Persistent Temporal Streams

David Hilley and Umakishore Ramachandran

College of Computing / *Georgia Institute of Technology*
davidhi@cc.gatech.edu / rama@cc.gatech.edu

Abstract. Distributed continuous live stream analysis applications are increasingly common. Video-based surveillance, emergency response, disaster recovery, and critical infrastructure protection are all examples of such applications. They are characterized by a variety of high- and low-bandwidth streams as well as a need for analyzing both live and archived streams. We present a system called *Persistent Temporal Streams (PTS)* that supports a higher-level, domain-targeted programming abstraction for such applications. PTS provides a simple but expressive stream abstraction encompassing transport, manipulation and storage of streaming data. In this paper, we present a system architecture for implementing PTS. We provide an experimental evaluation which shows the system-level primitives can be implemented in a lightweight and high-performance manner, and an application-based evaluation designed to show that a representative high-bandwidth stream analysis application can be implemented relatively simply and with good performance.

1 Introduction

Continuous live data streams are ubiquitous and their analysis is a central component of many applications. Network monitoring, surveillance, robotics, inventory tracking, traffic or weather analysis, disaster response and many other application domains fall under this umbrella. All of these applications have one common trait: live streaming data is analyzed continuously, and the results are used in some sort of feedback loop to direct further analysis and perform external side-effects such as triggering alerts, producing continuous data summaries for human monitoring or manipulating the environment. We call this class of applications *live stream analysis applications*, because streams are analyzed and consumed "live," as the data is produced. Many such applications also require access to historical data – data that was streamed in the past and is now archived.

While such applications are becoming ubiquitous, programming support is relatively immature. Our broad goal is the development of a unified distributed programming abstraction for accessing live and historical stream data, suitable in scenarios requiring significant signal processing on heavyweight streams such as audio and video. Existing solutions for constructing such applications tend to fit into two broad categories: 1) "stream database" or "stream processing engine"-style systems or 2) general-purpose distributed programming systems. The former category has centrally managed and controlled execution, while the latter does not impose a particular computational model on applications, only modeling data interactions. The latter support loosely coupled systems of independent communicating components with no centralized control. To the best of

our knowledge, no prior system has provided a unified abstraction for both transport and storage of live streams as a simple distributed programming primitive. In this paper we propose Persistent Temporal Streams (PTS), a novel distributed system that provides simple and efficient programming idioms for dealing with distributed stream data. At the heart of PTS is the *temporal stream* abstraction, providing a uniform interface for both time-based retrieval of current streaming data and data persistence.

PTS fits between very high-level and heavyweight solutions like full databases with query languages and lower-level non-stream oriented distributed communications facilities typically used for distributed applications (MPI, RMI, etc.) plus separate storage facilities. Our approach represents a middle-ground in a vast design space. At one extreme are high-level heavy-weight solutions that incorporate full databases with query processing capabilities. At the other extreme are lower-level non-stream oriented distributed communication facilities for constructing distributed applications that require a separate treatment of persistent data. This middle-ground for continuous streaming data is roughly analogous to solutions such as Distributed Data Structures [1], BerkeleyDB [2] and Boxwood [3] for non-streaming data. The *temporal stream* provides first-class recognition of time, which is a critical distinguishing aspect of continuous live streams over other types of data; the tailoring of the abstraction to the problem domain makes live stream analysis applications more straightforward to build, as does eliminating a programmer-visible artificial distinction between past streamed data and current "live" data.

This paper's contributions are the following:

- A persistent *temporal stream* abstraction targeted for live stream analysis applications (Section 2)
- An architecture for stream persistence and an analysis of the potential design space of persistent temporal streams (Sections 2.1 & 2.2)
- A system design and implementation of PTS, a distributed runtime realizing the persistent temporal stream abstraction (Section 3)
- A system-level experimental evaluation of PTS and an application-based evaluation using a video-based surveillance system [4] (Section 4)

We conclude with related work (Section 5) and a summation (Section 6).

2 Persistent Temporal Streams

Temporal streams are a key feature of live stream analysis applications – as a concrete example, think of a video feed: the stream is unbounded and produced at a finite rate, and the video frames are temporally ordered. Each frame represents some sampled interval of time based on the frame rate. Event streams and other aperiodic streams may not have fixed output rates, but trigger based on certain environmental conditions, like a temperature sensor sending an alert when a threshold is reached. In both cases, data items are associated with specific time information. All "live" streams have a natural relationship with time (wall-clock time). Broadly, our model of temporal streams is a time-indexed sequence of

discrete data items; each item has a timestamp and spans a time interval ending with the timestamp of the next item.

In PTS, a temporal stream is represented by a *channel*, which is a distributed data structure encompassing an interface for both transport and manipulation of streaming data; each channel holds a time-indexed sequence of discrete data items (such as video frames) and analysis code retrieves data items by specifying time intervals of interest. Applications interact with channels by means of "get" and "put" operations. The basic stream operations are 1) put(i, t) – put data item i on the stream with timestamp t (typically the current time); and 2) get(l, h) – get all items falling within the interval $[l, h)$. A variety of expressive *time variables* [5] (such as *now*) are also provided to formulate intervals such as "the most recent 10 seconds of video data", and a wide range producer/consumer patterns can be expressed using these time variables. The system maintains a window of current stream data, automatically garbage collecting older items – for example, an application could specify that channel c_1 should keep 30 seconds worth of data, and items older than that may be reclaimed.

The benefit of this model is that it provides a higher-level stream abstraction, which fits at the intersection of an application's manipulation of data and stream transport. Since the stream abstraction has a familiar get/put-based interface as a data structure, it is simple to use. Finally, by providing first-class recognition of time, it provides a more natural way to write analysis code that deals with continuous streams – similar to tuple models in streaming database work, and higher level than general-purpose distributed programming mechanisms appropriate for high-volume data transfer. Rather than managing and buffering an ephemeral, linear flow of data, the application can access stream data in terms of higher-level time information. Since many live stream analysis applications also need to store and retrieve historical data for trend analysis, a persistence mechanism that fits within the temporal stream model is a useful feature. For instance, a surveillance system might store historical video streams for some predetermined archival period in a degraded form (e.g. lower resolution).

Integrating persistence into the temporal stream abstraction avoids an artificial distinction between data that is currently available in streams (a window of recent data) and data that was streamed in the past but is now archived. This change elevates the temporal stream abstraction from a communication abstraction to a general-purpose *data* abstraction, uniformly modeling stream data interactions. Although the same abstraction is used for live and stored data, information about the source of data should still be made available to the programmer since the difference in access time and data quality or representation can be significant. From a programming perspective, eliminating unnecessary non-uniformity is desirable as it can make applications simpler to construct and less brittle in the face of change.

