StreaMIT: A Language for Streaming Applications*

Bill Thies, Michal Karczmarek, and Saman Amarasinghe

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

{thies, karczma, saman}@lcs.mit.edu

August 13, 2001

ABSTRACT
We characterize high-performance streaming applications as a new and distinct domain of programs that is becoming increasingly important. The StreaMIT language provides novel high-level representations to improve programmer productivity and program robustness within the streaming domain. At the same time, the StreaMIT compiler aims to improve the performance of streaming applications via stream-specific analyses and optimizations. In this paper, we motivate, describe and justify the language features of StreaMIT, which include: a structured model of streams, a messaging system for control, a re-initialization mechanism, and a natural textual syntax. We also present a means of reasoning about time in terms of "information flow": a concept that we believe is fundamental to the streaming domain. Using this concept, we give a formal semantics for StreaMIT's messaging system, as well as a simple algorithm for detecting deadlock and buffer overlow.

1. INTRODUCTION
Applications that are structured around some notion of a "stream" are becoming increasingly important and widespread. There is evidence that streaming media applications are already consuming most of the cycles on consumer machines [8], and their use is continuing to grow. In the embedded domain, applications for hand-held computers, cell phones, and DSP's are centered around a stream of voice or video data. The stream abstraction is also fundamental to high-performance applications such as intelligent software routers, cell phone base stations, and HDTV editing consoles.

Despite the prevalence of these applications, there is surprisingly little language and compiler support for practical, large-scale stream programming. Of course, the notion of a stream as a programming abstraction has been around for decades [2], and a number of special-purpose stream languages have been designed (see [9] for a review). Many of these languages and representations are elegant and theoretically sound, but they often lack features and are too inflexible to support straightforward development of modern stream applications, or their implementations are too inefficient to use in practice.

*This document is **MIT/LCS Technical Memo LCS-TM-620**, August 2001. A similar version was submitted to POPL 2002. Please do not distribute.

Consequently, most programmers turn to general-purpose languages such as C or C++ to implement stream programs, resorting to low-level assembly codes for performance-critical loops. This practice is labor-intensive, error-prone, and very costly, since the performance-critical sections must be re-implemented for each target architecture. Moreover, general purpose languages do not provide a natural and intuitive representation of streams, thereby having a negative effect on readability, robustness, and programmer productivity.

StreaMIT is a language and compiler specifically designed for modern stream programming. Its goals are two-fold: first, to raise the abstraction level in stream programming, thereby improving programmer productivity and program robustness, and second, to provide a compiler that performs stream-specific optimizations to achieve the performance of an expert assembly programmer.

To address the first of these goals, this paper motivates, describes, and justifies StreaMIT's high-level language features. The version of StreaMIT described in this paper requires static flow rates in the streams; applications such as compression that have dynamic flow rates will be the subject of future work. This paper also presents a notion of "information flow" that we believe is fundamental to the streaming domain. Using this notion, we give a clear semantics to StreaMIT's messaging system and describe simple algorithms for static deadlock and overflow detection.

This paper is organized as follows. In Section 2, we characterize the domain of streaming programs that motivates the design of StreaMIT. In Section 3, we provide a detailed description of StreaMIT, and in Section 4 we formally define the semantics of StreaMIT using a notion of "information flow". We present a detailed example of a software radio in Section 5 and describe related work in Section 6. Section 7 contains our conclusions, and the Appendix gives an overview of StreaMIT syntax.

2. STREAMING APPLICATION DOMAIN
The applications that make use of a stream abstraction are diverse, with targets ranging from embedded devices, to consumer desktops, to high-performance severs. However, we have observed a number of properties that these programs have in common–enough so as to characterize them as be-

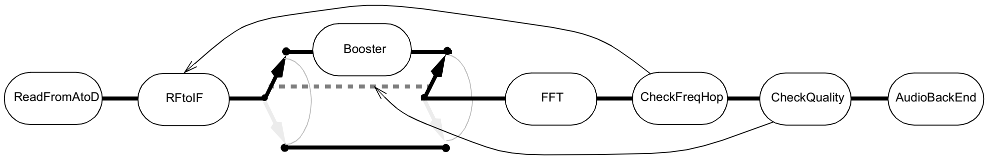


Figure 1: A block diagram of a software radio. A detailed StreaMIT implementation appears in Figure 17.

longing to a distinct class of programs, which we will refer to as *streaming applications*. The following are the salient properties of a streaming application, independent of its implementation:

1. *Large streams of data.* Perhaps the most fundamental aspect of a streaming application is that it operates on a large (or virtually infinite) sequence of data items, hereafter referred to as a *data stream*. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. This is in contrast to scientific applications, in which a fixed input set is manipulated with a large degree of data reuse.

2. *Independent stream filters.* Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. We will refer to the basic unit of this transformation as a *filter*: an operation that–on each execution step–reads one or more items from an input stream, performs some computation, and writes one or more items to an output stream. Filters are generally independent and self-contained, without references to global variables or other filters. A stream program is the composition of filters into a *stream graph*, in which the outputs of some filters are connected to the inputs of others.

3. *A stable computation pattern.* The structure of the stream graph is generally constant during the steady-state operation of a stream program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.

4. *Occasional modification of stream structure.* Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. For instance, if a wireless network interface is experiencing high noise on an input channel, it might react by adding some filters to clean up the signal; a software radio re-initializes a portion of the stream graph when a user switches from AM to FM. Sometimes, these re-initializations are synchronized with some data in the stream–for instance, when a network protocol changes from bluetooth to 802.11 at a certain point of a transmission. There is typically an enumerable number of configurations that the stream graph can adopt in any one program, such that all of the possible arrangements of filters are known at compile time.

