StreamFlex: High-throughput Stream Programming in Java

Jesper H. Spring
Ecole Polytechnique
Fédérale de Lausanne

Jean Privat
Computer Science Dept.
Purdue University

Rachid Guerraoui
Ecole Polytechnique
Fédérale de Lausanne

Jan Vitek
IBM Research
and
Computer Science Dept.
Purdue University

Abstract

The stream programming paradigm aims to expose coarse-grained parallelism in applications that must process continuous sequences of events. The appeal of stream programming comes from its conceptual simplicity. A program is a collection of independent filters which communicate by the means of uni-directional data channels. This model lends itself naturally to concurrent and efficient implementations on modern multiprocessors. As the output behavior of filters is determined by the state of their input channels, stream programs have fewer opportunities for the errors (such as data races and deadlocks) that plague shared memory concurrent programming. This paper introduces STREAMFLEX, an extension to Java which marries streams with objects and thus enables to combine, in the same Java virtual machine, stream processing code with traditional object-oriented components. STREAMFLEX targets high-throughput low-latency applications with stringent quality-of-service requirements. To achieve these goals, it must, at the same time, extend and restrict Java. To allow for program optimization and provide latency guarantees, the STREAMFLEX compiler restricts Java by imposing a stricter typing discipline on filters. On the other hand, STREAMFLEX extends the Java virtual machine with real-time capabilities, transactional memory and type-safe region-based allocation. The result is a rich and expressive language that can be implemented efficiently.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010…$5.00

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems.

General Terms Languages, Experimentation.

Keywords Real-time systems, Java virtual machine, Memory management, Ownership types, Stream processing.

1. Introduction

Stream processing is a programming paradigm fit for a class of *data driven* applications which must manipulate high-volumes of data in a timely and responsive fashion. Example applications include video processing, digital signal processing, monitoring of business processes and intrusion detection. While some applications lend themselves naturally to a distributed implementation, we focus on single node systems and, in particular, on programming language support for efficient implementation of systems that require microsecond latencies and low packet drop rates.

In a stream processing language, a program is a collection of *filters* connected by *data channels*. Each filter is a functional unit that consumes data from its input channels and produces results on its output channels. In their purest form, stream processing languages are ideally suited to parallel implementations as the output behavior of a filter is a deterministic function of the data on its input channels and its internal state. As filters are independent and isolated from one another, they can be scheduled in parallel without concern about data races or other concurrent programming pitfalls that plague shared memory concurrent programs. The appeal of this model is evidenced by a number of stream processing languages and systems include Borealis [2], Infopipes [10], StreamIt [32] and Gryphon [9]. These languages have a long lineage which can be traced back to Wadge and Ashcroft's Lucid [5] data flow language and, to some extent, to the Esterel and Lustre family of synchronous languages [17, 13].

High performance stream processing systems that deal with large volumes of data should be designed to fulfill at least the following two key requirements (out of the eight requirements given in [31]):

- *keep the data "moving"*: this means messages must be processed with as little buffering as possible;
- *respond "instantaneously"*: any substantial pause may result in dropped messages and must be avoided.

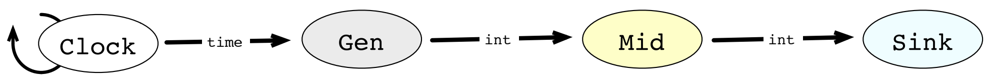
Stream processing applications, if they are to meet their non-functional requirements, must be engineered with great care. Moreover, the underlying infrastructure, the stream processing language and its runtime system, must be designed and implemented so as to avoid inherent inefficiencies. The challenge for stream processing language designers is to provide abstractions that are expressive enough to allow rapid development of applications without giving up on efficiency and predictability. Consider, for instance, a network intrusion detection system with a number of detection modules defined as filters over a stream of network packets. Throughput of the system is crucial to process event streams at rates in the hundreds of MBps and latency is important to avoid dropping packets and thereby possibly failing to detect attacks. These requirements have obvious implications on both user-defined streaming code and the underlying infrastructure.

Unlike stream programming languages such as StreamIt, Lucid or Esterel, we propose to provide only a limited set of new abstractions for stream processing and leverage a host language for its general purpose programming constructs. This has the advantage of providing a familiar framework for developers but comes at the cost of having to deal with the impedance mismatch between the requirements of stream processing and features provided by the host language. In this paper, we introduce STREAMFLEX, an extension to the Java programming language with support for high-throughput stream processing. Java is a pragmatic choice as it is a mainstream language with a wealth of libraries and powerful IDEs. Not only does this make it easier for programmers to accept the new abstractions, but it opens up opportunities for seamlessly integrating stream processors in larger applications written in plain Java. However, Java presents significant implementation challenges. In fact, it is not obvious at first that Java is at all suitable for applications with stringent quality of service requirements. A Java virtual machine implementation is the source of many potential interferences due to global data structures, just-in-time compilation and, of course, garbage collection. In [30], we have performed empirical measurements of the performance of standard and real-time garbage collectors. Our stop-the-world collector introduced pauses of 114 milliseconds; using a real-time collector pause time went down to around 1 milliseconds at the expense of application throughput. In both cases, the pauses and performance overheads were too severe for some of our target applications.

The STREAMFLEX programming model is inspired both by the StreamIt language and, loosely, by the Real-time Specification for Java [11]. STREAMFLEX includes changes to the virtual machine to support real-time periodic execution of Java threads, a static type system which ensures isolation of computational activities, a type-safe region-based memory model that permits filters to compute even when the garbage collector is running, and software transactional memory for communication between filters and Java. The contributions of this paper are the following:

- **Programming Model:** We present a new programming model for real-time streaming which allows developers to write stream processors in Java. The proposal *does not* require changes to Java syntax. Filters are legal Java programs and, STREAMFLEX can be manipulated by mainstream IDEs such as Eclipse. A number of standard libraries and API can be used within filters, and filters can be integrated into Java applications.
- **Filter Isolation:** STREAMFLEX filters are isolated components that communicate by non-blocking bounded channels. Software transactional memory is used for synchronization of shared data channels.
- **Zero-Copy Message Passing:** The STREAMFLEX type system allows mutable data objects to be transefered along linear filter pipelines without requiring copies.
- **Implementation:** We have implemented our proposal on top of a real-time virtual machine and extended a version of the `javac` compiler to enforce the STREAMFLEX type system.
- **Evaluation:** We present an empirical evaluation of our system. We show that STREAMFLEX programs outperform the corresponding Java variants. We also show our implementation achieves a high degree of predictability.

STREAMFLEX is built on top of the Ovm virtual machine [4] which comes with an implementation of the Real-time Specification for Java (RTSJ). While we initially envisaged using the RTSJ directly, we found the API too complex and error prone for our needs. Instead, we based STREAMFLEX on a simpler real-time programming model called Reflexes [30]. Reflexes already provide some of the features that are required by STREAMFLEX, namely, real-time periodic threads, region-based allocation and software transactional memory. The main differences are in the programming model, Reflexes are stand alone components with no support for cooperation across multiple Reflexes. The STREAMFLEX type system is an extension to the ScopeJ type system of [34]. The relationship with previous work is further detailed in Section 8.



```
class Gen extends Filter {
  IntChannel out; TimeChannel t;
  int counter;
  void work() {
   out.put(counter++);
  }
}
```

(a) Filter Gen

```
class Mid extends Filter {
  IntChannel in, out;

  void work() {
   out.put(in.take());
  }
}
```

(b) Filter Mid

```
class Sink extends Filter {
  IntChannel in;

  void work() {
   print("got "+in.take());
  }
}
```

(c) Filter Sink

Figure 1. Example of a STREAMFLEX pipeline.

2. Stream Processing

This section introduces the main concepts of STREAMFLEX. A stream processing application is built out of a number of filters connected by channels to form a *filter graph*. Filter graphs are executed by the STREAMFLEX runtime engine which manages both concurrency within any given graphs and across multiple graphs.

The basic computational unit for processing one or more data streams is the *filter*. A filter is a schedulable entity consisting of user-defined persistent data structures, typed input and output channels, an activity and an (implicit) trigger on channel states. An activity can become schedulable any time new data appears on one of the associated filter's input channel, more precisely a filter becomes schedulable when it's trigger predicate evaluate to true. The STREAMFLEX scheduler is responsible for releasing the activity without any guarantee of timeliness. It only promises that any schedulable activity will eventually be released. If programmers require timely execution of filters, they must use *clocks*. Clocks generate events on time channels. When filter is connected to a clock, the scheduler arranges for the filter to be released periodically.

Simple Example. Figure 1 presents a purposefully simple STREAMFLEX graph. User-defined filters are Java classes that extend the pre-defined Filter class. The filters in this example are arranged to form a simple pipeline. Activities are implemented by the work() method of each filter, these methods are invoked repeatedly by the STREAMFLEX scheduler when the filter's trigger predicate evaluates to true. By convention, work() methods are expected to eventually yield—in most applications it would be a programming error for an activity to fail to terminate as this could prevent evaluation of other triggers and block the entire pipeline.