The issues surrounding the incorporation of persistence into the stream programming model are the core of this paper. In the following subsections, we present an architecture for accomplishing this goal. The architecture explores and answers several important questions related to incorporating persistence in

a seamless manner into a distributed programming model for a wide range of streaming applications:

- How is persistence integrated into the programming model API?
- How are data items mapped to persistent forms?
- What factors affect the choice of storage backends for persistent stream data?
- How do we account for information lifecycle management (ILM) issues (e.g. redundancy, free space management, hierarchical secondary storage)?

2.1 An integrated architecture for live and archived streams

Our high-level persistence interface is directly enabled by extending the time-oriented channel interface – a channel can now be marked by an application as persistent (at creation time or later). Persistent channels empower the application programmer in the following ways: 1) items are automatically committed to persistent storage with related time-stamp information, and 2) time intervals for retrieval of items may now reference both live and persistent items. Figure 1 depicts a get operation with an interval spanning live and stored data. Other high level interface decisions are described below.

Get interface: The application may optionally constrain a retrieval operation to adjust for the difference in latency of access and potential data format differences of stored versus live items. The options are as follows: 1) `ANY` – any items, live or stored; 2) `LIVE` – only live items, 3) `STORED` – only stored items; 4) `ANYSPLIT` – return live items and load stored items from disk in the background, caching them in a temporary in-memory cache for a subsequent get.

Per-stream data representation: An application can also control how items are mapped to a persistent form. Some may wish to degrade the quality of items, reduce the number of items or otherwise change their format. An application can provide a *pickling handler*, which is responsible for mapping items to their persistent representation (defaulting to the identity function). For example, a video channel's handler may JPEG compress video frames or reduce the image resolution. In addition to one-to-one item mappings, the pickling handler can take N items and produce a single item to store: for example, an event channel's handler may transform thirty small events into some sort of digest. When N items are mapped to one item, the original timestamp information is retained, so the same get request will operate similarly on live and stored data. That is to say, if two items are mapped to a single stored item, it will span the combined time interval of the original items. As a direct extension of this functionality, an application may provide multiple handlers with varying levels of disk usage versus processing time and the runtime can automatically switch based on system-level cues.

Per-item persistence control: In addition to per-stream control via pickling handlers, per-item control is possible: a data producer may mark an item placed into a channel with the `NOPERSIST` flag. This will cause the persistence mechanism to ignore it, so the item will disappear for good when it is garbage collected from the live stream.

All-in-all, the programmer visible interface to a channel is essentially unchanged – put(i, t) and get(l, h) still operate as before, but the potential span

of items available in a channel now includes historical data rather than just a window of current live data. Put takes an optional `NOPERSIST` flag and get takes an optional `ANY`, `LIVE`, `STORED` or `ANYSPLIT` modifier (`ANY` is the default).

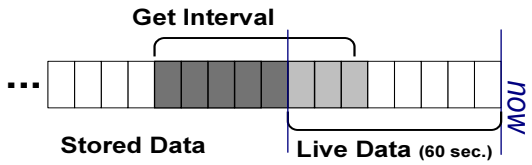


Fig. 1. Get operation spanning stored and live data

2.2 Storage requirements & design choices

At the high level, the stream persistence interface is natural and intuitive; all an application needs to know is that data items are mapped to persistent forms using a known transformation and stored along with timestamp information. Underneath this abstraction, however, the data must be stored to "stable" storage somehow, and the potential design space is large. The streams could be stored to a local filesystem, a distributed filesystem, a DBMS, a distributed virtual block device, an object store, or some other storage abstraction (Boxwood [3]'s persistent B-link tree abstraction is potentially quite well-suited), and there are many orthogonal design choices associated with each. In this subsection, we discuss several PTS design properties.

Redundancy/Availability: Some properties of the underlying storage mechanism manifest themselves as higher-level concerns. For example, an application may desire some form of redundancy so a stored stream does not become inaccessible due to disk or host failure. This could be accomplished in a variety of ways such as using a redundant, distributed storage mechanism as a backend, using primary copy replication, or making use of shared disks (e.g. via a SAN).

Free space management: Another storage-level property exposed at a higher-level is the management of free space. For high-bandwidth data streams, like video, an application will often want to use local storage as a ring-buffer so the oldest stored data will be overwritten when storage is full. Support for some policies may already be provided by a storage backend, however. For example, the GPFS [6] distributed filesystem provides internal support for rich information lifecycle policies based on filesystem metadata – a policy could specify that old data can be reclaimed or moved to lower performance storage.

External applications: One may also want a persistent stream stored in a particular backend for reasons external to the application: for example, a user may want sensor readings inserted into a table in a relational database for offline analysis by another application or a third party.

Suitability for workload: The access patterns created by storing streaming data are atypical workloads for some potential backends. Stored items are never updated and are read rarely (relative to the number stored). From a storage perspective, the data is essentially append-only, which affords simple and efficient consistency management strategies. Ideally, the backend should not block concurrent reads of older data while appending newer data. The system must also

support ranged queries since data is accessed by specifying intervals. When multiple streams are involved, the typical access patterns of storing many append-only streams simultaneously do not interact well with most general-purpose filesystem layouts [7]. Hyperion [7] addresses the problem of writing and querying multiple streams of captured high-data rate network traffic with a custom filesystem called StreamFS. The authors also present a "log file rotation" strategy for improving stream data layout on typical Unix filesystems.

To deal with diversity in requirements, we provide pluggable storage backends. Given the design tradeoffs discussed above, our initial prototype supports three backends: 1) a local filesystem backend (called `fs1`), 2) a distributed filesystem backend using GPFS (called `gpfs1`), and 3) a MySQL backend. Since we want to be able to handle multiple high bandwidth streams, we think Hyperion's StreamFS [7] (or a slightly modified version) is best-suited to our target domain when using local disks. StreamFS is not publicly available, so we implemented our own filesystem-based backend called `fs1` as a first-order approximation using the "log file rotation" approach presented in the Hyperion paper. We would also like to provide a distributed storage solution with advanced ILM functionality, so we leverage the distributed filesystem GPFS for this purpose. A MySQL backend is provided for scenarios where streams need to be stored in a relational database (e.g. for analysis by other applications). In general, we do not believe MySQL is a good general backend choice because it imposes a relatively large overhead on the storage process and was not designed for this particular workload.

