocessing and feature extraction stages. WaveScope eliminates one redundant FFT computation per window, yielding significant savings. These optimizations become particularly important when the user is relying heavily upon high-level subquery constructors contained in libraries. They may not be aware of redundant computations within the abstractions they invoke.

Domain-specific Rewrite Optimizations: WaveScope will support a rule-based framework for *rewrite optimizations* that rely on domain-specific properties of particular signal processing library functions. This framework is similar to previous work on extensible relational optimizers [15, 26]. Optimization rules are written in

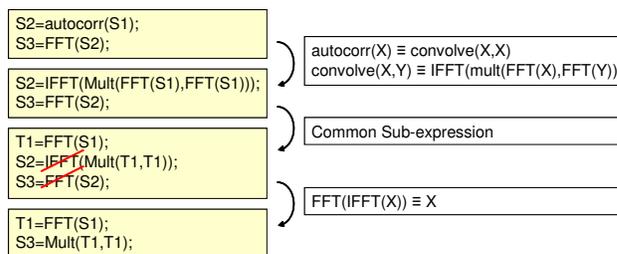


Figure 5: Diagram illustrating rewrite optimizations in WaveScope.

terms of named operators, so that when a new operator is added to the WaveScope library, the programmer can also add rewrite rules for that operator (*e.g.*, a typical rule might express that IFFT is the inverse for FFT).

Some signal processing operators permit complex rewrite optimizations. For example, the denial-of-service detection scheme described in [17] analyzes the power spectrum of network packet counts to classify attacks. This technique involves an *autocorrelation* operation, followed by power spectrum analysis to determine if activity at a particular frequency is unusual. The standard way to compute the autocorrelation is to convolve a signal with itself, and the standard way to compute a power spectrum is to compute the integral of the FFT of a signal. To optimize this query, the optimizer takes advantage of several signal processing identities, which can be specified as rules to the optimizer as follows:

1. $\text{autocorrelate}(S) = \text{convolve}(S, S)$
2. $\text{convolve}(X, Y) = \text{IFFT}(\text{FFT}(X) * \text{FFT}(Y))$
3. $\text{FFT}(\text{IFTT}(X)) = X$

Applying the rules reduces the above sequence of operations to finding the FFT of the packet count sequence and squaring it, as shown in Figure 5. This requires only $\Theta(n \log n)$ operations—faster than the original implementation which performs $\Theta(n^2)$ operations, where n is the length of the packet-count time-series.

6. RELATED WORK

WaveScope is related to streaming data management systems [1–3, 8, 9, 23]. The key differentiating feature of WaveScope is that it provides a single language and a unified framework that integrates both stream and signal processing operations. This also distinguishes WaveScope from existing math, scientific and engineering tools like MATLAB, Simulink and LabVIEW [4–6] that have excellent support for signal processing functions but unsatisfactory support for event-stream operations.

There is a large body of related work on temporal and sequence databases (databases containing time-varying data) that provide a context for our research, including SEQ and Gigascope [10, 27]. These systems do view time-series data as first class objects, but are targeted at simpler trend analysis queries, as opposed to our effort, which is focused on supporting more complex signal processing operations.

The compiler technology in WaveScope is related to the authors’ previous work on the Regiment programming language [25]. In fact, many programming languages have been proposed for stream-processing, and many general purpose languages include streaming features [29]. For example, StreamIt system [30] is a programming language and compiler that targets high performance streaming applications. StreamIt’s compiler backend targets architectures ranging from highly-parallel research platforms to commodity uniprocessors. StreamIt, however, is based on a synchronous data-flow

model where the data production and consumption patterns of operators are static and known at compile time. WaveScope, in contrast, is targeted at the asynchronous data-flow world where operators are less predictable and produce data at varying rates. Instead, our proposal exploits isochrony to leverage some of the benefits of synchronous data-flow.

Ptolemy II [21] is widely-used data-flow system for modeling and simulation of signal processing applications. Unlike WaveScope, it is not focused on efficiency or on providing a high-level, optimizable programming language.

There has been work on individual signal processing applications in the sensor network community [14, 28], but these systems are typically built from scratch, requiring many months of effort to implement. The Acoustic ENSBox [14] is a development platform specifically designed to support distributed signal processing. While this has significantly reduced application development time, it requires error-prone C programming and lacks an interactive, query-like interface.