The first filter in Figure 1 is an instance of the built-in Clock class. The clock periodically puts a signal on the timing channel of the first filter, an instance of the user-defined class Gen shown in Figure 1(a). Gen is a stateful filter with a single integer output channel. Like any Java class, a filter may have instance variables and methods. In this case, Gen keeps a counter which it increments at each release before putting the counter's value on its output channel. The filter in Figure 1(b) is simple and stateless. It consists of two integer channels, one for input and one for output. Its only behavior is to read a single value from its input and put it on its output channel. Finally, Figure 1(c) is a seemingly innocuous filter for printing the value received on its input channel.

Constructing Filter Graphs. STREAMFLEX filter graphs are constructed by extending the built-in StreamFlexGraph. The protocol is simple: the constructor creates and connects all filters and clocks and then calls validate() to verify that the graph is well formed. Once validate() has been called the graph cannot be changed. Figure 2 demonstrates the construction of the graph of Figure 1.

```
class Simple extends StreamFlexGraph {
  Simple(int period) {
    Clock clk = makeClock(period);
    Filter gen = makeFilter(Gen.class);
    Filter mid = makeFilter(Mid.class);
    Filter sink = makeFilter(Sink.class);
    connect(clk, gen, "timer");
    connect(gen, "out", mid, "in", 1);
    connect(mid, "out", sink, "in", 1);
    validate();
  }
}
```

Figure 2. Constructing the filter graph of Figure 1.

Interaction with Java. Running a stream processing application on a standard JVM would uncover a number of drawbacks of Java for applications with any quality of service requirements. For starters, the print() statement in Figure 1 allocates a StringBuffer and a String at each release. Eventually filling up the heap, triggering garbage collection and blocking the filter for hundreds of milliseconds. Another issue is the default Java scheduler may decide to preempt a filter at any point of time in favor of a plain Java thread. StreamFlex ensures low-latency by executing filters in a partition of the JVM's memory which is outside of the control of the garbage collector. This allows the StreamFlex scheduler to safely preempt any Java thread, including the garbage collector. Activities can thus run without fear of being blocked by the garbage collector. Another danger is potential for priority inversion. To prevent synchronization hazards, such as an activity blocking on a lock held by a Java thread which in turn can block for the GC, filters are isolated from the Java heap. Non-blocking synchronization in the form of software transactional memory is used when Java threads need to communicate with filters.

With these protections in place, integrating a StreamFlex filter graph into a Java application is simply a matter of having plain Java code invoke public methods of a filter.

Memory Model. We mentioned that in StreamFlex filters are protected from interference from the garbage collector. But then, how does StreamFlex deal with the allocations occurring in Figure 1(c)? The answer is that we use a region-based allocation scheme. Each time an activity is released, a new memory region is created and, by default, all newly allocated objects are created in that region. When the activity terminates, at the return of the corresponding work() method, all objects allocated in the region are reclaimed at the same time. Region-based allocation allows programmers to use standard Java programming idioms without having to incur disruptive pauses.

In terms of memory management, a StreamFlex graph is thus composed of three kinds of objects: *stable* objects, *transient* objects and *capsules*. Stable objects include the filter instance itself and its internal state, their lifetime is equal to that of the filter. Transient objects live as long as the activity. Finally, capsules are data objects used in messages and are managed by the StreamFlex runtime engine. Specifying whether an object is stable, transient or capsule is done at the class level. By default, data allocated by a StreamFlex thread is transient. Only objects of classes marked stable or that extend Filter will persist between invocations. Stable objects must be managed carefully by the programmer as the size of the stable area is fixed and the area is not garbage collected. Figure 3 gives an abstract representation of memory. In order to preserve type safety, the StreamFlex compiler enforce constraints on patterns of references across the different partitions. Arrows in Figure 3 indicates allowed directionality for references.

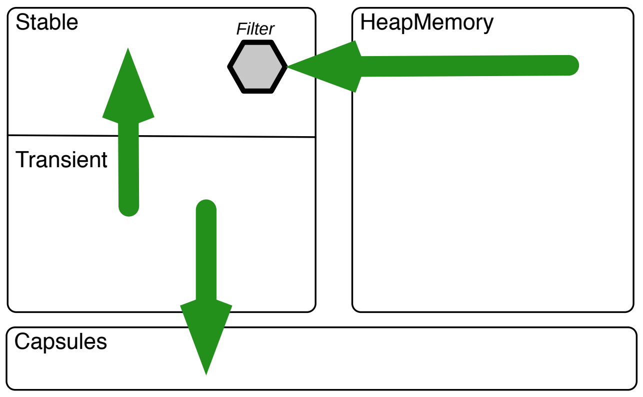


Figure 3. Valid cross-regions references. Arrows indicate allowed reference patterns between objects allocated in different regions.

StreamFlex allows allocation and transfer of user-defined data types along channels. This should be contrasted to systems that limit communication to primitives and arrays. The advantage of using primitives is that one does not need to worry about memory management or aliasing for data transferred between filters, on the other hand if, for example, one wants to communicate complex numbers, they have to be sent as a pair of floats over a float channel. While this may be acceptable in the case of simple data, encoding richer data structures is likely to be cumbersome. In StreamFlex, a channel can carry objects, thus Figure 4 shows a natural way to express channels of complex numbers.

Primitive types only:

```
float realv = in.take();
float image = in.take();
```

StreamFlex

```
Complex c = in.take();
```

Figure 4. Communicating complex numbers as pairs of primitive numbers over a channel. In StreamFlex complex numbers can be communicated as objects.

As suggested above, there are good reasons for restricting the data types transferred on a channel. As soon as one adds objects to the computational model, it is necessary to provide support for their automatic memory management. The problem is compounded if garbage collection pauses are to be avoided. Consider for example the filter Err in Figure 5. This filters retains a reference to a value that was taken from a channel and puts the same value down its output channel. When is it safe to reclaim the instance of Complex? There is no obvious way, short of garbage collection, of ensuring

```
class Err extends Filter {
  Complex retain;
  void work() {
    out.put( retain = in.take() );
  }
}
```

Figure 5. When is it safe to reclaim the retain?.

that the virtual machine will not run out of memory. The StreamFlex approach is to use a static type discipline and rule out this program as ill-typed.

Example: Processing biological data. To conclude, we consider a slightly more sophisticated example inspired by a stochastic algorithm for protein backbone assignment [33]. The class MutatingFilter of Figure 6 processes a stream of Strand objects which are capsules representing a sequence of protein pseudoresidues and a score summarizing the "quality" of an assignment [33]. The filter takes a Strand object, performs a random mutation and evaluates the resulting score. If this score indicates an improvement, the data is copied to the Strand object and the Strand is sent on to the next filter. The example illustrates the use of capsules. Here a Strand extends the Capsule class. Capsule can contain arrays and primitive fields. What is most noteworthy about this code is that it looks exactly like normal Java.

```
class Strand extends Capsule {
  final double[] residues;
  double score;
}

class MutatingFilter extends Filter {
  Channel<Strand> in, out;
  void work() {
    Strand strand = in.take();
    double[] rds, mutated;
    rds = strand.residues;
    mutated = new double[rds.length];
    mutate(rds, mutated);
    double score = compute(mutated);
    if (score < strand.score) {
      arraycopy(mutated,rds);
      strand.score = score;
    }
    out.put(strand);
  }
}
```

Figure 6. An example of a filter using capsules to communicate with other filters.

3. The Programming Model

This section gives a more detailed presentation of the StreamFlex programming model.

3.1 Graphs

A StreamFlexGraph is the abstraction of a stream processing application. This class must be subclassed, and the programmer must implement at least one constructor and possibly redefine the start() method. The constructor is responsible of creating filters and connecting them. Once a graph is fully constructed, the validate() method must be invoked to check consistency of the graph. The checking involves verifying that all channels are connected to filters with the right types, that there is sufficient space available for the stable and transient stores of filters, and that clocks are given periods supported by the underlying virtual machine.[1] Once validate() returns the graph cannot be changed—supporting dynamic filter graphs is left for future work. The other methods of the class are all declared protected and support the reflective creation of filters and channels. Reflection is needed because the creation of both channels and filters must be performed in specific memory areas under the control of the StreamFlex runtime, it would be unsafe to construct any of these objects on the heap.

```
public abstract class StreamFlexGraph {
  ...
  public void start();
  public void stop();
  protected void validate() throws StreamFlexError;
  protected Filter makeFilter(Class f);
  protected Filter makeFilter(Class f, int stbSz,
    int tranSz);
  protected Clock makeClock(int periodInMicrosecs);
  protected void connect(Clock src, Filter tgt,
    String tgtField);
  protected void connect(Filter src,
    String srcField, Filter tgt, String tgtField, int size);
}
```

Figure 7. Graph interface (extract).