3 System Design & Implementation

In this section we describe the concrete system architecture of PTS and salient implementation details. First, we provide general high-level system details (Section 3.1), followed by channel implementation details independent of whether a channel is persistent or not (Section 3.2). Section 3.3 summarizes the implementation of the stream persistence architecture. Figure 2 shows the structure of the PTS system software stack.

3.1 System Structure

The system is structured as a distributed runtime and the core of the system is a set of cooperating *peers* using the PTS library – peers are data consumers or producers and host resources. In typical usage, a peer can be thought of as a multi-threaded process with a distinguished identity in PTS. We also have a distributed, replicated directory storing system metadata (for instance, naming information or mappings between opaque PTS endpoints and network endpoints) which is accessed by peers via an RPC-like protocol. Understanding the persistent stream architecture does not require knowledge of the metadata directory design; for the purposes of this discussion, one can simply imagine naming/location metadata is available in some centralized directory.

Channels are PTS's distributed and time-indexed representation of temporal streams and the fundamental mechanism for data transport, manipulation and storage. Almost all of the implementation complexity of peers revolves around

hosting or accessing channels. Peers place timestamped items into channels ("put" operations) and retrieve items based on time intervals ("get" operations). Channels are hosted at a single peer, but they may be read-only replicated (primary copy replication) for capacity or availability; a channel may also migrate dynamically to another peer if necessary. Architecturally, every peer is a first class entity that may host channels or interact with existing channels.

The system assumes data producers will have synchronized clocks, which is not an unreasonable burden. NTP [8] is widely deployed and can keep hosts over the Internet synchronized with high precision. For extremely limited devices, more lightweight techniques or producer proxies can be used.

The implementation described here is written in ANSI/ISO C89 with pthreads.

3.2 Channels without data persistence

A channel stores current live stream items ordered by timestamp; items older than a given "currency" bound (e.g. 30s) are automatically reclaimed. Conceptually, a channel may be viewed as an ordered list of data items and associated metadata (e.g. timestamps) located at the peer hosting a channel. Each peer has a single *gatekeeper* TCP/IP endpoint where other peers can either interact with channels hosted locally or negotiate a separate dedicated connection to a particular channel for bulk data transfer. The transport protocol of dedicated connections can be negotiated on a per-connection basis (e.g. shared memory for colocated processes, RDMA or SCTP within a cluster, etc.). A pool of worker threads is used to handle remote get/put requests on dedicated connections.

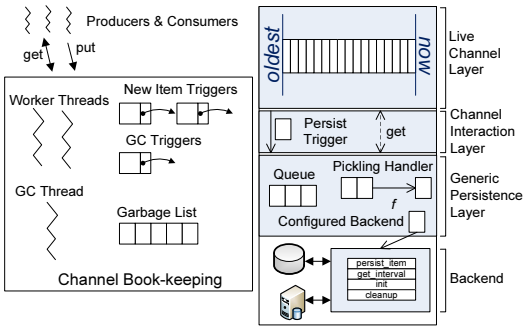


Fig. 2. PTS Architecture

When performing get or put operations, a channel is identified by a *channel descriptor*, which is an opaque reference to a particular channel data connection. Each peer has a table mapping channel descriptors to concrete connection endpoint information, which acts like a cache: normally, channel operations use the cached information and no metadata lookup or binding is necessary. When a channel moves or a new connection to a channel is needed, the runtime contacts the system metadata directory to find out which peer is hosting a channel and then contacts the peer's gatekeeper endpoint to negotiate a data connection.

A channel also has an integral *trigger* mechanism with two different types of triggers: 1) garbage collection triggers and 2) new item triggers. Both types of triggers are functions which apply to a single item at a time. Garbage collection triggers are invoked when an item is about to be removed from the channel's "live" data and either freed or placed on a garbage list; new item triggers are invoked when a new item is added to a channel. While this is a very simple concept, it is also remarkably flexible. Triggers are used to implement a variety

of functionality – new item triggers are the basis for replication of channels, multicasting channel data, an optional push-style programming interface, and a virtual synchronization mechanism [5]. For example, to set up a copy of channel A replicated at host B, the system creates a new locally hosted channel at host B, and sets up a new item trigger on channel A to send each item to the replica. Any host can now use the copy by updating its channel descriptor table to point to the replicated channel.

To execute triggers, each channel maintains a list of functions to call for each trigger type and invokes them sequentially and in the execution context of the thread that added an item or caused an old item to be prepared for garbage collection. Consequently, trigger functions are expected to have short, bounded execution times. When a trigger is added, an initialization function is run which can set up an event queue and a dedicated listening thread or bind to a shared thread/thread pool for asynchronously servicing longer triggers (analogous to "bottom half" processing for interrupt handlers). Triggers can be loaded by name statically or dynamically (via `dlopen`).

The C-based runtime uses reference counting for internal storage management of channel data. Without persistence, channel garbage collection is easy: since a *put* call places a single timestamped item into a channel, we just check to see if we can reclaim the oldest item in the channel after a put call. If the span between the newest and oldest items is greater than the channel's specified currency bound, the item is removed from the live channel and the system invokes the GC trigger functions. The last trigger will either place the item on a garbage list if its refcount is non-zero or immediately free it otherwise. If the item isn't immediately freed, we walk through a small fixed number of items on the garbage list and free those with refcounts of zero. There is no need for a background GC thread because new garbage is only generated when old items are displaced by newly arriving data, so the system can maintain stasis by doing a small amount of GC work cooperatively during each put call.

3.3 Persistence

The persistence mechanism implements the high-level channel persistence semantics and is separated into three general layers: 1) the channel interaction layer, 2) the generic persistence layer and 3) specific persistence backends. The persistence backends are loaded dynamically and handle interfacing with a particular storage mechanism (e.g. a filesystem). Both the generic persistence layer and concrete persistence backends provide a simple API with four basic calls: `persist_item` and `get_interval` as well as `init` and `cleanup`. `persist_item` and `get_interval` directly correspond to the live channel get/put operations.

Channel interaction layer: The channel interaction layer is the small set of hooks in the channel implementation (described in Section 3.2) which interfaces with the generic persistence layer. For channel get operations, this consists of the logic to interpret get types (`ANY`, `LIVE`, etc.) and to call down to the persistence layer if stored items will be needed. If a get operation is performed on interval $[l, h]$ and some live item has a timestamp $\leq l$, then no call to the persistence layer is needed. After a `get_interval` call to the persistence layer is made, this

layer also handles placing temporary items retrieved from the storage backend on the garbage list.

