XStream: a Signal-Oriented Data Stream Management System

Lewis Girod [1], Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan
Hari Balakrishnan, Samuel Madden

Computer Science and Artificial Intelligence Laboratory, MIT
32 Vassar St, Cambridge, MA, 02139, USA
[1]ldgirod@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 now widely deployed in many industrial, civil engineering, scientific, networking, and medical applications. In aggregate, these sensors easily generate several million samples per second that must be processed within milliseconds or seconds. The computation required includes both signal processing and event stream processing. XStream is a stream processing system for such applications.**

XStream introduces a new data type, the *signal segment*, which allows applications to manipulate isochronous (regularly spaced in time) collections of sensor samples more conveniently and efficiently than the asynchronous representation used in previous work. XStream includes a memory manager and scheduler optimizations tuned for processing signal segments at high speeds. In benchmark comparisons, we show that XStream outperforms a leading commercial stream processing system by more than three orders of magnitude. On one application, the commercial system processed 72.7 Ksamples/sec, while XStream processed 97.6 Msamples/sec.

I. Introduction

XStream is a stream processing system that is designed to efficiently support high-rate signal processing applications. The motivating applications for this system come from a variety of industrial, scientific, and engineering domains. These applications use embedded vibration, seismic, pressure, magnetic, acoustic, network, and medical sensors to sample data that are on the order of millions of samples per second in aggregate. Such high-rate sample streams need to be processed and analyzed within milliseconds or at most seconds of being produced, using a mix of event stream and signal processing operations. Example operations include selecting out time ranges and samples of interest, applying signal processing transforms to them, correlating them with other signals, time-aligning them with other signals, and applying a variety of application-specific aggregate operations on the signals.

Existing stream processing engines (SPEs) [1], [2], [3], [4], [5], [6] provide some of the features needed to express these applications, but suffer from two limitations. First, they do not scale to the rates that many signal processing applications need to support (often several million samples per second) on conventional PC-class machines. Second, existing SPEs do not provide an operator set that is adequate for signal processing—for example, operations like FFTs and Wavelets are not provided, and must be implemented as user-defined functions in external languages.

To understand these two limitations more concretely, we discuss a synthetic benchmark, StatFilter. This query computes statistics over streams of audio data that have been regularly sampled in time (are *isochronous*). This benchmark consists of two sequentially connected filter operators. The first operator calculates the standard deviation of the last 4096 samples, and passes the window onward if it is greater than a threshold α. The second operator works similarly, passing only windows whose average value is less than β.

Let's consider how such an application might be implemented in a conventional SPE. First, the streams carry one tuple for each input sample, with `<sample,timestamp>` as the schema. The first filter collects the next 4096 tuples in a window, and must either stride through the data or strip the timestamps out to compute the standard deviation. For each window that satisfies the predicate over standard deviation, the tuples in that window are passed, one at a time, as an output stream to the second filter. The second filter again collects a window of 4096 tuples and conditionally outputs tuples from windows that satisfy the average predicate.

In contrast, the streams in the XStream implementation are tuples containing chunks of sample data called *signal segments*, or SigSegs. Rather than performing per-operator windowing, the SigSegs are formatted into blocks of 4096 samples by a special re-window operator. Timing information is summarized as part of each SigSeg data type. Per tuple timestamps are not necessary because all samples are separated by the same amount of time. Each filter is invoked once per tuple, but since each tuple contains a SigSeg, it processes the entire block at once. When a SigSeg is passed to the next operator, the entire SigSeg of tuples is passed by reference, reducing overhead further.

XStream achieves a 1000× performance gain for this type of query, relative to a commercial SPE: the commercial system processed 72.7 Ksamples/sec, while XStream processed 97.6 Msamples/sec. This gain is the result of several design choices that enhance XStream's performance for signal-oriented queries, in particular an ADT designed to efficiently manipulate signal data, a set of operators, a memory manager, and a scheduler optimizes for that ADT. We expand on these results in Section IV.

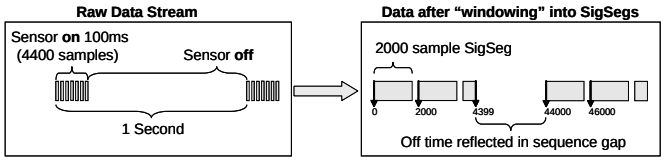

Fig. 1. Example illustrating how a raw data stream, with 100 ms of data every second, sampled at 44 KHz, is packaged into 2000-element SigSegs.

In a recent position paper at CIDR [7], we outlined the basic system architecture and functional query language of a system that makes it easy to express signal-oriented streaming applications. In this paper, we describe the details of XStream, focusing on the features that achieve high performance. These performance gains result from design choices arising from three important observations:

1) Isochronous data is the common case: We have observed that many sensor-based streaming applications produce high-rate data isochronously (*e.g.*, once per microsecond). Representing and processing these data streams as timestamped tuples is inefficient, since the timestamp of each sample is implicit from its position in the stream. Many block-oriented signal processing operators such as FFTs perform optimally when operating on densely packed arrays. For such operators, processing data interleaved with timestamps presents a major hurdle. Additionally, conventional streaming operators such as temporal joins suffer when working at the granularity of individual samples, because of the sheer number of comparisons that must be made. To address this, we define a new abstract data type which we call a *signal segment*, or SigSeg. A SigSeg encapsulates a finite sequence of isochronous samples into an array-like data structure with associated timing metadata, as shown in Fig. 1. XStream applications manipulate SigSegs as first-class objects.

2) Scheduler and tuple passing overhead is a bottleneck: Left unchecked, scheduling and tuple passing overhead can easily dwarf the costs of the actual computations performed on the data stream. We demonstrate this in Table I (Section IV-A), in which our performance benchmarks show that engine overhead is often multiple orders of magnitude greater than the actual cost of data processing operators, even in a *commercial* SPE. We design XStream's engine to minimize the engine overhead. The XStream memory manager takes advantage of isochrony to manage SigSegs efficiently, allowing applications to pass, append, and subdivide SigSegs with low overhead. In particular, it avoids expensive copying operations whenever possible, while supporting dynamic manipulation of signal data. In addition, passing data in SigSegs rather than individual tuples substantially reduces the context switching overhead in the scheduler. The XStream runtime also includes a novel design for a "depth-first" scheduler that dispatches tuples to operators in a (mostly) depth-first traversal of the wiring diagram, avoiding expensive scheduling decisions.