3.2 Filters

The abstract class Filter, shown in Figure 8 provides the basic functionality for stream processing. A filter's activity is specified by providing an implementation of the work() method of the abstract class. A filter can, optionally, implement the boolean trigger() method that indicates whether an activity is schedulable or not.

```
public abstract class Filter {
  ...
  public abstact void work();
  protected boolean trigger();
}
```

Figure 8. Filter interface (extract).

The previous section introduced the notion of partitioned memory for Filters. Objects allocated by a filter can have either a lifetime that is bound to the lifetime of the entire

[1] Most operating systems can go down to the millisecond. In our experiments, we use a release of Linux that has been patched to provide microsecond periods.

filter graph or to the activation of the work() method. Operationally, this is achieved by associating the filter with two memory regions, shown pictorially in Figure 9, the *stable* region is used for allocation of persistent data while the *transient* region is used as a scratch pad. The filter instance itself is allocated in stable memory to ensure that it is also protected from the garbage collector.

The default allocation context while executing a filter's work() method, or any of its public methods, is always transient memory. Thus, by default any occurrence of **new** will result in allocation of a transient object. In order to allocate an object in stable memory, the class of that object must implement the Stable marker interface. The assumption behind this design choice is that allocation of persistent state is the exception and that there is only a small number of object types that will be used in stable memory.

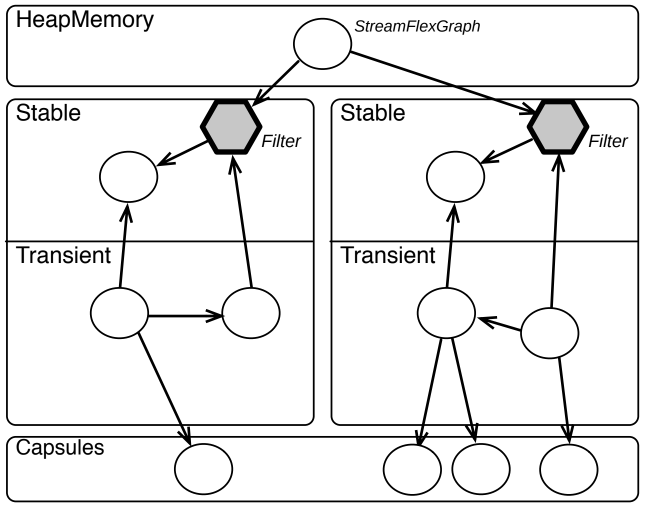


Figure 9. Memory model of a StreamFlexGraph application having two filters.

When the work() method returns, all transient data allocated during its execution is reclaimed in constant time.[2]

In the current version of the STREAMFLEX, the sizes of both memory regions are fixed and are chosen at filter instantiation. Supporting dynamically sized regions is possible but has not been implemented.

Exceptions that escape from the work() method must be treated specially as the exception object and stack trace information are allocated in transient memory. To avoid creating a dangling pointer, the exception will be discarded and replaced by a BoundaryError object which will terminate the execution of the STREAMFLEX graph.

The implementation of filters is subject to a number of constraints that aim to prevent dangling pointer errors. These are detailed in Section 4.

[2] Finalizers are not supported for objects allocated in transient memory, allowing them would violate the constant time deallocation guarantee. Considering the isolated nature of filters, they are of dubious value anyway.

3.3 Channels

A Channel is a fixed-size buffer. Figure 10 gives an overview of the channel interface which is straightforward. Each channel has methods for querying the number of available datums, for adding a value at the end, taking a value from the front of the buffer, and finally for returning a value to the front. Channels are strongly typed. STREAMFLEX supports generic channels, time channels as well as primitive channels for all of Java's primitive types (IntChannel, FloatChannel, etc.).

```
public class Channel<T extends Capsule> {
  ...
  public int size();
  public put(T val);
  public T take();
  public void untake(T val);
}

public class TimeChannel {
  public double getTime();
  public int missedDeadlines();
}
```

Figure 10. Channel interface (extract).

Operations performed on the set of channels attached to a filter are atomic. From the point the work() method is invoked to the point where it returns, all of the put/take operations are buffered, it is only after work() returns that all channels are updated.

The current version of STREAMFLEX does not support growable channels and, in case of overflow, silently drops capsules. Other policies are being considered but have not been implemented. Variable sized channel, for example, can be added if users are willing to take the chance that resize operation triggers a garbage collection (an unlikely but possible outcome).

Timing channels are a special kind of primitive channel. Their size is fixed to 1 and the replacement policy is to retain the oldest value and keep a counter for overflows. The only components allowed to write to a time channel are clocks. Filters have access to only two methods: getTime() which returns the first unread clock tick in microseconds and missedDeadlines() which returns the number of clock ticks that have been missed. All time channels of a filter are emptied when the work() method returns and their overflow counters are reset to zero.

3.4 Triggers

A trigger is a predicate on the state of the input channels of a filter. A filter is said to be schedulable if its trigger evaluates to true. The default trigger supported by all STREAMFLEX filter is to yield true if data is present on any of the filter's input channels. More sophisticated trigger expressions are planned but have not yet been incorporated in the system. A simple one is to support rate specifications as proposed by [32]. Rate specifications let users define the minimal accept rate for channels. The trigger will only yield true if sufficient input is available. A user-defined trigger() method which evaluates an arbitrary predicate over the channels of the filter could be used to ensure, for example, the presence of data on all channels.

3.5 Clocks

To allow for periodic filters, StreamFlex provides Clocks. Clocks are special kind of filters which can not be subclassed or extended. They have a single output TimeChannel. Like all other filters, they are created reflectively, using the makeClock() method, and configured with a period in microseconds. During execution, a clock outputs a tick on its time channel at the specified period.

In the current StreamFlex prototype, a periodic real-time thread is associated with each clock. At each period the following operations are performed: a new time is written to the time channel, the trigger of the attached filter is evaluated and if it yields true the filter's work() method is evaluated. The evaluation strategy is eager in the sense that, when a work() method returns the same thread will try to evaluate all of the subsequent filters. To ensure timeliness, the Clock instance should be configured with a period that is larger than the maximum processing time of the sequence of filters that will be evaluated at each release.

3.6 Capsules

Subclasses of the built-in class Capsule are used as messages on channels. Capsules are designed with one key requirement: allow for zero-copy communication in a linear filter pipeline. This seemingly simple requirement turns out to be challenging as it is necessary to answer the twin questions of where to allocate capsules, and when to deallocate them. Capsules cannot be allocated in the transient memory of a filter as they would be deallocated as soon as the filter's work() method returns. They should not be allocated in a filter's stable memory as that area would quickly run out of space. Instead, we allocate them from a pool managed by the StreamFlex runtime. A capsule is deallocated if it was taken from a channel and, by the time work() returns, not been put back onto another channel.

Capsules are user-defined classes that must abide by certain structural constraints. They are restricted to having fields of primitive types or of primitive arrays types. They are constructed reflectively by the StreamFlex runtime, as they must be allocated in a special memory area.

While we strive for zero-copy communication, there is one case where copying capsules is necessary, this is when a filter needs to put the same capsule on multiple output channels. The copies are done when modifications to channels are published after the work() method returns.

3.7 Borrowed Arguments and Atomic Methods

Interacting with Java presents two main challenges. Firstly, it is necessary to ensure that the interaction does not cause the filters to block for the GC. Secondly, we would like to avoid having to copy data transferred from the Java world. We achieve these two goals with features inherited from the underlying Reflexes system [30]: borrowed arguments and atomic methods.

StreamFlex prevents blocking operations by replacing lock-based synchronization with a simple form of transactional memory called *preemptible atomic regions* in [24]. Any method on a filter that is invoked from plain Java code must be annotated @atomic. For such methods, the StreamFlex runtime ensures that their execution is logically atomic, but with the possibility of preemption if the filter is released. In which case the Java call is aborted and transparently re-executed later.

The @borrow annotation is used to declare a reference to a heap-allocated object that can be read by a filter with the guarantee that it is in consistent state and that the filter will not retain a reference to it. The guarantee is enforced by the StreamFlex type system discussed in Section 4.[3]

Figure 11 shows a filter with an atomic method that takes a borrowed array. This method can be safely invoked from Java code with a heap-allocated argument.

```
public class Writeable extends Filter {
  ...
  @atomic public int write(@borrow short[] b) {
    for (int i=0,j=0; i<b.length; i++)
      if (b[i]>64) data[j++]=b[i];
  }
}
```

Figure 11. The method write() is invokable from Java. The method is declared @atomic and the parameter b is *borrowed* as it references a heap allocated object.