Triggers are used to send items to the lower levels of the storage stack by calling `persist_item` in the generic persistence layer. The channel interaction layer also contains routines to initialize the persistence interface. When a channel is initially marked as persistent, a background garbage collection thread is spawned since get operations spanning persisted items may create significant additional garbage and our previous strategy may not be able to keep up (particularly if put calls are rare).

Generic Persistence Layer: This layer sits between the channel implementation and a particular concrete storage backend. It maintains a small queue of items to be persisted in batches, and is responsible for calling pickling handlers to map items into their persistent representation. The `persist_item` call just increments an item's refcount and enqueues it on a processing work queue handled by a worker thread. This structure has several key properties: 1) it prevents the `persist_item` call from blocking a long time (since it is called from a trigger), 2) queueing is necessary for pickling handlers that transform N items to 1 item, 3) if items are eagerly pushed to storage on a channel with multiple producers, some queueing is necessary to ensure items are written out in temporal order, and 4) it allows the generic persistence layer to serialize writes to the backend.

Several of these properties simplify the assumptions a storage backend must deal with. For example, serializing writes to the backend by the persistence layer simplifies backend implementation – it may assume there are no concurrent writers, although a single writer may overlap with item reads. Another feature of this layer is that it guarantees that items will be presented in temporal order to the storage backend, which again can simplify the backend's implementation.

To process a `get_inverval` request, the generic persistence layer must search its work queue for items that are waiting to be persisted as well as call down into the concrete storage backend layer to retrieve items that have reached "stable" storage. Finally, the generic persistence layer is also responsible for dynamically loading storage backends and pickling functions (via `dlopen`) when a channel is first marked as persistent. The generic persistence layer can also monitor and react to different kinds of resource contention: by measuring the latency of backend `persist_item` calls, it can determine if storage contention is too high. Similarly, by timing pickling handler execution, it can estimate CPU load. The generic persistence layer can adjust to these conditions by switching between pickling handlers or disabling pickling. The persistence layer primarily affects CPU and storage contention; network and memory usage can be monitored and controlled by the live channel implementation (Section 3.2).

Storage backends As mentioned earlier, the storage backends are responsible for implementing `persist_item` and `get_interval` calls.

MySQL backend: The MySQL backend is not designed for streams with high data rates, but it is certainly appropriate for low bandwidth sensors. `persist_item` simply puts a tuple with (timestamp, data) into a specified table and `get_interval` performs a `SELECT` of items with timestamps in the interval $[\text{low}, \text{high})$. Given

the `SELECT` query, the timestamp column should also have an index suitable for range queries (e.g. B-trees). Currently the backend simply stores item data as BLOBs, which is not very flexible. We are looking into providing a richer interface by allowing user-defined functions to map binary item data into some number of separate data items matching the desired schema.

Filesystem backend: The `fs1` filesystem backend is implemented as a lightweight overlay on top of a filesystem (SGI's XFS [9] generally has the best overall performance for these workloads [7]), but could also be implemented directly on a block device. The data layout is quite simple and uses the properties provided by the generic storage layer to avoid unnecessary complexity and synchronization. We use the log file rotation approach from the Hyperion paper [7].

A backend needs to store timestamped data items in order and retrieve them by bounded time intervals. To accomplish this, `fs1` uses a two-level indexing scheme. A given channel has a single top-level index file and many individual data files, each with a second-level index; a data file's size is roughly bounded by a *chunk size* parameter (default 16MB per file) and the small, fixed index is stored at the beginning. The overall organization is similar to ISAM.

Since the generic persistence layer guarantees that items arrive in order and there is only a single writer, the data files are append-only, which leads to simple logic for `put_item`. Items are added by first writing file data, adding the offset and length to the index and finally by writing the timestamp into the index. This allows readers to co-exist with writers without much synchronization – a memory fence may be needed to ensure that item data appears before the timestamp in the index, depending on underlying hardware write ordering semantics.

Distributed filesystem backend: This backend is a variant of the `fs1` filesystem backend. It stores streams as whole files with a separate multi-level index directly on GPFS [6], which is already relatively well-tuned for streaming workloads. This backend also takes into account desired replication/failure semantics in placing data into proper filesets/storage pools with GPFS tools.

4 Evaluation

Here we present two sets of PTS evaluations. The first consists of system-level benchmarks testing the performance of pieces of our persistence architecture. The second is an application-based evaluation using a video-based surveillance application. We believe it is representative of a variety of live stream analysis applications in its basic structure and requirements.

4.1 System/Architectural Benchmarks

In order to measure the architectural overhead of our design, we perform several sets of targeted experiments. We start with a relative comparison of the storage backends, the lowest layer. After that we use our most lightweight backend and target the higher layers, showing the overhead for get operations, performing storage scaling tests with pickling handlers and adaptive load shedding, and finally showing the relative performance between live and stored gets in both pathological situations with no locality and locality-friendly scenarios.

The experiment in this section are performed on an x86_64 Linux 2.6.24 host with an Intel Core 2 Duo E6750 (2.66Ghz) processor and 4GB of DDR667 RAM. For storage results, we use the `fs1` backend (on a dedicated 300GB Seagate 7200.8 drive with XFS) since it has the lowest overhead and is self-contained.

Single Producer Storage Backend Overhead: As a baseline, we compare the relative overhead of the different storage backends using OProfile [10], a low-overhead, sampling-based system-wide profiling tool integrated with the Linux kernel capable of profiling un-instrumented binaries. Although the results are elided for space, we found that the overheads associated with an RDBMS like MySQL are very high for such workloads (e.g. 2-3x the user+kernel cycles compared to `*fs1` backends). These trends validate our decision (and intuition) to build lighter-weight, task-specific storage backends – `fs1` is only about 600 lines of C code and `gpfs1` is similar.