3) A unified query language enables whole-program optimization: Implementing the SigSeg ADT and associated signal processing functions in existing SPEs relies heavily on the use of user-defined functions (UDFs). However, because these UDFs are written and compiled separately from queries, query optimizers cannot "see inside" UDFs. If, on the other hand, we write the entire application in a single language, we gain the ability to perform global optimizations. For example, the compiler can fuse UDFs or specialize them to the specific contexts in which they are invoked. Such a framework makes it easy to ensure type-safety, and removes the necessity for awkward type conversions and parameter marshaling when calling UDFs.

This paper focuses on the first two points mentioned above. An overview of the WaveScript language was given in a position paper [7]; a detailed evaluation of the compiler optimizations and programmer productivity benefits are subjects of on-going work.

A. Contributions

This paper describes and evaluates the implementation of XStream. It makes the following contributions:

- The SigSeg ADT, which enables high performance manipulation of signal-oriented data, and a memory manager that supports (large) SigSegs efficiently.
- A scheduler that dispatches tuples to operators in a (mostly) depth-first traversal of the query plan, avoiding expensive scheduling decisions.
- A detailed experimental evaluation of several microbenchmarks analyzing XStream, including a comparison to a commercial SPE.

We begin with a few motivating applications, and then describe the implementation of SigSegs and the XStream engine in Section III. We evaluate XStream by presenting microbenchmarks of engine performance in Section III, and a detailed evaluation of the XStream engine with benchmark comparisons to a commercial SPE in Section IV.

II. Motivating Applications

To motivate our design decisions, we summarize three applications that we have implemented using XStream: pipeline leak detection, network monitoring, and acoustic localization. All three applications process high-rate input data (many kilohertz to megahertz), whose aggregate in a deployment would be several million samples per second. Although space constraints preclude covering these implementations in detail, we quote performance numbers from our XStream-based implementations, using the benchmarking setup described in Section IV.

1) Pipeline Monitoring: Pipeline monitoring uses vibration and/or acoustic sensors deployed along water pipelines to detect incipient ruptures. Cracks in pipes cause small disruptions that can be detected at multiple sensors to determine the approximate locations of leaks. When deployed in the water distribution networks underneath cities, these technologies might save millions of dollars a year by preventing leaks and outages [8], [9]. Typical data rates are tens to hundreds of

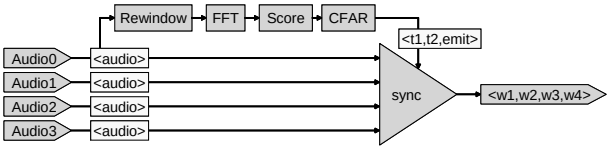

Fig. 2. DETECTAUDIO workflow.

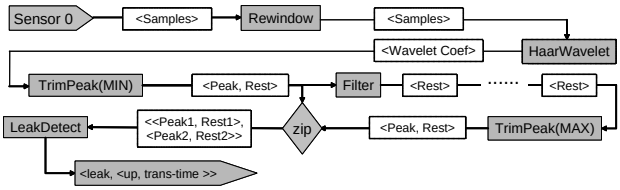

Fig. 3. PIPELINE workflow.

kilohertz from each sensor; deployments typically consist of about ten sensors each. Wavelet analysis is used to identify signatures characteristic of leak reflections, which are located based on time difference of arrivals. Our XStream implementation of processed vibration data at 4.3 Msamples/sec.

2) Network Monitoring: Recently, several groups have applied signal processing methods to network monitoring [10], [11]. In our application we analyze wireless bit errors at different receivers in an 802.11g network, in an effort to reconstruct the original packets from multiple corrupted copies. In the analysis, we align the (possibly corrupted) bit-streams from multiple observations of each packet, and compute pairwise correlations. The most correlated receivers for each source are identified by clustering, and receivers from different clusters are selected to maximize the likelihood of reconstructing corrupted packets. Our XStream implementation processed a packet stream at 241.5 Kpackets/sec.

3) Acoustic Localization: In a typical acoustic localization application, several small arrays of omni-directional microphones are placed in the environment surrounding a target. Typical data rates are 48 kSamples/sec from each channel, with each array hosting 4 or more channels. These systems typically run a low-cost local detection stage, followed by a more expensive multi-node localization and classification stage. These systems apply to a number of domains, including animal tracking [12], [13], signal enhancement for improved event recognition [14], and military applications such as sniper identification [15]. Our XStream implementation processed 4 channel audio data at 7.5 Msamples/sec.

A. DETECTAUDIO *and* PIPELINE *Benchmarks*

To provide a more realistic performance assessment for XStream, we developed two benchmarks based on our motivating applications: DETECTAUDIO and PIPELINE. In this section we describe these algorithms in more detail.

Fig. 2 shows a block diagram of our XStream implementation. The sensor inputs consist of four independent audio channels, receiving 16-bit audio data from a microphone array at 48 KHz. The application passes one of the channels to a streaming event detector. This data stream is windowed into blocks of 32 samples each, and each window is passed to a Fast Fourier Transform (FFT) to compute a frequency map. The algorithm then calculates a "detection score" by computing a weighted sum of the magnitudes of the frequency bins.

The resulting summary signal is passed to a *Constant False Alarm Rate (CFAR)* algorithm [14]. The CFAR algorithm assumes a Gaussian noise model $N(\mu, \sigma^2)$ and computes online estimates of the model parameters $\bar{\mu}$ and $\bar{\sigma}$. Detection is triggered whenever the summary signal exceeds a threshold β standard deviations above the current noise estimate.

Whenever an event is detected, the time range corresponding to the event is passed as the control input to a `sync` operator. All four channels of raw signal data are fed into the inputs of the `sync` operator; for every detection range submitted on the control input, `sync` emits a tuple containing a SigSeg from each input channel corresponding to the requested detection range.

Fig. 3 shows a diagram of the PIPELINE benchmark. Most of the work is done by the `haarwavelet` operator, which finds the energy in particular frequency bands. PIPELINE combines several standard WaveScript operators (*e.g.*, `rewindow`, `zip`, `haarwavelet` and `trimpeak`) with `LeakDetect`, a custom operator that determines the presence of a leak in the pipeline.