4. Type System

We present a type system that ensures memory safety by preventing dangling pointers. The StreamFlex type system is a variant of our work on ScopeJ [34] and Reflexes [30]. We give a succinct presentation of the type system. The StreamFlex type system is an *implicit* ownership type system. As in other ownership type systems [14] there is a notion of a dominator that encapsulates access to a subgraph of objects—in our case every Filter instance encapsulates all objects allocated within its stable and transient memory regions. The type system ensures that references to objects

[3] The @borrow is retained for backwards compatibility with [30], the StreamFlex type system treats all reference arguments as implicitly borrowed.

owned by a filter are never accessed from outside and prevents dangling pointers caused by references to transient objects from stable ones. This remainder of this section reviews the constraints imposed on the implementation of filters.

4.1 Partially Closed-World Assumption

A requirement for type-checking a filter is that all classes that will be used within the filter must be available. We refer to this as a *partially* closed-world assumption, as there are no constraints on code outside of a filter. Classes used within a filter fall in one of three categories: *stable*, *transient* and *capsule* classes. The reachable class set (RCS) denotes the union of these sets of classes. The first task of the checker is to compute the RCS. This done by a straightforward reachability analysis starting with subclasses of Filter and Capsule. Rapid type analysis [8] is used to resolve the targets of method calls. The following informal rules define the RCS and are implemented in a straightforward way in the checker.

$\mathcal{D}1$: *Any subclass of* Filter *or* Capsule *is in RCS.* □

$\mathcal{D}2$: *If class* C *is in RCS, all parents of* C *are also in RCS.* □

$\mathcal{D}3$: *Given the instance creation expression* new C(...) *in class* D*, if* D *is in RCS then* C *is in RCS.* □

$\mathcal{D}4$: *Given an invocation of a* static *method* C.m() *in class* D*, if* D *is in RCS then* C *is in RCS.* □

The type checker validates all classes in the RCS. Taken together rule $\mathcal{D}1$-3 ensure that any object that can be created while executing a method of a filter are in RCS. Furthermore, the defining class of any static method that can be invoked will be added to RCS. Native methods are currently allowed on a case by case basis and are validated by hand.

Observe that the above rules do not prevent a class in the RCS from having subclasses which are not in RCS—except for filters and capsules. Basically, it means that the closed-world assumption does not preclude the modular evolution of software through subclassing which is standard strategy used to evolve Java programs. While these rules are an accurate description of the current implementation, they are stricter than necessary. A more precise analysis would only consider reachable methods, while our analyzers checks all methods of a class. Similarly, for static methods it is not necessary to add the defining class to the RCS, one only need to verify the methods reachable from static method being invoked. While the imprecision has not affected the applications we have considered, we plan a more precise analysis, along the lines of [6], for future versions of the system.

4.2 Implicit Ownership

The key property to be enforced by the type system is that all objects allocated within a filter must be properly encapsulated. No object allocated outside of a filter may refer to a stable or transient object of that filter. Conversely, no stable or transient object may refer to an object allocated outside of the filter.

$\mathcal{R}1$: *The type of arguments to* public *and* protected *methods of any subclass of* Filter *can be primitive types as well as arrays of primitive types. Returns types of these methods are limited to primitive types. The type of* public *and* protected *fields of any subclass of* Filter *are limited to primitive types.* □

$\mathcal{R}1$ ensures that methods and fields visible to clients of a filter do not leak references across the filter boundary. To be safe the rule requires encapsulation in both direction. Arrays are a special case described in Section 4.4.

Within a filter is it necessary to prevent a stable object from retaining a reference to a transient one, as this would lead to a dangling pointer. This enforced by making it illegal for a stable object to ever have a reference of transient type, and similarly for static variables (see Section 4.3). This is done at the class granularity. If a class is declared stable, then it can only refer to other stable classes. Again arrays are a special case discussed in Section 4.4.

Since the type system tracks classes, rather than objects as would be done by a more precise escape analysis, we must ensure that the subtyping relation cannot be used by transient types to masquerade as stable types. $\mathcal{D}5$ makes it so that any subtype of stable type is stable.

$\mathcal{D}5$: *Any class in RCS (transitively) implementing the marker interface* Stable *is stable. The* Filter *class is stable.* □

$\mathcal{R}2$: *An instance field occurring either in a stable class or in a parent of a stable class in RCS must be of either a primitive type or a stable type.* □

4.3 Static Reference Isolation

Enforcing encapsulation requires that communication through static variables be controlled. Without any limitations, static variables could be used to share references across encapsulation boundaries and open up opportunities for memory errors.

A drastic solution would be to prevent code in RCS from reading or writing static reference variables. Clearly this is safe as the only static variables that a filter is allowed to use are ones with primitive types and these can not cause dangling pointer errors. The question is of course how restrictive is this rule? While, for newly written code, it may be straightforward, if a little awkward, to replace static variables with context objects threaded through constructors, the same can not be said for library classes. It would be difficult to refactor them and if one did, they would loose backwards compatibility. We should thus strive to be as permissive as possible to increase opportunities for code reuse. The key observation here is that errors can only occur if it is possible to store an object allocated within a filter in a static field or in a field of an object reachable from a static field. This observation motivates extending the type system with the notion of *reference-immutable* types. These are types that are transitively immutable in their reference fields.

$\mathcal{D}6$: *A class* C *is reference-immutable if all non-primitive fields in the class and parent classes are declared final and are of reference-immutable types.*

$\mathcal{D}7$: *A type* T *is reference-immutable if it is primitive, an array of reference-immutable types, or a reference-immutable class.* □

The analysis infers which types must be immutable based on the use of static variables.

$\mathcal{R}3$: *Let* C *be a class in RCS, an expression reading a* static *field of reference type* T *is valid only if the field is declared* final *and* T *is reference-immutable.* □

4.4 Encapsulating Arrays

The rules as stated until now allow programs to use primitive arrays if they are static final fields as they are then reference-immutable. Furthermore, any kind of array can be safely allocated in transient memory. But it is not possible to allocate an array in stable memory or use an array within a capsule. We propose an extension to the type system that is just large enough to allow some common stream processing coding patterns.

$\mathcal{R}4$: *An instance field of a uni-dimensional array type is allowed in a stable class if it is declared* private final *and is assigned to a freshly allocated array in all constructors.* □

This ensures that array fields of stable objects cannot reference either transient objects nor borrowed objects.

4.5 Capsules

A capsule is an object that is manipulated in a linear fashion. At any given time the type system enforce that both of the following holds: (1) there is at most a single reference to the capsule from data channels, and (2) there are no references to a capsule from stable memory. These invariants permit zero-copy common uses of capsules.

$\mathcal{R}5$: *The type of field of a subclass of* Capsule *may be either primitive or an array of primitive.* □

The above rule ensures that capsules are reference-immutable, while the next rule ensures that capsules can only be instantiated by the STREAMFLEX runtime.

$\mathcal{R}6$: *A subclass of* Capsule *must have only a single constructor. It must be* private *and without parameters.* □

The motivation for $\mathcal{R}6$ is that STREAMFLEX must manage all allocation and reclamation of capsules. Otherwise, it would be possible to allocate a capsule in transient memory and push a transient object an output channel, eventually leading to dangling pointer error.

$\mathcal{R}7$: *Subclasses of Capsule cannot be stable classes.* □

From the point of view of stable and transient classes, a capsule is "just" like any other transient class. Thus, we inherit the guarantee that when work() returns there will be no reference to the capsule in the state of a filter.

5. Intrusion Detection System Example

To evaluate the power and applicability of STREAMFLEX on real-world applications, we have implemented a real-time Intrusion Detection System (IDS), inspired by [28], which analyzes a stream of raw network packets and detects intrusions by pattern matching. Figure 12 shows the declaration of the filter graph class Intrusion which instantiates and connects the six filters that implement the intrusion detection system. Figure 14 gives a graphical representation of the filter graph.

The capsules being passed around the system represent different network packets: Ethernet, IP, TCP and UDP. Object-oriented techniques are useful in the implementation as we model nested structure of protocol headers by inheritance. For instance, the IP capsule (IP_Hdr) is a subclass of the Ethernet capsule (Ether_Hdr) with extra fields to store IP protocol information.

Figure 15 shows PacketReader. This filter creates capsules representing network packets from a raw stream of bytes. For our experiments we simulate the network with the Synthesizer class (see start() in Figure 12). The synthesizer runs as a plain Java thread, and feeds the reader with a raw stream of bytes to be analyzed. Communication between the synthesizer and the PacketReader is done by calling PacketReader.write(). This method takes a reference to a buffer of data allocated in plain Java and parses it to create packets. write() is annotated @atomic to ensure that a filter can safely preempt the synthesizer at any time.

The PacketReader buffers data in its stable memory with the Buffer class. Buffer implements the Stable interface and contains an array of bytes. To satisfy the type system, this array had to be declared final and is freshly allocated in the constructor.