Time synchronization is a critical aspect of distributed sensing applications, and a fertile topic in the sensor network community [11–13, 18, 20]. The WaveScope timebase construct is similar to those used in the implementation of Reference Broadcast Synchronization [12] used in the Acoustic ENSBox system [14]. However, in WaveScope these principles are more deeply integrated, providing a more natural interface.

7. CONCLUSION

Today, developers of many streaming applications that require even the simplest forms of signal processing face a devil’s choice: either write significant amounts of custom code in a language like Java or C++, or use a stream processing engine together with an external system such as Simulink (MATLAB) or LabVIEW and move data back and forth between the two. Both approaches have a number of obvious flaws. In this position paper, we have sought to re-address these problems with WaveScope, a data stream management system that combines even stream and signal processing operations. We believe—and hope to demonstrate soon—that WaveScope will be significantly faster and more usable than current approaches.

8. ACKNOWLEDGMENTS

We thank Kyle Jamieson for his contributions to the ideas described in this paper. This work was supported by the National Science Foundation under Award Numbers CNS-0520032 and CNS-0205445 and by a National Science Foundation Fellowship.

9. REFERENCES

- [1] <http://www.streambase.com/>.
- [2] <http://www.coral8.com/>.
- [3] <http://www.aleri.com/>.
- [4] <http://www.mathworks.com/products/matlab/>.
- [5] <http://www.mathworks.com/products/simulink/>.
- [6] <http://www.ni.com/labview/>.
- [7] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient Similarity Search In Sequence Databases. In *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, Chicago, Illinois, 1993. Springer Verlag.
- [8] 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.
- [9] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Suresh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [10] C. Cranor, T. Johnson, O. Spatschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, 2003.
- [11] Jeremy Elson and Deborah Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing*, April 2001.
- [12] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI*, 2002.
- [13] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2003.
- [14] Lewis Girod, Martin Lukac, Vlad Trifa, and Deborah Estrin. The design and implementation of a self-calibrating acoustic sensing system. In *SenSys*, 2006.
- [15] Goetz Graefe. *Rule-based query optimization in extensible database systems*. PhD thesis, 1987.
- [16] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
- [17] Alefiya Hussian, John Heidemann, and Christos Papadopoulos. A Framework for Classifying Denial of Service Attacks. In *Proceedings of ACM SIGCOMM*, August 2003.
- [18] Richard Karp, Jeremy Elson, Deborah Estrin, and Scott Schenker. Optimal and global time synchronization in sensor networks. Technical Report CENS-0012, Center for Embedded Networked Sensing, 2003.
- [19] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD*, 1997.
- [20] B. Kusy, P. Dutta, P. Levis, M. Mar, A. Ledeczki, and D. Culler. Elapsed time on arrival: A simple and versatile primitive for canonical time synchronization services, in press, 2006.
- [21] Edward A. Leet. Overview of the ptolemy project. Technical Report Technical Memorandum No. UCB/ERL M03/25, UC Berkeley, 2003.
- [22] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.
- [23] R. Motwani, J. Window, A. Arasu, B. Babcock, S. Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation and resource management in a data stream management system. In *CIDR*, 2003.
- [24] D.D. Nanayakkara and D.T. Blumstein. Defining yellow-bellied marmot social groups using association indices. *Oecologia Montana*. In Press.
- [25] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.
- [26] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD*, 1992.
- [27] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database systems. In *VLDB*, 1996.

- [28] G Simon, M Maróti, À Lédeczi, G Balogh, B Kusy, A Nádas, G Pap, J Sallai, and K Frampton. Sensor network-based countersniper system. In *SenSys*, 2004.
- [29] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
- [30] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *ICCC*, April 2002.