III. XSTREAM IMPLEMENTATION

Fig. 4 shows the XStream architecture. This paper focuses on the right-hand side of the diagram: the *XStream engine* that efficiently implements the SigSeg ADT and schedules a compiled query plan with low overhead. The implementation of SigSegs and the scheduler are described in detail in the following sections.

A. The SigSeg ADT

As in most streaming databases, all data in XStream is represented as streams of tuples with pass-by-value semantics. These streams are used to pass data between XStream operators; typically a data source operator creates a stream as it reads data from sensors, the network, or a file, and feeds it into a query plan. Individual operators process this stream and in turn generate new streams.

The main addition to the data model is that tuples may contain SigSegs, which are used to pass signal data between operators. Conceptually, SigSegs provide an array-like interface that provides access to a subsection of a signal. SigSegs also conform to a pass-by-value semantics, although (as we describe in Section III-B), for efficiency reasons the implementation passes them by reference with copy-on-write. Elements of SigSegs are assumed to be regularly spaced in time, though individual tuples in a stream may arrive completely asynchronously. (Hence, the data model of a stream processor like Aurora [1] or STREAM [2]—which have no SigSegs—can be fully modeled as a stream of XStream tuples.) Each SigSeg also contains a reference to a *timebase*, an object that specifies the rate and phase of the signal. Tuples in XStream do

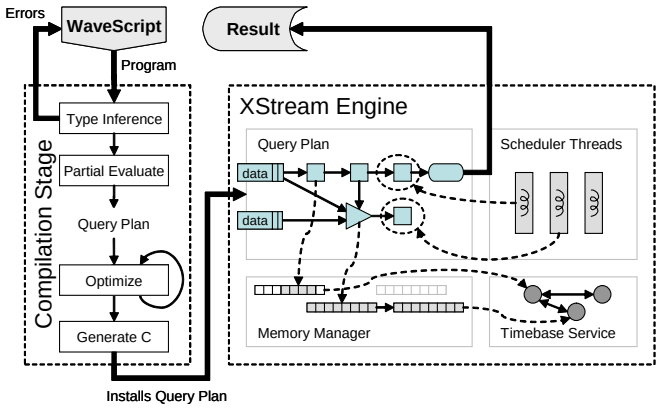

Fig. 4. The XStream architecture.

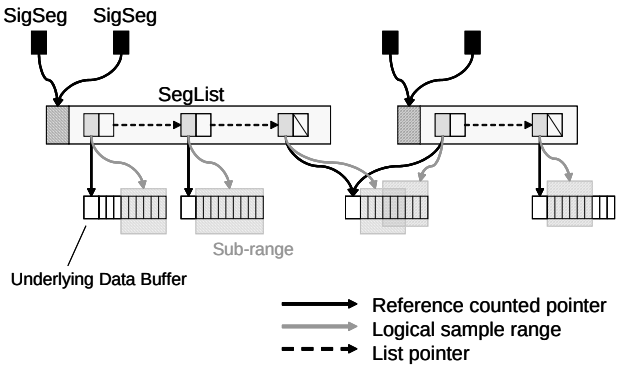

Fig. 5. The `Refcount-Lazy` data structure.

not intrinsically contain timestamps, though many application schemas do include timestamps.

SigSegs make it possible to pass windows of data between operators as first class objects. This, combined with the isochrony of SigSegs, offers three benefits:

- First, since SigSegs carry windows of data between operators, individual operators are *not* required to define their own window at their input.
- Second, operations that buffer data or change the windowing of data are very efficient. The `rewindow` operator uses copy-free subset and append operations to transform the windowing of a stream of SigSegs.
- Third, since SigSegs are isochronous, storage overhead is dramatically reduced by eliminating explicit per-sample timestamps, and enabling operators to index into a SigSeg by timestamp in constant time.

To achieve this, SigSegs add three additional methods to the standard array API:

- **subseg**($sigseg$, $start$, len): returns a new SigSeg representing a sub-range of the input sigseg.
- **append**($sigseg$, $sigseg$): returns a new SigSeg representing the union of two adjacent SigSegs.
- **timebase**($sigseg$): returns the SigSeg's timebase, used to relate a signal index to a timestamp.

B. Memory Manager and SigSegs

The principal goal of the XStream memory manager is an efficient implementation of the SigSeg API capable of scaling to data rates of millions of samples per second. In particular, the memory manager must provide efficient ways to:

- *Create* SigSegs at a data source (*e.g.*, a microphone or pressure sensor) or from intermediate results of computation (*e.g.*, output of an FFT).
- *Pass* SigSegs between operators in the query plan.
- *Manipulate* SigSegs using `append` and `subseg`.
- *Materialize* SigSegs into contiguous data buffers to support algorithms with non-sequential access patterns.

1) Design alternatives and tradeoffs: To motivate our design choices in this implementation, we present four successive versions, each making an incremental change over the previous, and yielding a performance improvement.

We first consider `CopyAlways`, a naive implementation in which SigSegs contain a copy of the signal data. In this version, SigSegs are passed between operators by copying, and `append` and `subseg` are also implemented by copying. This strategy is a good basis for comparison, with simple semantics even in the presence of concurrency.

To establish the overhead of copying, we compared the cost of creating a copy of a SigSeg to two common signal processing operations: FFT and $\mathbf{X} \cdot \mathbf{X}$. We found that while copy cost is insignificant compared to heavier algorithms such as a 256 point FFT, the cost of making a copy consistently dwarfs $\mathbf{X} \cdot \mathbf{X}$ by a factor of 10. This suggests that a lighter-weight method of passing SigSegs is required to support fine-grained modularity in the query plan.

Our second implementation, `RefCount`, is a straightforward extension to `CopyAlways` that stores signal data in a reference-counted buffer. `RefCount` implements standard copy-on-write semantics: mutating a SigSeg normally requires copying, but can be optimized to be in-place if no other SigSegs share the same data block. The `append` operation poses more of a challenge. We consider three possible implementations of `append`:

- `RefCount-Copy`, which maintains a contiguous buffer on `append` by allocating a new block and copying.
- `RefCount-Realloc`, which uses `realloc()` to extend the allocation of the "earlier" buffer without copying, and then appends data from the "later" buffer.
- `RefCount-Lazy`, which relaxes the requirement that the underlying buffers are contiguous, instead taking a lazy approach to materialization. Each SigSeg maintains a list of reference counted pointers to data blocks sorted by time. In-place destructive `appends` are fast since they simply splice lists.