The reader uses the readPacket() to initialize capsules from the data stored in the buffer. startRead(), commitRead(), and abortRead() are used to ensure that only whole packets are read from the buffer. They do not need synchronization since (i) potential higher priority filters have no way to access the buffer (thanks to the isolation), and

```
public class Intrusion extends StreamFlexGraph {
  private Clock clock;
  private PacketReader read;
  private Filter trust, vsip, tear, join, dump;

  public Intrusion(int period) {
    clock = makeClock(period);
    read = (PacketReader)
                    makeFilter(PacketReader.class);
    trust = makeFilter(TrustFilter.class);
    vsip = makeFilter(VSIPFragments.class);
    tear = makeFilter(TearDrop.class);
    join = makeFilter(Joiner.class);
    dump = makeFilter(PacketDumper.class);
    connect(clock, read);
    connect(read, trust, 10);
    connect(trust, vsip, 10);
    connect(trust, "ok", join, 10);
    ...
    validate();
  }
  public void start() {
    new Synthetizer(read).start();
    super.start();
  }
}
```

Figure 12. StreamFlex graph of the Intrusion Detection System Example.

```
public class TearDrop extends Filter {
  private Channel<Ether_Hdr> in, out, fail;
  private TearMatcher pm = new TearMatcher();

  public void work() {
    Ether_Hdr p = in.take();
    if (p instanceof TCP_Hdr) {
      TCP_Hdr t = (TCP_Hdr) p;
      if (pm.step(t)) {
        p.filtered = true;
        p.filtered_by_TearDrop = true;
        fail.put(p);
        return;
      }
    }
    out.put(p);
  }
}
```

Figure 13. TearDrop, a filter of IDS.

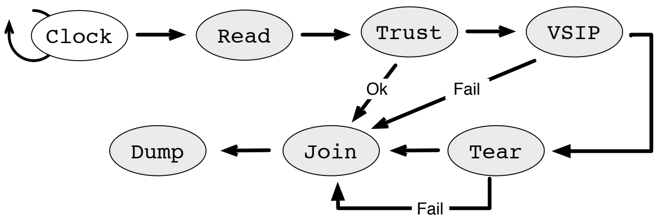


Figure 14. Filters connexion of Intrusion (Figure 12).

```
public class PacketReader extends Filter {
  private Channel<Ether_Hdr> out;
  private Buffer buffer = new Buffer(16384);
  public int underruns;

  public void work() {
    TCP_Hdr p;
    p = (TCP_Hdr) makeCapsule(TCP_Hdr.class);
    if (readPacket(p) < 0) underruns++;
    else out.put(p);
  }

  @atomic public void write(byte[] b) {
    buffer.write(b);
  }

  private int readPacket(TCP_Hdr p) {
    try {
      buffer.startRead();
      for (int i=0; i<Ether_Hdr.ETH_LEN; i++)
        p.e_dst[i] = buffer.read_8();
      ...
      return buffer.commitRead();
    } catch (UnderrunEx e) { buffer.abortRead(); ... }
  }
}

public class Buffer implements Stable {
  private final byte[] data;
  private int pos, lastpos;

  public Buffer(int cap) { data = new byte[cap]; }

  public int write(byte[] b) { ... }

  public void startRead() { lastpos = pos; }
  public int commitRead() { return pos−lastpos; }
  public void abortRead() { pos = lastpos; }

  public int read_32 throws UnderrunEx { ... }
  public short read_16 throws UnderrunEx { ... }
  public byte read_8 throws UnderrunEx { ... }
}
```

Figure 15. PacketReader and Buffer implementation.

(ii) plain Java threads, that can access the buffer trough write(), cannot preempt the filter execution.[4]

The packets first go to the TrustFilter which looks for packets that match a trusted pattern, these will not require further analysis. Other packets are forwarded to VSIPFragment. This filter detects IP fragments that are smaller than TCP headers. These are dangerous as they can be used to bypass packet-filtering firewalls. The TearDrop filter of Figure 13 recognizes attacks that involves IP packets that overlap.

The three filters, TrustFilter, VSIPFragment, and TearDrop, have a similar structure: an input channel (in) for incoming packets to analyse and two output channels, one for packets caught by the filters (ok or fail), the other one for uncaught packets (out). These filters also mark caught packets with meta-data that can be used in further treatment, logging or statistics. The TearDrop filter implementations rely on an automaton (TearMatcher in Figure 13) stored in stable space to recognize patterns on packet sequences that correspond to attacks.

A special built-in filter, Joiner, is used to transform a stream of data from multiple input filters to a single stream of data. The last Filter, PacketDumper, gather statistics of the whole intrusion detection process thanks to the meta-data written on packed by the previous filters.

6. Implementation

We have implemented StreamFlex on top of Ovm, a freely available Java virtual machine with an optimizing ahead-of-time compiler and support for real-time computing on uniprocessor embedded devices.

The Ovm virtual machine comes with a priority-preemptive scheduler. The complete priority range is from 1-42, where the subrange 12-39 represents real-time priorities and the remaining are used for Java threads. The Clock class is implemented as a thread with a real-time priority. The thread is started as a result of an invocation of StreamFlexGraph.start(). This causes all filter threads to be scheduled at a start time that may be the current time, or a user defined future time.

6.1 Memory Regions

For each filter, the underlying implementation allocates a fixed size continuous memory region for stable storage and another region for its transient data. The size of each of the above is set programmatically in the API. A filter and all of its implementation specific data structures are allocated in the stable area. These regions have the key property that they are not garbage collected. In Ovm, each thread has a default allocation area. The VM exposes low-level functionality for setting allocation areas. The method setCurrentArea() allows the implementation to change the allocation area for the current thread. Regions are reference counted, each call to setCurrentArea() increase the count of active threads by one. reclaimArea() decrease the counter by one for that area, if the counter is zero all objects in the area are reclaimed. reclaimAreaAndWait() is a blocking version of the above. Essentially, they reset the allocation pointer to the start of the area.

The VM statically identifies stable classes, and whenever an instance of a stable class is created by a thread running in a filter, the stable region is used instead of the transient region to allocate the object. The allocation of arrays encapsulated within the constructor of a stable class is rewritten to add code that checks if the thread is running within a filter and, if yes, allocates the array in stable memory. The virtual machine also supports allocation policies for meta-data. In particular, we rely on a policy for lock inflation that ensures that a lock is always allocated in the same area as the object with which it is associated, regardless of the current allocation area.

Capsules are managed by the implementation. The only way for a capsule to become garbage is if it is created or removed from an input channel and not put back on an output channel before the end of the filter's work() method. We thus keep track of all capsules created and used during an invocation of work() and reclaim those that are not published on output channels. Capsules are managed internally with object pools allocated in dedicated regions.

When an exception is thrown within a filter, the object is created with normal Java semantics. By default the exception object and its stack trace are created in transient memory. If the exception propagates out of the work() method, the stack trace is printed and the StreamFlex computation is terminated.

6.2 Atomicity

The implementation avoids blocking synchronization by supplementing Java monitors with a simple transactional facility built on top of the preemptible atomic regions of [24]. We implement channels using the @atomic annotation for methods that would otherwise be synchronized. The semantics of @atomic is simple: the method will execute atomically, unless another higher-priority thread preempts the current thread in which case the method is aborted. Since threads are scheduled with a priority preemptive scheduler, we know that a thread can only be preempted by a higher priority thread. If an atomic method is aborted, all changes performed within the atomic method are undone and the method will automatically be re-executed when the higher priority thread yields. For a schedulable task set, it is possible to prove the absence of livelocks [24]. For each write within an atomic the VM records the original value and address of field in a log. An abort boils down to replaying the log in reverse order. Enters and commits are constant time.

[4] We assume that filter run at higher priorities than plain Java threads as well as a priority-preemptive scheduling policy.

6.3 Borrowing

Borrowed arguments should not move or cause a garbage collection. A general way to do this would be to identify all borrowed objects, inflate their locks, and pin the objects to ensure that the garbage collector does not try to move them. As our prototype runs on uniprocessor VMs, careful assignment of priorities together with the use of @atomic methods ensures that a filter can never observe an inconsistent borrowed object.

6.4 Type Checking

The STREAMFLEX type checker is implemented as a pluggable type system. The checker is approximately 300 lines of code integrated as an extra pass in the javac 1.5 compiler. The type system defines a strict subset of the Java language [20] without requiring any changes to Java syntax. This approach is convenient as the rules are fairly compact and that error messages are returned by the Java compiler—no extra tool is required and message are returned with line numbers in a format that is familiar to programmers.

6.5 Static analysis

The use of reflection and native methods in STREAMFLEX code is limited to small set of operations. This together with the partially closed-world assumption (see Section 4.1) enforced by the type system permits the compiler to perform aggressive devirtualization and inlining.

7. Evaluation

We conducted a number of experiments to evaluate to which extent STREAMFLEX can be used to achieve high-throughput while remaining predictable, both important properties for streaming applications. We used the Intrusion Detection System of Section 5 as a larger, more realistic, benchmark.