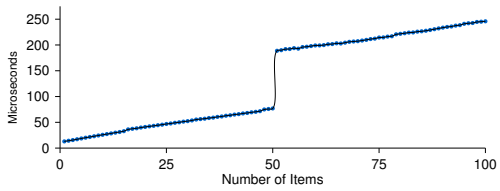


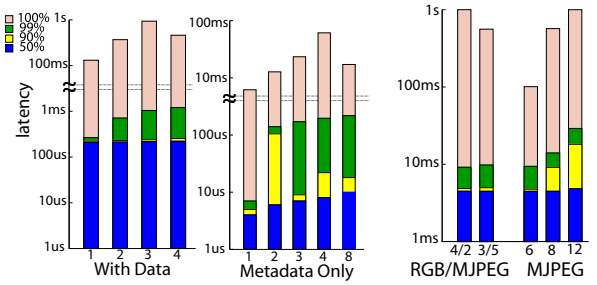
Fig. 3. Cost of gets with an increasing number of items: 50 live items, 100 stored items

Single Consumer Get Overhead: This experiment demonstrates baseline retrieval overheads of the storage layer with `fs1`. In Figure 3, we measure the cost of a get interval operation as we increase the maximum number of items in the interval to include stored items. We place 100 items in an persistent channel (using the `fs1` backend) which will hold up to 50 live items. Each item is 1024 bytes and the gets are performed over loopback TCP/IP networking. Each get is performed 10,000 times and we report the per-get averages (i.e. measured time / 10,000); the values are averaged over five runs (the standard deviations are less than 1% and thus not drawn on the graph). In the figure, get operations scale roughly linearly with the number of items requested until items must be fetched from the storage backend. At that point, each operation incurs a fixed cost of approximately 118 microseconds, and the roughly linear trend continues – obviously the additional cost of accessing stored items will vary widely depending on the storage backend and underlying storage media, but these figures show baseline overheads for `fs1` (when all data is in buffer cache).

Multiple Stream Scaling: This experiment shows how the `fs1` backend and our persistence architecture scale with increasing I/O rates by scaling the number of concurrent streams committed to the same disk. Figure 4(a) shows the results of multiple persistent channels simultaneously saving data to the same local XFS partition using the `fs1` backend with a chunk-size of 144MB. Each channel is filled by a producer putting 300KB RGB video frames at 30 frames per second, and the experiment runs for 36,000 items in each channel (20 minutes at 9MB/sec per stream). We scale the number of concurrent producers and show results for the normal configuration as well as results where data writes simply go to a file descriptor which throws away the data (`/dev/null`) – since the local disk will bottleneck long before other components, "no op" disk writes let us isolate the

overhead of other pieces of our architecture. We modified the backend to get the current time after an item's data is written out and modify the item's stored timestamp to provide an estimate of the total latency from the time it arrives in the channel to the time it is written out. We also set the level of queuing in the generic persistence layer to one, so each item is sent to the backend as soon as it arrives to the generic persistence layer. We present the results of item latencies in the form of several statistical percentiles (50%, 90%, 99%, 100%) because the general distribution is hard to characterize with a single number. For each percentile, we present the maximum among all producers. The vast majority of items have small latencies and then median times are quite low, but heavy I/O tends to induce a small tail of extreme outliers, particularly when the data rate streaming to disk is high (note the graphs' log scale and broken axes). The 99th percentile latencies seem to be primarily influenced by the amount of filesystem traffic and contention between multiple producers writing to a common disk. The absolute worst case measures (100%) have a high variance and are less meaningful across tests, because they are determined by a single high reading.

Multiple Stream Scaling with Pickling Handlers: The next experiment shows how applying pickling handlers to producers effectively reduces the data rate of streams committed to disk, enabling us to scale up the number of streams. We cannot reliably commit five concurrent 9MB/s streams using `fs1` with our particular hard disk and XFS, so we configure a pickling handler to compress each 300KB RGB video frame into a JPEG image. The average JPEG size is 20K, a fifteen-fold reduction in data committed to disk. Figure 4(b) shows the results for runs with 6, 8 or 12 producers all doing JPEG compression, and a mix of RGB and JPEG producers. The item latency now includes a JPEG compression step, performed by `libjpeg6b`, so the median item latencies are ∼4.5ms versus ∼210μs without the added compression and creation of temporary items. The raw measured cost of the JPEG compression by itself (without dynamic allocation of items or buffers) is ∼3.7ms per frame on average. Although the data rate of 12 MJPEG streams is still less than a single RGB stream, each producer requires at least 270MB of memory to hold 30 seconds of RGB data in the live channel (plus some extra memory for temporary JPEG items), and we run into some physical memory pressure around 14-15 streams. We could reduce the number of seconds of live data that each channel holds to add more producers, but we eventually hit a CPU bottleneck for JPEG compression before the disk bottlenecks. If we look at the all JPEG producer runs versus the



(a) RGB video streams (b) JPEG pickling

Fig. 4. Item latencies by statistical percentile

mixed runs, we see that the 99th percentile latencies are now more indicative of CPU contention versus disk contention; since we present the maximum value over all producers for each percentile and compression adds significant latency in the critical path for all JPEG streams, the storage latency for uncompressed items will generally be overshadowed by JPEG items. Again, the 100th percentile measures are less meaningful.

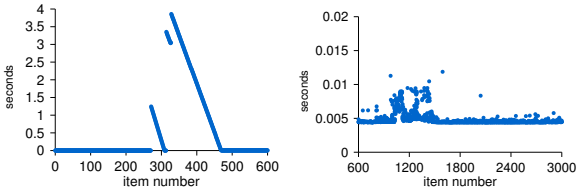


Fig. 5. 8 producers: latency before/after adjustment

Dynamic Load Adjustment with Pickling Handlers: This experiment shows how PTS can dynamically adjust to overload conditions. By measuring operation latencies in the generic persistence layer, the system can react by adding pickling handlers if the disk is overloaded or removing/changing them if the CPU is overloaded. The user could also provide several pickling handlers to compromise between stored item size and computational cost. In our current prototype we've implemented a simple proof-of-concept to illustrate the possibility of dynamic load adjustment: currently we only consider disk load and a single pickling handler, but if the item latency starts to increase heavily, some number of consumers automatically switch to using their pickling handlers until the overload is resolved. We ran successful tests starting with 6, 8 and 12 RGB video producers with JPEG pickling handlers; in all initial configurations (6, 8 and 12 uncompressed video streams), the load is too great for the local disk and the system would normally fall behind and never recover without removing producers. Figure 5 shows the item latencies for a single producer of the 8 producer run before and after it switches to JPEG frames.