Our experiments with the acoustic monitoring application quickly ruled out the first design, which suffered an unacceptable slowdown from appends within the `sync` operator (Fig. 6). The second design also suffers from the overhead of

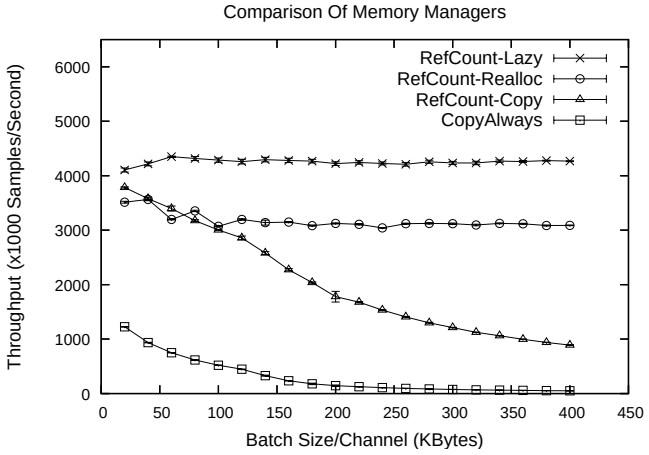

Fig. 6. Comparison of memory managers as batch size increases. Some error bars are not visible due to low variance.

a system call to `realloc()` for each append. More problematically, since reallocating a block can alter its memory address and SigSegs in different threads can share blocks, this requires locking on every access to a SigSeg.

We finally converge on `RefCount-Lazy`, which offers quick appends and avoids unnecessary materialization. As shown in Fig.5, this method uses two levels of reference counting. A SigSeg is a reference counted pointer to a *seglist*, consisting of SigSeg metadata (start time and length) and a linked list of reference counted pointers to ranges of underlying data buffers.

`RefCount-Lazy` makes an important design choice by deferring the creation of contiguous data buffers until it is necessary. When a non-sequential access pattern is required, the SigSeg must be materialized into a single contiguous buffer—but in the workloads we have examined this is rare in comparison to calls to `append`. Sequential iteration can be achieved efficiently without materialization using an iterator abstraction. Furthermore, in many workloads (*e.g.*, the `sync` operator in DETECTAUDIO) data is temporarily buffered using `append`, but an insignificant fraction is ever materialized.

2) Evaluation: We evaluate our memory manager using the DETECTAUDIO benchmark described in Section II-A. Our benchmark uses the same machine and method described in Section IV (maximum offered load, 2.8 GHz Xeon, 1 MB L2), except that data is not pushed all at once, but in fixed size *batches*, to control the amount of buffering in `sync`.

Fig. 6 shows the system throughput in kilosamples per second as a function of batch size (per audio channel). We observe that `CopyAlways` and `RefCount-Copy` perform poorly as batch size increases. This is because each `append` to the end of the accumulator in `sync` copies the appended data, consuming time proportional to the accumulator size and hence the number of previous appends. The total time for $\Theta(n)$ appends is therefore $\Theta(n^2)$. This slowdown is unacceptable for realistic workloads: we observe that the throughput begins to drop substantially in both implementations beyond a batch size of 200 KB, which corresponds to buffering roughly a couple of seconds' worth of 48 KHz audio.

While `RefCount-Realloc` does not perform as badly (its memory operations take linear time for `sync`), the overhead of `realloc()` and locking results in a larger constant factor compared to `RefCount-Lazy`. `RefCount-Lazy` performs the best, with throughput up to four times higher than the simple reference-counting approach, and an order of magnitude better than naive copying. Encouragingly, the lazy approach also has an almost flat performance curve that scales extremely well to high data rates, unlike the other strategies that exhibit decaying throughput with higher offered load.

C. Scheduler

The XStream scheduler determines the order of execution of operators in the query plan. This section describes various alternative designs, including some previously proposed, to show how the choice of scheduler can dramatically affect performance.

1) Design goals: A good scheduler must allow XStream applications to achieve high throughput by keeping overheads to a minimum. These overheads include the cost of switching from one operator to another, passing data between them, and queuing overheads. Scheduling at the coarse granularity of SigSegs helps reduce the scheduling overhead, relative to scheduling at the level of individual samples. Though bulky SigSegs are passed by reference, memory for *tuples* (which contain references to SigSegs) must still be allocated and deallocated as they get copied between operators. A good scheduler should also minimize the overall memory footprint of the running application; for example, a scheduler that relies on queues before every operator is likely to consume more memory than one that does not. Memory consumption is important because being able to hold the "working set" in the cache is crucial for high-rate processing. Moreover, schedulers that cause memory consumption to vary dramatically cause applications to run slower because resizing the heap incurs significant overhead.

2) Design alternatives and tradeoffs: We investigate three schedulers here: `FIFO-Slice`, `RTC` ("run-to-completion") and `DF` ("depth-first"). `FIFO-Slice` maintains per-operator queues of data waiting to be processed by the corresponding operator. Every `timeSliceDur` seconds, the scheduler picks the least recently dequeued input queue, determines which operator it belongs to, and processes as many tuples from that queue is possible until `timeSliceDur` expires or the queue becomes empty. If `timeSliceDur` is set to 0, then `FIFO-Slice` schedules one tuple per operator at a time (which of course incurs high scheduling overhead). We use the term `FIFO` for this special case.

The `RTC` scheduler shifts the granularity of processing from queues to operators. Here, when an operator is scheduled, *all* input queues feeding that operator are drained and the tuples are `iterated` upon by the operator before the next operator is scheduled. In addition, the `RTC` scheduler determines if the operator has `emit`ted any tuples. If so, it chooses one of the

successor operators to execute immediately afterwards, while the other successor operators are enqueued for execution in the scheduler's queue of pending operators. If not, the operator has *completed*, and the scheduler is free to pick the next, least recently scheduled operator with pending tuples to run. Both `FIFO-Slice` and `RTC` should exhibit good instruction cache locality.