We evaluated STREAMFLEX on two metrics: throughput and precision of inter-arrival time for periodically triggered STREAMFLEX filters. For the performance results, we considered two benchmark stream applications, and compared them head-to-head with baseline numbers from similar tests we conducted using plain Java. The baseline numbers are made up of test executions in Java using our virtual machine infrastructure as well as a standard Java platform as reference.

7.1 Base Performance

To evaluate the performance of STREAMFLEX, we performed various measurements of our implementation on the Ovm Java virtual machine. We considered here two benchmark applications developed at MIT for the StreamIt project, which we modified to make use of the STREAMFLEX API. The benchmark applications used were (1) a beam-form calculation on a set of inputs, and (2) a filter bank for multirate signal processing.[5] Figure 16 shows a graphical representation of the STREAMFLEX implementation of the BeamFormer benchmark. It shows the structure and number of filters as well as their interconnections.

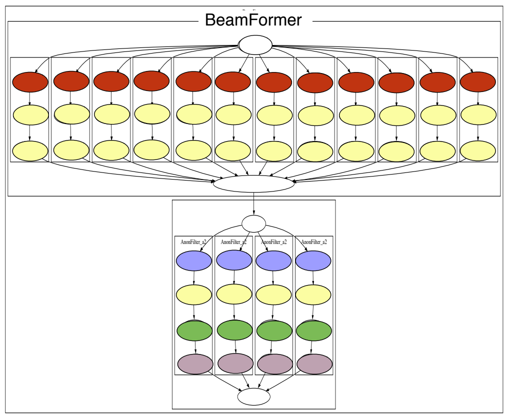


Figure 16. STREAMFLEX graph for the BeamFormer benchmark.

Both benchmark applications were configured to execute in a uniprocessor, single-threaded mode, and thus did not take advantage of the parallelization possibilities of the stream programming paradigm. All performance experiments were performed on a 3.8Ghz Pentium 4, with 4GB of physical memory. The operating system used was Linux (vanilla kernel, version 2.6.15-27-server). For the Ovm virtual machine, we configured it with a heap size of 512MB.

For the sake of comparison, we performed baseline measurements on the automatically generated Java variants of the StreamIt benchmark applications. The Java variants were benchmarked both on the Ovm virtual machine as will as the Java HotSpot virtual machine, version 1.5.0_10-b03, in mixed mode. Reported values are for the third run of the benchmark.

	STREAMFLEX Ovm	Java Ovm	Java HotSpot
BeamFormer	314 ms	1285 ms	1282 ms
FilterBank	1260 ms	4350 ms	3213 ms

Figure 17. Performance measurements showing actual runtime in milliseconds of performing 10,000 iterations of the benchmark applications using respectively STREAMFLEX and the Java variants of the StreamIt code on the Ovm virtual machine and on the Java HotSpot virtual machine.

[5] A description as well as the actual code for both the utilized StreamIt benchmark applications, SerializedBeamFormer.str and FilterBankNew.str are available for download at cag.csail.mit.edu/streamit.

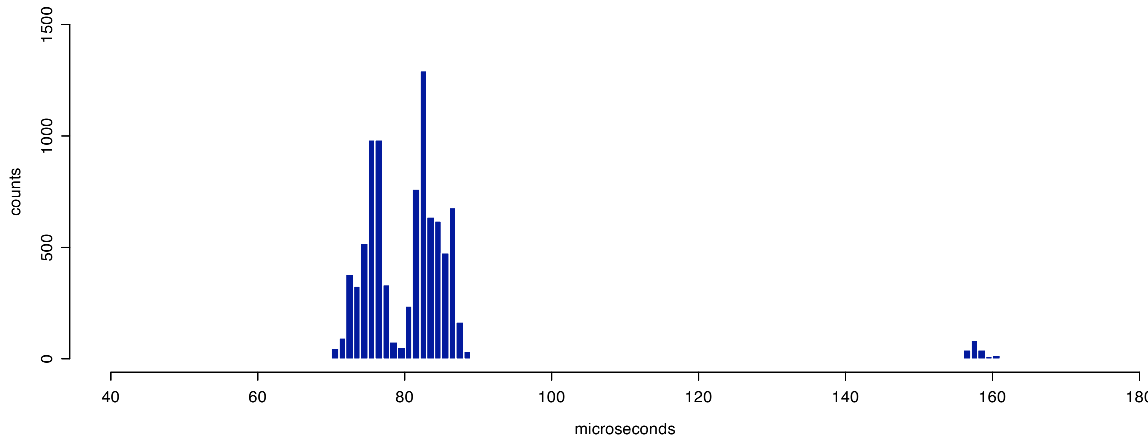


Figure 18. Frequencies of inter-arrival time (10,000 iterations) for a STREAMFLEX implementation of SerializedBeamFormer with periodic thread scheduled every 80 μs. The x-axes depict the inter-arrival time of two consecutive executions in microseconds of the periodic task whereas the y-axis depicts the frequency.

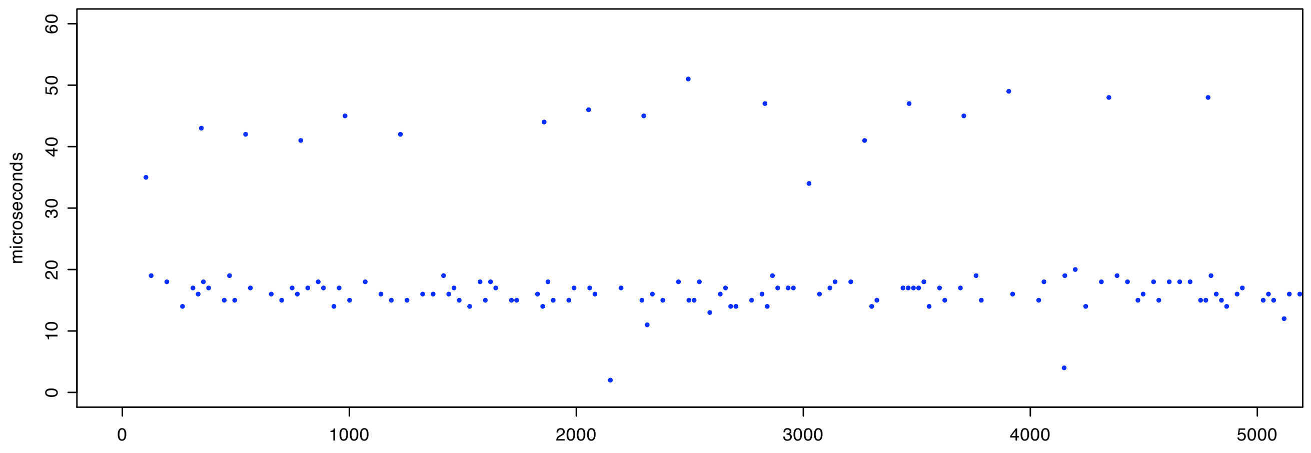


Figure 19. Missed deadlines over time (10,000 iterations; 5,000 depicted) for a STREAMFLEX implementation of the SerializedBeamFormer benchmark. The x-axes depict iterations of the filter whereas the y-axis shows the deadline misses in μs.

As depicted in Figure 17, STREAMFLEX performs significantly better than the Java variant executed on Ovm. Specifically, the performance improvement amounts to a factor 3.5 to 4. It is interesting to compare Ovm and HotSpot. Looking at the results for the Java code, we see that HotSpot is somewhat faster (25%) than Ovm for FilterBank. The slowdown can be in part explained by the fact that HotSpot is a more mature infrastructure and also because of known inefficiency in Ovm's treatment of floating point operations. It is interesting to observe that STREAMFLEX is a factor 2.5-4 times faster than the Java code running on HotSpot. This underlines that the performance gains are not caused by the virtual machine itself.

7.2 Predictability

To evaluate predictability, we measured the inter-arrival time and the number of deadline misses for a STREAMFLEX filter triggered periodically. A missed deadline occurs for the i'th firing of a filter with a period p if the actual completion time, α_i, comes after its expected completion time, ϵ_i, where $\epsilon_i = p(\lceil(\alpha_{i-1}/p)\rceil + 1)$.

We considered the SerializedBeamFormer benchmark application mentioned above, which we modified by scheduling the entry filter, a void splitter filter, with a period of 80 μs instead of being executed continuously. Experiments were performed on an AMD Athlon 64 X2 Dual Core processor 4400+ with 2GB of physical memory. The operating system used was Linux (kernel version 2.6.17-hrt-dyntick5), extended with high resolution timer (HRT) patches [1] configured with a tick period of 1 μs. We built Ovm with support for POSIX high resolution timers, and configured it with an interrupt rate of 1 μs. The time-critical STREAMFLEX filters were all scheduled to run at a 80 μs period and were executed over 10,000 periods.