Mixed Stored/Live Reader Workload: In order to demonstrate the performance impact of accessing stored versus live items, we vary the percentage of get operations requesting live versus stored items and measure the time to perform 10,000 get operations. Again we use 300KB RGB frames and perform get operations which request 50 items from a point in the channel determined by a probability distribution. 72,000 items are placed into the channel with a storage backend of `fs1`, and the last 200 items will stay in the live channel. Since the size of all of the items is ∼20.6GB, it is much larger than can fit in memory. We measure the cost of gets of exactly 50 items from some random point in the channel (containing all stored or all live items), and we limit the transferred data of each item to 100 bytes to eliminate the network transfer overhead and emphasize the overhead of stored data retrieval (all data is still read from disk when stored items are fetched). We vary the percentage of requests for live items from 0 to 100 and measure the total time to complete 10,000 requests with three different distributions – a uniform random distribution, a Zipf distribution ($s = 2.0$) and a binomial distribution ($p = 0.5$). The uniform random distribution exhibits no locality and rapidly bottlenecks by the raw speed of the disk. Both the Zipf and

binomial distributions exhibit a lot of locality and thus benefit from caching, scaling much better (in fact, their differences are too small to see on the graph scale). Figure 6 shows the average per-get time for the distributions (each point is also averaged over five runs). Although none of these test configurations are realistic models of an actual application, which might have many different clusters of "popular" historical data based on detected events, it does show the gamut of scaling behavior between pathologically bad and more locality-friendly workloads. Real workloads should fall in-between these extremes.

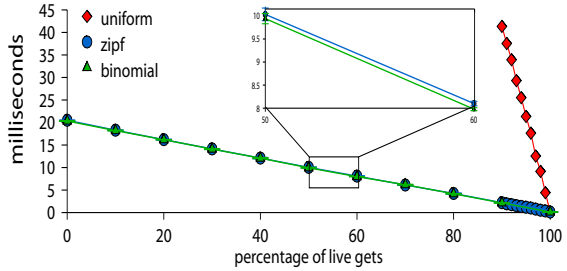


Fig. 6. Per-get time with historical query distribution

These system experiments show that the persistence architecture and primitives can be implemented in a lightweight manner, scaling to store relatively high data rate streams.

4.2 Application-based Evaluation

For an application-based evaluation, we use a representative kernel of a video surveillance application implemented on PTS. Although live stream analysis applications vary greatly, we believe that the core of a video analytics system can potentially represent a wide range of applications because the general structure of such applications is usually similar. Each application has some set of potentially high bandwidth streams (like video), a set of feature detectors running on these baseline streams to produce more structured high-level data, and a hierarchy of higher-level analysis modules which analyze and aggregate multiple potential feature streams and produce alerts/adjust future analysis/perform actions/etc. Some higher-level analysis will require historical data from archived streams.

Since distributed stream processing is at the heart of these applications, they require efficient low overhead stream transport with persistence management. Using a system like PTS simplifies the application logic significantly and provides greater functionality than non-stream oriented primitives, supporting richer domain-specific communication features, and transparent persistence of data streams. The video surveillance kernel implemented using PTS is only 670 lines of C code, not including command-line argument parsing or interfaces to OpenCV / `libjpeg`. Although it is subjective, the logic is very straightforward with PTS handling stream operations.

Components: Our PTS application consists of six parts: 1) the agents hosting video and sensor channels, 2) video data producers, 3) sensor data producers, 4) video feature detectors – face detection and optical flow, 5) random query agents,

and 6) feature aggregation agents. Figure 7 depicts the dataflow between components. Each agent hosts some number of persistent video and sensor channels. The video data is 225KB RGB video frames at ∼29fps, transformed to JPEG format using a pickling handler. The sensor data is random 1024 byte samples produced at ∼15fps. Video producers generate the RGB video frames by decoding MJPEG 3-4 minute compressed video files captured from TV and playing them back on a loop. The feature detectors get video frames one at a time from the channels and run either face detection or optical flow analysis on each frame. The optical flow process first converts each video frame to grayscale but only performs the subsequent optical flow computation on every other frame (the CPUs limited our ability to do full frame-rate optical flow). Each feature detector outputs a small 128-byte digest of the results into a channel. The random query agents generate random historical and live data queries on the video and sensor data with a specific probability distribution. Finally, there is a single feature aggregation agent for each feature detector type (face detector or optical flow); each agent gets the results from all feature detectors of the given type (corresponding to all video channels) and calculates the latency of processing video frames. All components process data in order and do not drop frames.

Topology: In our setup, we host four video channels and two sensor channels per agent, with one agent per cluster node. The four video producers and two sensor data producers corresponding to an agent are also colocated on the same node, although they are logically separate processes. This node will be decoding 4 MJPEG video streams to produce RGB video frames, encoding 4 MJPEG video streams from the same RGB frames for pickling handlers and committing all six data streams to disk. It is also responsible for serving video content to eight feature detectors (four of each type) and handling live and historical queries from the random query agents. The rest of the pieces run on independent nodes in different groupings. The feature detectors run two per node and host their own output channels locally. The random query agents run six per node (four video, two sensor) and both measurement agents run on separate nodes. For our experiments, we use two agents and eight video streams total. Table 1 summarizes this setup.

Component	Configuration	Total
Agent	1 per node, hosts 4 vid. / 2 sensor	2
Producers	6 per agent node, one per stream	12
Historical Query	6 per dedicated node	12
Face Detection / Optical Flow	2 per dedicated node	8 / 8
Feature Aggregators	1 per dedicated node	2

Table 1. Video surveillance experiment

Experimental Setup: Our experiments are run on 14 nodes from a cluster of dual-processor 64-bit Linux nodes. Each node has two Pentium 4-based Xeon 3.2Ghz processors with 1MB of L2 cache, 800Mhz FSB, 6GB of RAM, and IP over Infiniband networking (4x SDR). The nodes run RHEL 4u6, kernel 2.6.9-67.0.1.ELsmp (64-bit). The feature detector functionality is from OpenCV 1.0

and `libjpeg6b` is again used for JPEG operations. All binaries are built with gcc 4.1.2 with -O2 and -g. Persistent channels use `fs1`, writing to an ext3 filesystem on a Seagate ST373207LC 10k SCSI drive.