The `DF` scheduler starts by pushing a set of tuples from an input stream through an operator and then *directly* invokes the successor operator in the query plan whenever the operator `emits` a tuple. Thus, the `DF` call graph is a depth-first traversal of the query plan. This scheme does not require an operator scheduled next to allocate and deallocate memory for tuples (unlike in the previous cases, where tuples need to be copied) because synchronously executing operators can pass tuples by reference. But some queuing is unavoidable because the query plan is rarely a linear processing chain, and backtracking to an operator in the depth-first traversal requires data to be queued. Because a XStream query plan may contain cycles, the `DF` scheduler maintains the stack of the execution so far and enqueues those operators which would close the loop on to the scheduler queue for later execution. Once the depth-first traversal for a set of tuples has completed, the `DF` scheduler selects the least recently processed operator with enqueued data to run next.

3) Evaluation: We evaluate the performance of our schedulers on the DETECTAUDIO application described in Section II-A. We perform two sets of experiments: a scalability experiment that measures the performance of the schedulers under increasing offered load and a peak-performance experiment that measures end-to-end performance of different XStream applications. These experiments use the same machine and methodology described in Section IV.

Fig. 7(a) displays the results of scalability experiments, as the offered load increases beyond the maximum processing capacity of each scheduler. The figure shows a drop-off in the sustained throughput as the offered load increases. We determined that this degradation is due to the buildup in input queues when the system is overloaded. As queue sizes increase, a large amount of memory must be allocated and deallocated to keep up, which causes the heap maintained by the GNU C library's `malloc` implementation to grow and shrink, placing a heavy burden on Linux's virtual memory manager. Smaller amounts of allocation and deallocation could be handled entirely within a heap cache, or even the CPU cache.

To solve this problem, we augment our schedulers with source *rate control*, which synchronously admits a maximum of `tupleBurst` tuples from each source, waits until all the operators reach completion, and then repeats. This has the effect of limiting the total amount of memory used by the XStream engine at any time. Fig. 7(b) shows that admission control causes each scheduler's response to increased load to be flat rather than collapsing. (Alternatively, we could have limited the maximum size of the input queues and used back-pressure to throttle the rate of allocation/deallocation in the system.)

We assume that the admission-controlled results represent the true peak performance of the scheduler algorithms. As expected, the `FIFO` scheduler performs the worst, topping out around 4.5M samples/second. The `FIFO-Slice` scheduler with slice $= 28$ μs (labeled `FIFO-TS` in the figure) performs 11% better. The `RTC` performs 20% better than `FIFO`, while `DF` is faster by about 51%.

We now look at the relative peak performance of the schedulers. After the `FIFO` scheduler finishes running an operator on an input tuple, it is highly likely to switch to a different operator. The overhead of switching results in some "cache-busting"—both the instruction cache (operator code) and the data cache (input queues and operator state) on the CPU experience churn. Both `FIFO-Slice` and `RTC` mitigate this problem; in fact, `RTC` drains *all* the queues and automatically transitions to the one of the successors, so performs a little better than `FIFO-Slice`. The `DF` scheduler has poor cache locality, but is able to more than compensate for it by avoiding the expensive allocation, deallocation, and copying associated with emitting tuples into input queues. In fact, in a pure engine benchmark (EngineBench) consisting of a 1000 *pass-through* operators, the `DF` scheduler can be 5.8$\times$ faster than the other schedulers.

To confirm our intuition regarding the CPU cache behavior, we profiled the cache usage of these schedulers, and found that the execution of EngineBench using the `FIFO` scheduler resulted in $\approx 47\times$ the rate of misses in the L1 data cache compared to `RTC`. Also, it produces 20$\times$ the L1 instruction cache misses relative to `RTC`. `DF` has about 50% of the instruction cache miss rate as `FIFO` and a similar data cache miss rate because it switches rapidly between operators.

We would like to point out that, although the `DF` scheduler performs best in the majority of the benchmarks, there are some rare cases where it does not. Some applications involve "heavy-hitter" operators which generate a lot of output, which must be scheduled sparingly and whose output must be consumed quickly. In such scenarios, schemes like `FIFO` which schedule the operators in the query plan more uniformly than `DF` [1], have a better chance of avoiding memory allocation "spikes" which we have determined to be detrimental to performance. We show one such case in Section IV-B.

IV. EVALUATION

In the previous section, we described the implementation of XStream and presented some results to illustrate the performance of each individual component. In this section, we present data to specifically quantify the performance gains that stem from our system design. We demonstrate these performance gains within the XStream framework and also provide some comparative results for both the Borealis [16] engine and a commercial system, to show that the same ideas can produce a similar gain in other stream processing systems.

[1] `DF`'s decisions are dictated by the query plan connectivity.

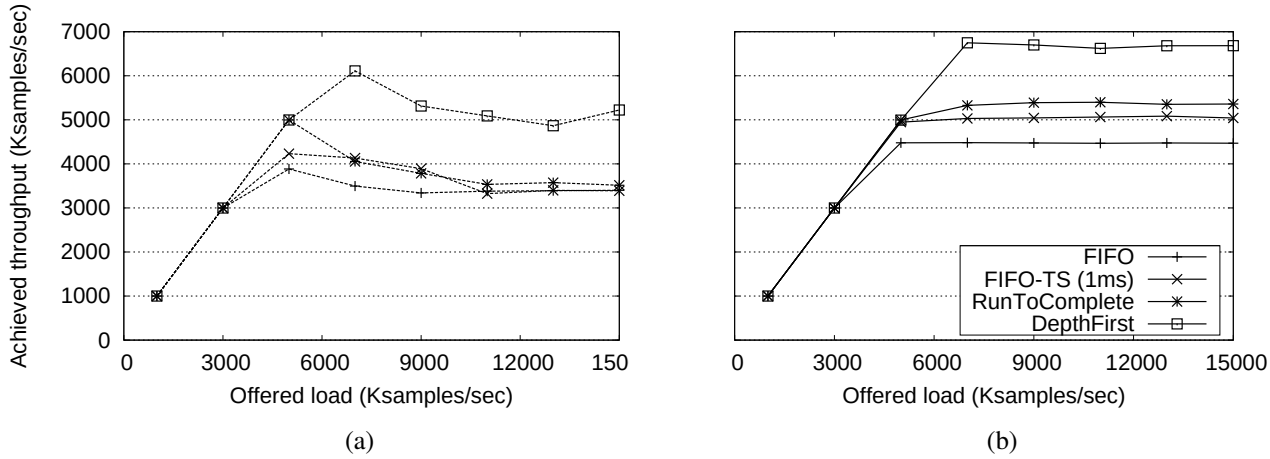

Fig. 7. Scheduler performance with increasing offered load, without rate control (a), and with rate control (b).