As depicted in Figure 18, nearly all interesting observations of the inter-arrival time are centered around the 80 μs period with only a few microseconds of jitter. This is as it should be considering that the average iteration time of the benchmark is to be around 50 μs, leaving sufficient time for the underlying virtual machine to prepare and schedule the next period. In addition to the expected peak at 80 μs, there is a number of outliers around 160 μs. We attribute these perturbations to coincidental measurement noise, probably caused by buffering or flushing in the underlying operating system.

Figure 19 depicts missed deadlines over time for the STREAMFLEX benchmark application. Specifically, out of 10,000 periodic executions, we observed 223 missed deadlines, corresponding to a miss-rate of 2%. The missed deadlines are primarily centered around a range between 15-20 μs throughout the iterations. Most likely, these missed deadlines are a consequence of a slight jitter in the inter-arrival time, as depicted in Figure 18. Additionally, Figure 19 conveys a few observations randomly scattered around 30-50 μs. These deadline misses are directly linked with the outlier observations of inter-arrival time around 160 μs in that, generally speaking, a deadline miss between two consecutive periodic executions can cause for the inter-arrival time of the two to be larger than twice the actual period, as depicted in Figure 20.

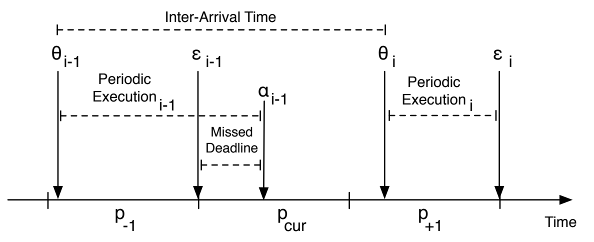


Figure 20. Timeline showing how a missed deadline can cause an inter-arrival time between two consecutive periodic executions to be larger than twice the period.

Figure 20 shows that in the event of a deadline miss (when actual completion time, α_{i-1}, lies after the expected completion time, ϵ_{i-1}) of a firing, $i-1$, the expected completion time, ϵ_i, of the subsequent firing, i, is set to be the end of the first-coming complete period, i.e., any time remaining in the current period is skipped. If the start of the subsequent periodic execution, i, is delayed (reflected in the actual start time, θ_i, lying after the period start) it can cause the inter-arrival time between the two consecutive periodic executions, $i-1$ and i, to be larger than twice the period p.

7.3 Intrusion Detection System

We performed various measurements of the Intrusion Detection System, Section 5, on the Ovm virtual machine. The PacketReader creates capsules at a rate of 12.5kHz (a period of 80μs). At this rate, the filter is able to generate packets in to the attack detection pipeline without experiencing any underruns from the simulator. In other words, at a rate of 12.5KHz the simulator can provide packets at the rate which matches the rate with which the IDS can analyze them. The time used to analyze a single network packet (from the capsule creation to the end of the Dumper filter) varies from 4μs to 10μs with an average of 5μs. One reason for this variation is that some packets are identified as a possible suspects by one of the filters, and thus require additional processing in the automata. If we consider raw bytes on a period of 0μs (no idle time), the intrusion detection system implemented using the STREAMFLEX API delivers an analysis rate of 750Mib/s.

7.4 Event Correlation

To evaluate the performance of a STREAMFLEX application executed on Ovm with reflex support compared to a plain Java variant executed on Ovm without reflex support. We implemented a transaction tracking scenario in which a filter graph is set up to analyze a real-time stream containing a constant flow of three different event types. Within this event flow, the filter graph searches for and puts together transaction tuples consisting of one of each of the three different event types; all sharing the same transaction number. The plain Java version only differs from the STREAMFLEX version by replacing realtime threads by plain Java thread and not exploiting memory area management, but instead allocating all objects on the heap.

The filter graph is composed of three filters: EventCreator $\rightarrow$EventMatcher $\rightarrow$EventSummarizer, where the former randomly generates a real-time stream of the three event types, the subsequent filter analyzes the stream for matching event types, and the final filter maintains real-time statistics of number of found transactions, the latency between the time the individual event times were found etc.

Figure 21 depicts the inter-arrival time between consecutive executions when executing the application variants scheduled with a period of 200 microseconds. Both for STREAMFLEX and the plain Java variant, Ovm can achieve a 200 microsecond period. However, in the plain Java variant, huge deadline misses are observed (2 peaks of 67 milliseconds) due to the garbage collection. No deadline misses are observed with the STREAMFLEX application.

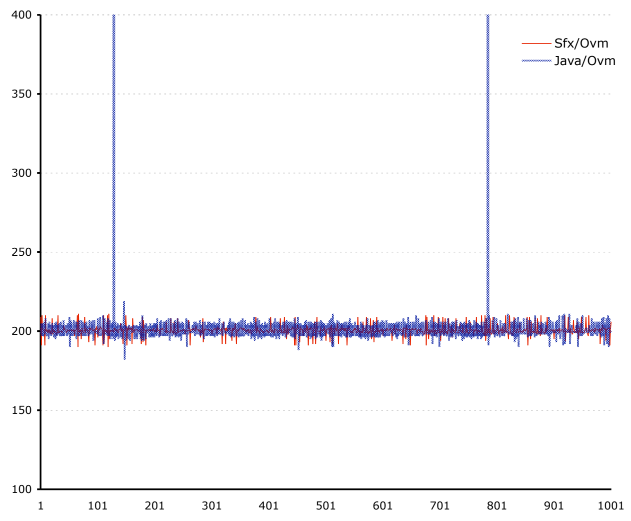


Figure 21. Inter-arrival time over time for a StreamFlex and a plain Java variant of a transaction tracking scenario scheduled with a frequency of 5,000 Hz. The x-axis shows the periodic executions (1,000 shown) and the y-axis shows the logarithm of the inter-arrival time (in μs).

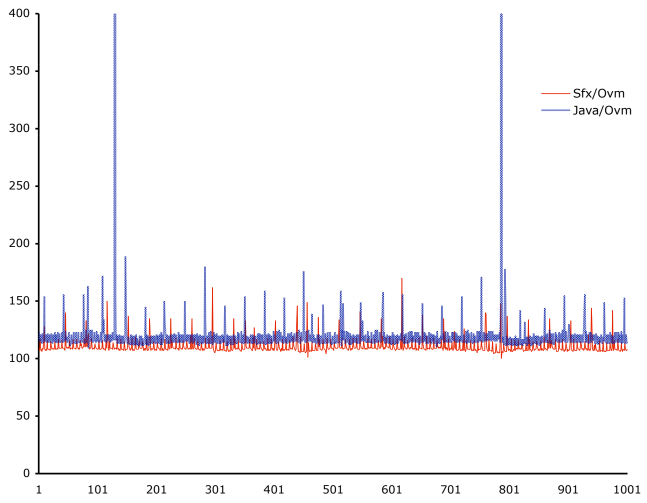


Figure 22. Processing time over time for a StreamFlex and a plain Java variant of a transaction tracking scenario (non periodic). The x-axis shows the periodic executions (only 1,000 shown) and the y-axis shows the logarithm of the processing time (in μs).

Figure 22 depicts the processing time when executing the application variants with filters that fire continuously. As can be seen, the plain Java variant suffer regular delays that correspond to GC activations that cost around 67 milliseconds each. As expected there is no GC activations for StreamFlex. Note that, even if we ignore activations that involve GC, the StreamFlex version is still faster than the plain Java one.

8. Related Work

There are many languages and systems supporting stream processing. The following stand out among them, Borealis [2], Infopipes [10] and StreamIt [32]. These languages have a history that can be traced back to Wadge and Ashcroft's Lucid [5] data flow language and the Esterel family of synchronous languages [13, 21]. Infopipes [10] come as an extension to a variant of Smalltalk (Squeak) and has very rich set of operators. StreamIt [32], although it started as a subset of Java, now comes with its own language and compiler infrastructure that generate both Java and native code and has a number of restrictions to ensure efficient compilation to native code.

StreamFlex resembles these projects in that it introduces a set of abstractions, such as filters, pipes/channels, splitters, and joiners designed for programming stream-based applications. Using the Java programming language for stream processing, and especially when aiming for high-throughput is not obviously a good idea. Java is a general purpose language whereas the above mentioned languages enjoy implementations and compilers specially tuned for efficient execution of streaming applications. A Java virtual machine introduces overheads due to, e.g., garbage collection and array bound checks, and must support dynamic loading—a major drawback for compiler optimizations. The benefits of using Java are significant as it has: a large community of programmers, high-quality IDEs such as Eclipse, and numerous libraries.

StreamFlex and Infopipes support periodic scheduling of filters. Infopipes, to the best of our knowledge, have to deal with garbage collection by the underlying runtime system. Hence, one must be very careful to ensure to limit allocation in order not to which might hamper responsiveness and thus predictability. In contrast, StreamFlex relies on Reflexes to provide high responsiveness and, as demonstrated earlier, is easily able to operate at periods of 80 μs.