APPENDIX: MARMOT PHASE 1 FILTER

Below is the full source for the first pass of marmot-detection application. This pass performs the fast, real-time prefiltering of the audio-stream, looking for time windows containing potential marmot calls. Note that before each top-level function, or variable we have included a type annotation of the form “name : type”. (Function types are written using arrows (->). These are optional, but provide useful documentation. Further, they may be used to assert more restrictive types than those that would be inferred by the type-inferencer. For example, we assert that marmotScore expects complex numbers rather than any numeric type.

```
// We use rewindow, sinc4, cnorm, hanning, and fft
// from the WaveScript standard library.
include "stdlib.ws";

//=====
// Main query:

Ch0, Ch1, Ch2, Ch3 : Stream< SigSeg<int16> >
Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

// Invoke profileDetector with appropriate scoring
// function and settings.
control : Stream<bool, int, int>
control = profileDetect(Ch0, marmotScore,
    <64,192>, <16.0, 0.999, 40, 2400, 48000>);

// Use the control stream to extract actual data windows.
datawindows : Stream<SigSeg<int16>, SigSeg<int16>,
    SigSeg<int16>, SigSeg<int16>>
datawindows = sinc4(control, Ch0, Ch1, Ch2, Ch4);

// Final query result:
return datawindows;

//=====
// Application POD functions and subquery-constructors

// POD function: Compute a score for a time-window of audio data.
marmotScore : SigSeg<complex> -> Stream<float>
fun marmotScore(freqs) {
    // Return the complex magnitude for energy
    // in particular hardcoded frequency bins.
    return cnorm(freqs[6] + freqs[7] + freqs[8] + freqs[9]);
}

// The main detection function.
profileDetect : Stream<SigSeg<int16>>, (SigSeg<complex> -> float),
    <int,int>, <float,float,int,int,int>
    -> Stream<bool, int, int>

fun profileDetect(S, scorefun, <winsize,step>, threshsettings) {
    // Window input stream, ensuring that we will hit each event
    wins = rewindow(S, winsize, step);
    // Take a hanning window and convert to frequency domain.
    scores : Stream< float >
    scores = iterate(w in hanning(wins)) {
        // Compute frequency decomposition
        // Note: our fft produces complex numbers.
        freq = fft(w);
        // Score each frequency-domain window
        emit scorefun(freq);
    };
    // Associate each original window with its score.
    withscores : Stream<float, SigSeg<int16>>
```

```
withscores = zip2(scores, wins);
// Find time-ranges where scores are above threshold.
// threshFilter returns <bool, starttime, endtime> tuples.
return threshFilter(withscores, threshsettings);
}

// Aggregate a stream of scored windows into labeled, contiguous
// time ranges that are either above or below threshold.
threshFilter : Stream<float, SigSeg<int16>>,
    <float, float, int, int, int>
    -> Stream<bool,int,int>

fun threshFilter(scorestrm, <hi_thresh, alpha, refract_interval,
    padding, max_run_length>) {

    // Constants
    startup_init = log(0.75)/log(alpha);

    iterate((score,win) in scorestrm) {
        state {
            thresh_value = 0.0;
            trigger = false; trigger_value = 0.0;
            smoothed_mean = 0.0; smoothed_var = 0.0;
            startind = 0; refract = 0;
            startup = startup_init;
        }
        if trigger then {
            if win.end - startind > max_run_length then {
                print("Detection length exceeded maximum of " ++
                    show(max_run_length) ++
                    ", re-estimating noise\n");
                // Reset all state variables:
                thresh_value := 0.0;
                trigger := false;
                smoothed_mean := 0.0;
                smoothed_var := 0.0;
                startind := 0;
                trigger_value := 0.0;
                startup := startup_init;
                refract := 0;
            }
            // Over threshold; set refractory.
            if score > thresh_value then {
                refract := refract_interval;
            } else if refract > 0 then {
                // refractory counting down
                refract := refract - 1;
            } else {
                // Untriggering!
                trigger := false;
                emit <true, // yes, snapshot this
                    startind - padding, // start sample
                    win.end + padding>; // end sample
                startind := 0;
            }
        } else { // If we are not triggering...
            // Compute the new threshold.
            thresh = int_to_float(hi_thresh) *
                sqrt(smoothed_var) + smoothed_mean;

            if startup == 0 && score > thresh then {
                // We're over threshold and not in startup
                // period (noise estimation period).
                trigger := true;
                refract := refract_interval;
                thresh_value := thresh;
                startind := win.start;
                trigger_value := score;
            } else {
                // Otherwise, update the smoothing filters.
                smoothed_mean := score * (1.0 - alpha) +
                    smoothed_mean * alpha;
                delt = score - smoothed_mean;
                smoothed_var := (delt * delt) * (1.0 - alpha) +
                    smoothed_var * alpha;
            }
            // Count down the startup phase.
            if startup > 0 then startup := startup - 1;

            // We know there are no events here,
            // so we free from time 0 to the present.
            emit <false, 0, max(0, win.end - samples_padding)>;
        }
    }
}
```