All performance tests were run on a dual CPU 2.8 GHz Pentium 4, with 1 MB L2 cache per processor and 1 GB RAM. We selected a simple method for measuring the performance of a running application: we ran the system on a pre-recorded data file at maximum offered load and measured the elapsed wall time before the complete file was processed. In order to get a consistent performance measurement, we set up our tests to pre-load the complete test datafile into the file system cache, before the timer started.

A. SPE Performance

We evaluate the performance of XStream relative to two existing streaming database engines: *Borealis*, a research project, and a mature commercial system whose name we anonymize as *Commercial*. The benchmarks are driven by a data source which reads from a nine second long, 44.1 KHz sound file of a speaker counting from one to five. The file is in a CSV (comma-separated values) format, to maximize compatibility; we separate the burden of loading, parsing, and marshaling the audio data from the file from the actual costs of processing it.

To focus exclusively on the benefits of SigSegs, we eliminate the effects of scheduling by using benchmark applications whose query plans are small and simple. Additionally, we include the results obtained from running a modified version of XStream, XStream-NoSigSeg, which operates on individual tuples and does not use XStream's advanced memory management.

The first benchmark, PASSCHAIN, evaluates the scheduling and data passing costs, and consists of a chain of ten operators, each simply passing the data along to the next operator.

The second benchmark, STATFILTER, tests the handling of windows of data, and consists of two sequentially connected filter operators. The first operator calculates the standard deviation of the last 4096 samples, and passes the window onward if it is greater than a threshold α. The second operator works similarly, passing only the windows whose average value $< \beta$.

The final benchmark, SILENCEFILTER, evaluates the performance of joins. The sound source feeds into a *silence detector* and a *silence filter*. The silence detector accepts the stream of sound samples from the data source and outputs the ranges of timestamps of samples considered to contain speaker voice. A window of data is considered to be non-silent if its standard deviation is $> \gamma$. The silence filter joins the original sound stream with the ranges of timestamps produced by the silence detector, to produce a stream of audio containing concatenated "non-silent" audio.

We note that the SILENCEFILTER application is a simplified version of the workflow of DETECTAUDIO (see Section II-A), only with a single audio channel, and without the use of FFT. We use this version because FFT is hard to express in the query languages of some of the systems we evaluate.

The results in Table I show an average of three trials, with negligible variance. We disabled any debugging output during the execution of the applications, and prewarmed the filesystem cache by running the benchmark six times, but recording only the last three results. We also recorded the amount of time each system spent loading the file, and subtract that from the total time-to-completion of each benchmark.

The results achieved by XStream-NoSigSeg, in comparison to XStream, demonstrate the drastic benefits of using the SigSeg ADT (note that the units for the XStream column are in *millions*). They also validate our engine's performance with respect to other systems: XStream-NoSigSeg performs

Units: samples/s	Commercial	Borealis	XStream	XStream NoSigSeg
File loading	4.2 sec	8.1 sec	0.2 sec	2.24 sec
PASSCHAIN	177.7K	5.47K	71.4**M**	92.8K
STATFILTER	72.7K	7.05K	97.6**M**	57.9K
SILENCEFILTER	4.58K	142.8	64.5**M**	75.1K

TABLE I
THROUGHPUT OF SPES ON MINI-BENCHMARKS (FIRST ROW OF NUMBERS IS FILE LOAD TIME).

on par with the commercial SPE. XStream loads the file faster because the commercial SPE's input file parser is more elaborate. We suspect that *Commercial* does better than XStream on PASSCHAIN because of its internal query optimizations, such as operator merging. On the other hand, the numbers in Table I clearly indicate the advantages of the `sync` operator over generic joins, even in a system which uses tuple-by-tuple stream processing.

Our results show that XStream outperforms the commercial system by a factor of more than 400 in simple message passing, a factor of more than 1340 in windowing and statistics operations, and a factor of 14000 (**four factors of magnitude**) in time-based joins. XStream outperforms Borealis by more than four factors of magnitude in simple benchmarks, and more than **five** factors of magnitude in temporal joins. Why does our system outperform the existing systems so drastically?

XStream uses the isochrony of signal data to store a single timestamp per SigSeg. The competing systems store a timestamp per every tuple. In addition to the timestamp, some of the systems attach extraneous metadata to every tuple. For example, Borealis attaches a 53 byte header to every sample it passes around, greatly increasing the chance of cache misses.

XStream takes advantage of memory management optimizations: passing signal data by reference (via SigSegs), and using a single copy of any piece of signal data throughout the system. The other systems suffer from the overhead of extraneous memory management and copying.

XStream implements *sync*, an efficient form of time-based join which operates on time ranges as opposed to samples. Other systems join on a sample-by-sample basis, which is considerably more costly in terms of the number of timestamp comparisons.

B. Performance Benefits of SigSegs

In this section, we present data to specifically quantify the performance gains that stem from using SigSegs. We have found that SigSegs affect performance in four ways: (1) they eliminate per-sample processing which incurs a high scheduling overhead, (2) they represent data in column-major order, thus compacting it for a lower cache footprint, (3) they reduce the memory footprint by eliminating redundant time-stamp information, and (4) they enable `sync`, an optimized `join` construct.

To separately quantify each of these factors, we implemented special versions of our PIPELINE and DETECTAUDIO applications in which we enable each improvement in turn, with the results shown in Fig. 8. These versions are named as follows:

- SAMPLES: passes each sample between operators as a separate `<time,value>` tuple.
- WINDOWSTRIDE: passes whole windows of `<time, value>` tuples between operators. We implemented this only for PIPELINE.
- WINDOWS: same as above, but re-orders to column-order before heavy operators (*e.g.*, `fft`).
- SEGS: the SigSeg-based version.