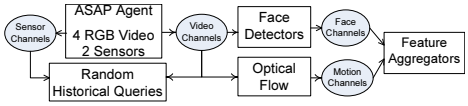


Fig. 7. Video surveillance components

Workload Characteristics: We perform five runs of each experiment, each normal run lasting six minutes and involving about 10,500 frames of video for each channel. Each query agent makes a video query every 100ms requesting a live or historical frame with equal probability. The historical frames' timestamps are chosen based on a probability distribution that is roughly Zipfian(we use a power-law distribution to approximate a situation where most video captured will be uninteresting with a few periodic events of high interest). The standard configuration has all streams converted to MJPEG before being stored to disk. The 1RGB configuration has one stream per agent (two total streams) stored to disk without compression to increase the size of the historical data-set. Similarly, the 2RGB configuration has two streams per agent stored without compression. Due to the large amount of RAM on each node, the set of files comprising all historical streams can fit in buffer cache easily on the shorter runs. Consequently, we also run some significantly longer experiments to ensure that the historical data set size is large enough to ensure that all requests cannot be serviced from RAM. All of the longer experiments have one RGB stream per agent.

Feature Detector Results: Figure 8 shows the feature detector latencies (in milliseconds) of several different configurations: the first two columns are the processing latency measurements at the face detector and optical flow feature detectors. The Agg columns show the measured latency at the aggregation agents.

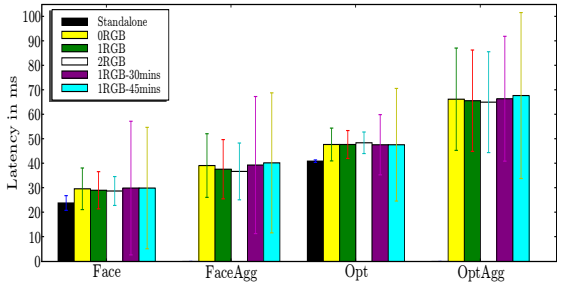


Fig. 8. Component latency in ms.

In both cases, the latency is calculated using the timestamp of the original video frame. The aggregation agents include another network hop since they consume the feature detection output data stream; in addition, the aggregation agents get the newest item from all feature detectors of a given type in sequence rather than concurrently, and each get call can potentially block. Consequently, the feature aggregation agents' latencies (and standard deviation) increase with the number of streams they are consuming. Having a separate thread handle each feature stream independently would alleviate this, but the most straightforward implementation (sequential) is entirely adequate for our target application and the performance is still quite good.

Face detection is less expensive than the optical flow calculation, so the latencies are as expected. The baseline costs for the stand-alone feature detectors run on the same datasets on an unloaded node are shown as "Standalone." The face detection standard deviation is slightly higher because the face detection operation takes a varying amount of time depending on how many potential faces are present in an image, while the optical flow is only dependent on the image resolution. The deviation drops slightly going from nRGB to $n+1$RGB because the processing load decreases slightly with the removal of JPEG encoding pickling handlers. The variance on all components increases on the longer runs because of the effect of historical queries that cannot fit into RAM.

Historical Query Results: Figure 9 shows the average time to make a random historical query (in milliseconds) for video streams. We separate the RGB and the compressed streams to show the effect of larger historical data sets. We can see that the average query time and variance grows as the amount of historical data grows – since the RAM size is constant, our locality gets worse as the total dataset grows. The compressed streams' latencies are affected too (but not as severely) because the same node and disk are used to host both types of streams.

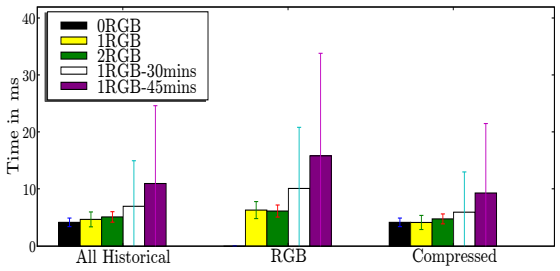


Fig. 9. Query time in ms.

To provide a frame of reference for our numbers, ASAP [4] is a video surveillance system implemented in Java; its published end-to-end latency results for live queries are between 135-175 ms, which in practice are perfectly adequate for the application domain. In our PTS-based evaluation, the highest latency we have measured for historical queries is 18ms (+/- 16ms). Although the PTS-based implementation and functionality are not directly comparable to the ASAP system, the structure of our application components is derived from the ASAP design and both evaluations were run on the same hardware using the same OpenCV library primitives for analytics. This does show that the results are promising and potentially provide headroom for higher fidelity. These preliminary results also show that the PTS runtime adds minimal overhead to the baseline stream processing operations which are at the core of such applications.

5 Related Work

As mentioned earlier, most work in alleviating higher-level concerns for live stream analysis applications comes in the form of stream processing engines or stream databases. These systems manage execution of stream analysis functionality and often use *continuous queries* in declarative query languages. There are a variety of relevant research systems like domain-specific Gigascope [11] and Hyperion [7] (for network monitoring) as well as more general-purpose systems

like TelegraphCQ [12], Borealis [13] (and its predecessor Aurora [14]), and Stanford's STREAM Data Manager [15]. IBM's Stream Processing Core [16] (part of System S) is another research system using a continuous query approach for "stream mining" (data mining on streaming data). General-purpose commercial systems include StreamBase and Coral8. Various extended SQLs and non-SQL based temporal query languages have been proposed over the last twenty years: CQL [17] and GSQL [11] are recent examples.

While these systems are impressive, they represent a different architectural approach than *temporal streams*. These systems provide centrally managed and controlled execution (often with high level query languages), while our system is a glue for loosely coupled systems of independent communicating components with no centralized control. Our system is also targeted at scenarios involving significant feature detection/analysis on streams such as audio and video, in contrast to SQL-like declarative query languages often more suited to domains with highly structured data like network monitoring or stock trading. The authors of the WaveScript language [18]/XStream engine [19] note that traditional stream database approaches are not well suited to signal analysis applications (audio, for example) and provide an augmented stream management system for isochronous signal processing. The Linear Road benchmark [20], the only standard benchmark for stream databases/stream processing engines, uses highly structured data for analysis and does not include signal analysis or feature detection from data sources like video.

Our approach does not impose a particular computational model on stream analysis applications; PTS only models stream *data* interactions, supporting arbitrary communication/data dependencies between components at the expense of being less declarative. In the end, we believe this tradeoff is acceptable given the added flexibility of general distributed applications (e.g. components can be developed independently/in different languages, hold internal state, utilize external resources, be integrated into existing systems, etc.). Our approach also provides a substrate for higher-level domain-specific solutions which raise the level of abstraction for a set of applications.

Ultimately, our approach represents another point in the design space balancing tradeoffs between flexibility/generality as well as performance and the level of abstraction. Our choices are similar to Distributed Data Structures [1] and BerkeleyDB [2], where some higher-level and heavyweight features of a full DBMS are traded off for a simpler, more procedural programmatic interface. In some ways our approach is similar to Boxwood [3], which provides distributed, managed data structures as a fundamental storage abstraction; in our case, the stream abstraction also serves as the storage interface.