High Responsiveness. Achieving sub-millisecond response time in Java has been the topic for numerous research papers. The Achilles heel of Java is its reliance on garbage collection. In order reach such response time one must circumvent the abrupt interference from the garbage collector which for a standard Java virtual machine means freezing of threads up to 100 milliseconds. We conducted a comparative study of the Real-time Java Specification (RTSJ) [11] region-based memory management API and a state-of-the art real-time garbage collection algorithm [7]. Our conclusion [27] is that real-time collectors are not suited for sub-millisecond response times and that RTSJ scoped memory is too error-prone for widespread adoption.

StreamFlex relies on a simplified version of the RTSJ region API to ensure that sub-millisecond deadlines can be met. We depart from the RTSJ by our use of static type system to ensure memory safety. This has the major advantage of avoiding the brittleness of RTSJ applications and also brings performance benefits as we do not have to implement run-time checks to prevent dangling pointers. STREAMFLEX is built on top of Ovm and a simple real-time programming model [30] which provides real-time threads, region-based allocation and an extended type system. STREAMFLEX extends that model with stream programming constructs and adapts the type system to particular needs of stream processing.

Related approaches include Eventrons [29] and Exotasks [6]. Eventrons[6] are closely related to Reflexes in that they provide very low latency real-time processing, with periods of down to 50 μs. Unlike the approach presented in this paper, Eventrons use a run-time data-sensitive program analysis to verify the logic of the real-time part of an application. This has the advantage of being more precise, at the cost of a heavier run-time and delayed error reporting. Exotasks are closer to STREAMFLEX as they allow allocation and can be arranged in a graph of communicating real-time processors. One of the main difference is that Exotasks use real-time GC. For each filter in an exotask graph there is one real-time collector. This means that Exotasks do not need to differentiate between stable and transient data, but this comes at the price of a higher latency.

Ownership types. Ownership types were first proposed by Noble, Potter and Vitek in [25] as a way to control aliasing in object-oriented systems. Most ownership type system require fairly extensive changes to the code of applications to add all the annotations needed by the type checker. The STREAMFLEX type system is an extension of the implicit ownership type system of [34] which is the latest in a line of research that emphasized lightweight type systems for region-based memory [3, 35]. STREAMFLEX ownership is *implicit* because, unlike e.g. [14, 12], no ownership parameters are needed. Instead, ownership is defaulted using straightforward rules. Most other ownership type systems require each class to be equipped with one ore more owner parameter. Much like Java generics, these parameters are expected to be erased at compile time. This approach has however an important drawback: it requires a complete refactoring of all library classes and does not interact well with raw types. While an implicit ownership type system is less expressive, the cost in complexity and the disruption to legacy code arguably outweighs the benefits of the added expressive power [34].

Real-time Event Channels. Previous work on event channels, in particular the Facet [23] event channel, is related to our work. Facet is an aspect-oriented CORBA event channel written in Java with the RTSJ API. Facet is highly configurable and provide different event models. However, it shares the drawbacks given above for the RTSJ. In the RTSJ it is very difficult to implement a zero-copy message safely. The Zen real-time CORBA platform [22], written with the RTSJ, is another platform on which one could conceivably implement a stream processor. Unfortunately, its implementation still suffers some performance problems. In our experiments with Zen, we have not been able to achieve sub-millisecond message round-trip times.

Zero-Copy Message Passing. The Singularity operating system supports a notion of channels with messages allocated in a region of restricted inter-process shared memory [18]. The use of language techniques to avoid copying is similar to our approach for capsules. Singularity messages are owned by a single process and are transferred in a linear fashion. Ennals *et al.* presented a linear type system for programming network processors which ensured that every packet is owned by a single thread at a time [15].

Logical Execution Time. Programming language based on the logical execution time assumption such as Giotto [19] have garnered much interest in the real-time community lately. Using LET, the programmer specifies with every task invocation the logical execution time of the task, that is, the time at which the task provides its outputs. If the outputs are ready early, then they are made visible only when the specified logical execution time expires. This buffering of outputs achieves determinacy in both timing and functionality. We believe STREAMFLEX could be a good platform to investigate LET in the context of Java. Our filters are already deterministic (due to the isolation invariant), what seems to be missing is the scheduling and deadline monitoring component.

Exotasks [6] use a scheduling policy based on LET to ensure time portability of real-time programs. Considering the similarities between the two models, we believe that it would be possible to have time portable STREAMFLEX graphs. This makes for an interesting direction for future work.

9. Conclusion

We presented a programming model, STREAMFLEX, for high-throughput stream processing in Java. On the one hand, STREAMFLEX extends the Java virtual machine with transactional channels and type-safe region-based allocation. On the other hand, STREAMFLEX restricts Java in that it provides a stricter typing discipline on the stream components of the code. STREAMFLEX relies on the notion of priority-preemptive threads that can safely preempt all other Java threads, including the garbage collector. By introducing a STREAMFLEX type system based on an implicit ownership, we showed that using a simple set of type constraints, we are able to provide a statically type-safe region-based memory model.

Our evaluation of STREAMFLEX is encouraging both in terms of performance and predictability. In fact, when comparing the benchmark applications using STREAMFLEX to

[6] Eventrons are available under the name XRTs in the IBM Websphere Real-time product.

equivalent implementations in Java, STREAMFLEX ran up to 4 times faster than the Java version. As for predictability, our evaluation indicated that we can achieve 80 μs response times with only 2% of the executions failing to meet their deadlines.

In this work we have only looked at static filter graphs. In future work we intend to investigate more dynamic communication mechanisms such as type-based publish/subscribe systems [16]. We will also look at alternative memory management models such as the hierarchical real-time garbage collection technique of [26].

Acknowledgments. We thank Filip Pizlo, Jason Baker and the member of the Purdue Ovm team for their help with Ovm internals. We thank Rodric Rabbah and Joshua Auerbach for helpful comments on this text. Finally, our work greatly benefited from the availability of the StreamIt benchmarks. This work is supported in part by NSF grants 501 1398-1086 and 501 1398-1600, an IBM Faculty Award, as well as the EU 6th Framework Programme, project IST-002057.

References

[1] http://www.tglx.de/projects/hrtimers/2.6.17/.

[2] 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 *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[3] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 124–147, Nantes, France, July 2006.

[4] A. Armbuster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 2006.

[5] E. A. Ashcroft and W. W. Wadge. Lucid, a non-procedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[6] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with exotasks. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*, pages 51–62, 2007.

[7] D. F. Bacon, P. Chang, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 285–298, New Orleans, Louisiana, Jan. 2003.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324–341, 1996.

[9] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. In *Proceedings of the Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.

[10] A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Syst.*, 8(5):406–419, 2002.

[11] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[12] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of Conference on Programming Languages Design and Implementation (PLDI)*. ACM Press, 2003.

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 178–188, Munich, West Germany, Jan. 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, Oct. 1998.

[15] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In *Proceedings of the 13th European Symposium on Programming (ESOP)*, pages 204–218. Springer, 2004.

[16] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.

[17] F. Boussinot and R. De Simone. The ESTEREL language. *Proc. IEEE*, 79(9):1293–1304, Sept. 1991.

[18] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys2006*, Leuven, Belgium, Apr. 2006. ACM SIGOPS.

[19] A. Ghosal, T. Henzinger, C. Kirsch, and M. Sanvido. Event-driven programming with logical execution times. In *Proceedings of the 7th International Workshop, Hybrid Systems Computation and Control*, March 2004.

[20] J. Gosling, B. Joy, G. Steele, Jr., and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[21] P. L. Guernic, M. L. Borgne, T. Gauthier, and C. L. Maire. Programming real time applications with signal. *Proceedings of the IEEE*, September 1991.

[22] A. Krishna, D. Schmidt, and R. Klefstad. Enhancing Real-Time CORBA via Real-Time Java Features. In *24th International Conference on Distributed Computing Systems*

(ICDCS 2004), pages 66–73, Hachioji, Tokyo, Japan, March 2004.

[23] R. P. M, R. K. Cytron, D. Sharp, and E. Pla. Transport layer abstraction in event channels for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 144–152, 2003.

[24] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2005.

[25] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *12th European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[26] F. Pizlo, A. Hosking, and J. Vitek. Hiearchical real-time garbage collection. In *ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 123–133, 2007.

[27] F. Pizlo and J. Vitek. An empirical evalutation of memory management alternatives for Real-time Java. In *27th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2006.

[28] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, pages 8–17, 1999.

[29] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. In *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 283–294, 2006.

[30] J. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. Reflexes: Abstractions for highly responsive systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, 2007.

[31] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction (CC'02)*, Apr. 2002.

[33] O. Vitek, B. Craig, C. Bailey-Kellog, and J. Vitek. Inferential backbone assignment for sparse data. *Journal of Biomolecular NMR*, 2006.

[34] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. ScopeJ: Simple ownership types for memory management. Submitted for publication, Dec. 2006.

[35] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Lisbon, Portugal, Dec. 2004.