To show the differential impact of `sync` and `join` operators, we created additional versions of DETECTAUDIO: one version that cuts off the query plan directly before `sync`, and another that implements the complete DETECTAUDIO application, substituting `join` for `sync` in the WINDOW and SAMPLES cases. We now describe each of these cases in detail, starting from SAMPLES and adding optimizations.

1) Window-passing optimization: Processing high-rate data one sample at a time places incurs a large queuing and scheduling overhead. In our first optimization, an operator that would emit N individual tuples now packs those tuples into a SigSeg and thus passes them by reference in a single `emit` call. Note that although we use a SigSeg to enable sharing and pass-by-reference, the SigSeg is defined on `<time,value>` tuples so the data is interleaved with timestamps. This optimization is analogous to the *synopsis sharing* optimization of Stanford STREAM [2].

Comparing the results in Fig. 8 (PIPELINE), we see that there is an increase in performance from SAMPLES to WINDOWSTRIDE, with some variation among schedulers. Since the `Depth-First` scheduler bypasses inter-operator queues wherever it can, it far outperforms the others on SAMPLES where queue and scheduler costs dominate. The window-passing optimization is a clear win: `Depth-First` improves by 1.4×, while the other schedulers improve by a factor of 3.

2) Column-order optimization: When we implement signal processing operations such as `haarwavelet` over windows of `<time,value>` tuples, we have the option to process the data by "striding" through the row-major data, or to first copy the data into a column-major vector. In our tests, WINDOWSTRIDE executes `haarwavelet` directly on row-major data while WINDOWS first copies the data into column order, and then copies it back into tuple form. By comparing WINDOWSTRIDE to WINDOWS, we can see that re-ordering the data yields a 2× improvement for all schedulers except `DF`, which yields 1.7×. This effect is caused by a larger memory footprint and by cache exhaustion, as discussed in Section III-C.

3) Cost of interleaved time-stamps: While operators such as `haarwavelet` are more heavily influenced by cache performance, the performance of the system in general is affected by the memory footprint. Maintaining sampled data as `<time,value>` is very inefficient. We can see the effect of maintaining interleaved time-stamps by comparing the performance of SEGS and WINDOWS. In this implementation, timestamps are 64 bit integers—doubling the memory footprint of PIPELINE (since samples are `double`) and tripling that of DETECTAUDIO (since samples are `float`). Averaging over all schedulers, eliminating time-stamps yields a 1.6× improvement in PIPELINE, and a 1.8× improvement in DETECTAUDIO.

Note that different scheduling disciplines also affect performance. In particular, disciplines that attempt to drain the queue of an operator run the risk of increasing the memory footprint if the operator emits more data than it consumes. This explains the performance penalty incurred by `RTC`; in

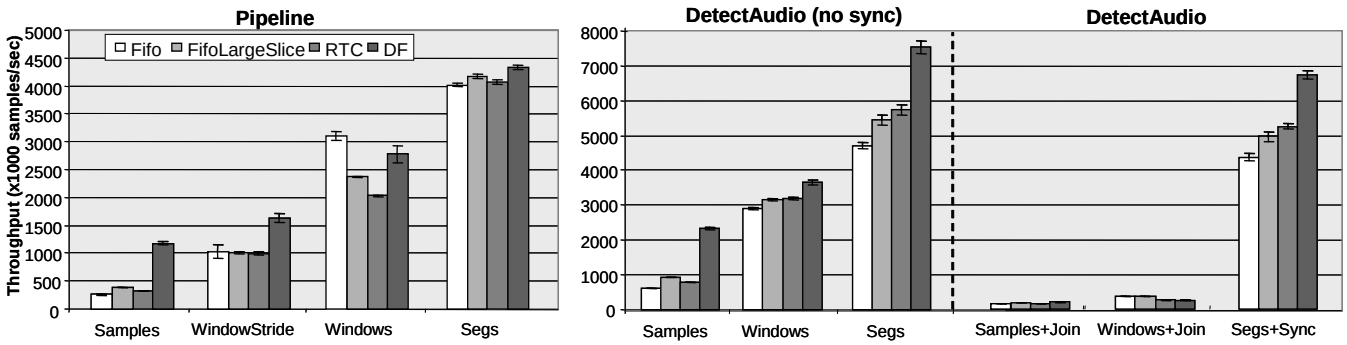

Fig. 8. Relative performance of different approaches to data modeling and schedulers. Each bar shows the average rate over 10 trials, with error bars representing 95% confidence intervals.

this case, the `haarwavelet` operator produces outputs for each of the sliding windows, increasing the memory footprint and overwhelming the L2 cache. Although this effect is present in both SEGS and WINDOWS, it is more pronounced in WINDOWS because of its increased footprint.

4) Comparison of `sync` *and* `join`*:* To assess the performance of `sync`, we implemented two versions of the DETECTAUDIO application, one using our `sync` operator, and the other performing a symmetric streaming hash join on timestamp. In order to clearly show the added cost of these operators, we also show the baseline performance of DETECTAUDIO with the query plan terminated before `sync`. Comparing the two graphs, we see that the addition of `sync` reduces throughput to 0.9× of the original, vs. a reduction to 0.2× using `join`.

This problem is most likely due to the large increase in memory footprint from buffering data in the join. Because `sync` buffers data through the SigSeg mechanism, these buffers are stored efficiently and may be shared, resulting in a smaller increase in memory footprint—whereas the timestamp overhead in the hash `join` increases the memory usage by a factor of 5. The choice of scheduler also has a significant impact on memory costs: schedulers that enter the data into the hash tables more rapidly (*e.g.*, `RTC` and `Depth-First`) end up expanding the tables more than `FIFO`, which tends to service the input queues more evenly.

V. RELATED WORK

There has been considerable previous work on conventional stream processing [1], [2], [3], [4], [5], [6]. XStream differs from these systems in two main ways. First, it provides a single language, WaveScript, to express both traditional stream processing and signal processing functions. This offers advantages over both SQL-based systems where UDFs are typically written in a different language, and tools like MATLAB, Simulink and LabVIEW [17], [18], [19] which are good for writing standalone signal processing applications but do not include satisfactory streaming and relational support. Second, it supports windows as first-class entities, as opposed to conventional systems where windows are tied to particular operators in the query plan. Flexible support for windows enables queries like time alignment to be naturally expressed and efficiently executed.