PTS builds on the earlier work in Stampede^{RT} [5], which defines a programming model for live stream analysis applications. The Stampede^{RT} paper [5] surveys relevant work related to data-flow programming models like StreamIt [21] or TStreams [22] and the communications aspects of temporal stream abstractions: distributed programming systems/programming models, such as message-passing systems, distributed shared memory, RPC/RMI, group communication,

tuple spaces, or publish/subscribe systems. The workload of live stream analysis applications is unique and lends itself well to distributed programming, because stream processing has natural and explicit communication boundaries.

The general concept of processing streamed data as it is made available is fundamental – for example, the Unix pipe [23] is a ubiquitous streaming data flow abstraction, as are lazily-evaluated infinite lists in functional programming languages [24] or various reactive programming constructs. Hundreds of other abstractions in many diverse areas also model streams as a sequence of bytes or messages; this view is significantly different from the data-parallel array model typical in *stream programming*/GPGPU workand much closer to our preferred model for live stream analysis applications. Unlike most previous work, our abstract model of streams used in live stream analysis applications also includes a notion of time and random access.

Distributed programming models and runtime systems designed for processing/mining large amounts of data, such as MapReduce [25] and Dryad [26], often have similar concerns as live analysis applications, which makes many related ideas relevant to our domain. For example, Sawzall [27] provides a small domain-specific language for item-at-a-time processing of stored data sets within MapReduce, but it could also apply to streaming data. The key difference is that live stream analysis is continuous and data is explicitly time-related, while these aforementioned systems operate on stored data for batch processing. Although stored data is often streamed for processing, the time at which a streamed data item becomes available for processing is unrelated to the data itself. In live stream analysis, time is semantically significant. Also, systems such as MapReduce are generally optimized for throughput over latency, are not limited to one-pass processing, and often have foreknowledge of the size of a dataset to partition processing.

6 Conclusion

Many critical applications involve continuous and computationally intensive analysis on live streaming data and also require access to historical data. While distributed programming support for traditional high-performance computing applications is fairly mature, existing solutions for live stream analysis applications are still in their early stages and, in our view, inadequate. We have described *Persistent Temporal Streams* (PTS), which are specifically designed to address the needs of these distributed applications by providing a higher-level unified abstraction for dealing with live and archived streams. The *channel* primitive of our PTS system unifies transport, manipulation and storage of streams. We have presented a detailed description of the PTS system architecture and elements of its implementation. Finally, we have presented a set of system-level benchmarks looking at pieces of the system in isolation as well as a whole-system, application-based evaluation. Although preliminary, these results show that the PTS architecture can be implemented in a lightweight manner and provide good performance in a video-surveillance application scenario based.

References

1. Gribble, S.D., Brewer, E.A., Hellerstein, J.M., Culler, D.: Scalable, Distributed Data Structures for Internet Service Construction. In: Proceedings of OSDI'00. (2000) 22
2. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley DB. In: Proceedings of USENIX ATC '99. (1999) 43
3. MacCormick, J., et al.: Boxwood: Abstractions as the Foundation for Storage Infrastructure. In: Proceedings of OSDI '04. (2004) 8
4. Shin, J., et al.: ASAP: A Camera Sensor Network for Situation Awareness. In: Proceedings of OPODIS '07. (December 2007)
5. Hilley, D., Ramachandran, U.: Stampede^{RT}: Programming abstractions for live streaming applications. In: Proceedings of ICDCS '07. (June 2007) 65
6. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of FAST '02, Berkeley, CA, USA (2002) 19
7. Desnoyers, P., Shenoy, P.: Hyperion: High Volume Stream Archival for Retrospective Querying. In: Proceedings of USENIX ATC '07. (June 2007) 45–58
8. Mills, D.L., Thyagarajan, A.: Network Time Protocol Version 4 Proposed Changes. EE Deptartment Report 94-10-2, University of Delaware (October 1994)
9. Sweeney, A., et al.: Scalability in the XFS File System. In: Proceedings of USENIX ATC '96. (1996) 1–14
10. Levon, J., Elie, P.: OProfile: A System Profiler for Linux http://oprofile.sf.net.
11. Cranor, C., et al.: Gigascope: A Stream Database for Network Applications. In: Proceedings of SIGMOD '03, ACM Press (2003) 647–651
12. Chandrasekaran, S., et al.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proceedings of CIDR '03. (January 2003)
13. Abadi, D.J., et al.: The Design of the Borealis Stream Processing Engine. In: Proceedings of CIDR 2005, Asilomar, CA (January 2005)
14. Balakrishnan, H., et al.: Retrospective on Aurora. The VLDB Journal **13**(4) (2004) 370–383
15. Arasu, A., et al.: STREAM: The Stanford Data Stream Management System. In Garofalakis, M., Gehrke, J., Rastogi, R., eds.: Data Stream Management: Processing High-Speed Data Streams. Springer (August 2008) *(in press)*.
16. Amini, L., et al.: SPC: A Distributed, Scalable Platform for Data Mining. In: Proceedings of DMSSP '06, ACM (2006) 27–37
17. Arasu, A., Babu, S., Widom, J.: The CQL Continuous Query Language: Semantic Foundations & Query Execution. The VLDB Journal **15**(2) (2006) 121–142
18. Girod, L., et al.: The Case for a Signal-Oriented Data Stream Management System. In: Proceedings of CIDR '07, Monterey, CA (January 2007)
19. Girod, L., et al.: XStream: A Signal-Oriented Data Stream Management System. In: Proceedings of ICDE '08, Canc'un, M'exico (April 2008)
20. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark. In: Proceedings of VLDB '04, VLDB Endowment (2004) 480–491
21. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A Language for Streaming Applications. In: Proceedings of CC '02, Springer-Verlag (2002) 179–196
22. Knobe, K., Offner, C.D.: TStreams: How to Write a Parallel Program. Technical Report HPL-2004-193, HP Laboratories Cambridge (October 2004)
23. Ritchie, D.M., Thompson, K.: The UNIX time-sharing system. Communications of the ACM **17**(7) (1974) 365–375
24. Jones, S.P., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge, UK (April 2003)
25. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of OSDI'04. (2004) 10
26. Isard, M., et al.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: Proceedings of EuroSys '07, ACM (2007) 59–72
27. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming **13**(4) (2005) 277 – 298