Sequence databases like SEQ [20] support time-series data as first class objects. However, these systems are targeted at simple queries that analyze trends in data over time, and are less suited to expressing complex signal processing functions. The same is true of Tribeca [21], and Gigascope [22], which are both streaming database systems for networking applications that share our objective of handling high data rates.

There is a well-established literature on high-performance compiler optimization in stream processing and dataflow languages [23], [24]. Most of this work is predicated on a synchronous data flow model where operators produce and consume deterministic amounts of data at each time step, enabling static operator scheduling. In contrast, XStream has an asynchronous dataflow model similar to streaming databases. This limits static optimizations somewhat, but is essential to handle real-world signal processing operators that produce data at varying rates (*e.g.*, the DETECTAUDIO prefilter). Moreover, XStream exploits isochrony to reap some of the performance benefits of synchronous dataflow, helping bridge the gap between the two models.

Stanford STREAM employs a technique called synopsis sharing [2] to eliminate redundant materialization of windows shared between adjacent operators. Our reference counted SigSegs are similar in spirit but much more flexible than synopsis sharing, which does not support random access, efficient merging or range extraction operations on time windows. IO-Lite [25] is a buffering scheme to avoid redundant copying across process boundaries in high performance systems like web servers. It is instructive that our memory manager, developed in the entirely different context of signal processing, has a similar design.

The XStream scheduler builds on lessons from previous work on operator scheduling in streaming systems [26], [27] and shares high level ideas with these systems, notably tuple batching and `iterate` merging. However, those efforts are concerned with providing latency and memory consumption guarantees when tuples arrive unpredictably, while our design is more focused on minimizing overhead and scaling performance to very high data rates.

Our paper shares high-level ideas with a previous short position paper [7]. That paper made the case for integrating relational and signal processing functions in a data stream processing system for high-rate applications. It sketched the high-level details of a language and mentioned the run-time component, but left the design and implementation of the system and language for future work. In addition to describing these details, this paper describes how XStream applications from three different domains are written in WaveScript and evaluates their performance.

VI. Conclusion

This paper described the architecture and implementation of XStream, a system that combines event stream and signal processing. XStream aims to improve both programmer productivity, by making it easy to develop user-defined processing functions, and achieve high performance, processing several million samples per second on a standard PC. XStream incorporates a new basic data type to represent isochronous signal segments and uses the WaveScript language to express queries and write custom operators. The XStream runtime uses an efficient SigSeg memory manager and a depth-first operator scheduler to achieve high performance.

We described three real-world applications and measured their performance, obtaining both end-to-end results and a detailed experimental analysis of the various components of the XStream engine. These results showed the benefits of our system architecture and data model. They are also encouraging in comparison to traditional SPEs: our benchmark measurements show that, XStream is between 400 and 14000 times faster than a commercial SPE. On similar conventional PC hardware, XStream applications are able to implement non-trivial real applications running at speeds between 4.8 and 7 million samples per second. Thus, our architecture is well-suited for high-rate processing for an important emerging class of signal-oriented streaming applications.

VII. Acknowledgements

This work was supported by the National Science Foundation under Awards Number CNS-0520032 and CNS-0720079.

References

[1] 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.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," in *Book Chapter*, 2004.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.

[4] (2007) Streambase corporate homepage. [Online]. Available: http://www.streambase.com/

[5] (2007) Coral8 corporate homepage. [Online]. Available: http://www.coral8.com/

[6] (2007) Aleri corporate homepage. [Online]. Available: http://www.aleri.com/

[7] L. Girod, K. Jamieson, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "The case for WaveScope: A signal-oriented data stream management system (position paper)," in *Proceedings of Third Biennial Conference on Innovative Data Systems Research (CIDR07)*, 2007.

[8] I. Stoianov, D. Dellow, C. Maksimovic, and N. Graham, "Field validation of the application of hydraulic transients for leak detection in transmission pipelines," in *Proceedings of the International Conference on Advances in Water Supply Management. CCWI-Computing and Control for the Water Industry, London, UK*, September 2003.

[9] I. Stoianov, L. Nachman, S. Madden, and T. Tokmouline, "PIPENET: A wireless sensor network for pipeline monitoring," in *IPSN '07: Proceedings of the sixth international conference on Information processing in sensor networks*. New York, NY, USA: ACM Press, 2007.

[10] A. Lakhina, M. Crovella, and C. Diot, ""diagnosing network-wide traffic anomalies"," in *SIGCOMM*, Portland, OR, August 2004.

[11] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Volker, and S. Savage, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," in *SIGCOMM*, Pisa, Italy, August 2006.

[12] H. Wang *et al.*, "Acoustic sensor networks for woodpecker localization," in *SPIE Conference on Advanced Signal Processing Algorithms, Architectures and Implementations*, Aug. 2005.

[13] A. 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: ACM Press, 2007.

[14] V. Trifa, "A framework for bird songs detection, recognition and localization using acoustic sensor networks," Master's thesis, École Polytechnique Fédérale de Lausanne, 2006.

[15] A. Ledeczi *et al.*, "Countersniper system for urban warfare," *ACM Transactions on Sensor Networks*, vol. 1, no. 2, pp. 153–177, Nov. 2005.

[16] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *CIDR*, 2005.

[17] (2007) Matlab product homepage. [Online]. Available: http://www.mathworks.com/products/matlab/

[18] (2007) Simulink product homepage. [Online]. Available: http://www.mathworks.com/products/simulink/

[19] (2007) Labview product homepage. [Online]. Available: http://www.ni.com/labview/

[20] P. Seshadri, M. Livny, and R. Ramakrishnan, "The design and implementation of a sequence database system," in *VLDB*, 1996.

[21] M. Sullivan and A. Heybey, "Tribeca: A system for managing large databases of network traffic," in *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[22] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *SIGMOD*, 2003.

[23] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *ICCC*, April 2002.

[24] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[25] V. S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: a unified I/O buffering and caching system," *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 37–66, 2000.

[26] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *VLDB*, 2003.

[27] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," *The VLDB Journal*, vol. 13, no. 4, pp. 333–353, 2004.