5. *Occasional out-of-stream communication.* In addition to the high-volume data streams passing from one filter to another, filters also communicate small amounts of control information on an infrequent and irregular basis. Examples include changing the volume on a cell phone, printing an error message to a screen, or changing a coefficient in an upstream FIR filter.

6. *High performance expectations.* Often there are real-time constraints that must be satisfied by streaming applications; thus, efficiency (in terms of both latency and throughput) is of primary concern. Additionally, many embedded applications are intended for mobile environments where power consumption, memory requirements, and code size are also important.

```
class FIR extends Filter {
   Channel input = new FloatChannel();
   Channel output = new FloatChannel();
   int N;

   void init(int N) {
      this.N = N;
   }

   void work() {
      float sum = 0;
      for (int i=0; i<N; i++)
         sum += input.peek(i)*FIR_COEFF[i][N];
      input.pop();
      output.push(sum);
   }
}

class Main extends Stream {
   void init() {
      add(new DataSource());
      add(new FIR(N));
      add(new Display());
   }
}
```

Figure 2: An FIR filter in StreaMIT.

3. LANGUAGE OVERVIEW

StreaMIT includes stream-specific abstractions and representations that are designed to improve programmer productivity for the domain of programs described above. In this paper, we present StreaMIT in legal Java syntax for ease of presentation. Though this syntax can express the fundamental ideas of StreaMIT, in the longer term we plan to develop a cleaner and more abstract syntax that is designed specifically for stream programs.

Figure 17 contains a detailed example of a software radio implemented in StreaMIT; a block diagram of the system appears in Figure 1. In the following sections, we draw on different components of this example to describe and justify the major features of StreaMIT. A more detailed guide to the syntax of StreaMIT can be found in the Appendix.

3.1 Filters

3.1.1 StreaMIT Approach

The basic unit of computation in StreaMIT is the `Filter`. An example of a Filter is the `FIRFilter`, a component of our

```
int N = 5;
int BLOCK_SIZE = 100;

void step(float[] input, float[] output, int numIn, int numOut) {
   float sum = 0;
   for (int k=0; k<numIn; k++)
      sum = sum + input[k]*FIR_COEFF[k+numIn][N];
   for (int k=numIn; k<N; k++)
      sum = sum + input[k]*FIR_COEFF[k-numIn][N];
   output[numOut] = sum;
   input[numIn] = getData();
}

void main() {
   float input[] = new float[N];
   float output[] = new float[BLOCK_SIZE];
   int numIn, numOut;

   for (numIn=0; numIn<N; numIn++)
      input[numIn] = getData();

   while (true) {

      for (out=0; numIn<N; numIn++, numOut++)
         step(input, output, numIn, numOut);

      int wholeSteps = (BLOCK_SIZE-numOut)/N;
      for (int k=0; k<wholeSteps; k++) {
         for (numIn=0; numIn<N; numIn++, numOut++)
            step(input, output, numIn, numOut);

      for (numIn=0; numOut<BLOCK_SIZE; numIn++, numOut++)
         step(input, output, numIn, numOut);

      displayBlock(output);
   }
}
```

Figure 3: An optimized FIR filter in a procedural language. A complicated loop nest is required to avoid mod functions and to use memory efficiently, and the structure of the loops depends on the data rates (e.g., BLOCK_SIZE) within the stream. An actual implementation might inline the calls to `step`.

```
class FIRFilter {
   int N;
   float[] input;

   FIRFilter(int N) {
      this.N = N;
   }

   float[] getData(float[] output, int offset, int length) {
      if (input==null) {
         input = new float[MAX_LENGTH];
         source.getData(input, 0, N+length);
      } else {
         source.getData(input, N, length);
      }

      for (int i=0; i<length; i++) {
         float sum = 0;
         for (int j=0; j<N; j++) {
            sum = sum + data1[i+j]*FIR_COEFF[j][N];
         }
         output[i+offset] = sum;
      }

      for (int i=0; i<N; i++) {
         input[i] = input[i+length];
      }
   }
}

void main() {

   DataSource datasource = new DataSource();
   FIRFilter filter = new FIRFilter(5);
   Display display = new Display();
   filter.source = datasource;
   display.source = filter;
   display.run();

}
```

Figure 4: An FIR filter in an object oriented language. A "pull model" is used by each filter object to retrieve a chunk of data from its source, and straight-line code connects one filter to another.

software radio (see Figure 2). Each `Filter` contains an `init` function that is called at initialization time; in this case, the `FIRFilter` records `N`, the number of items it should filter at once. A user should instantiate a filter by using its constructor, and the `init` function will be called implicitly with the same arguments that were passed to the constructor[1].

The `work` function describes the most fine grained execution step of the filter in the steady state. Within the `work` function, the filter can communicate with neighboring blocks using the `input` and `output` channels, which are typed FIFO queues declared as fields at the top of the class. These high-volume channels support the three intuitive operations:

1. `pop()` removes an item from the end of the channel and returns its value.

2. `peek(i)` returns the value of the item i spaces from the end of the channel without removing it.

3. `push(x)` writes x to the front of the channel. The argument x is passed by value; if it is an object, a separate copy is enqueued on the channel.

[1]This design might seem unnatural, but it is necessary to allow inlining (Section 3.2) and re-initialization (Section 3.4) within a Java-based syntax.

A filter can also push, pop, and peek items from within the `init` function if it needs to set up some initial state on its channels, although this usually is not necessary.

A major restriction of the current version of StreaMIT is that it requires filters to have static input and output rates. That is, the number of items peeked, popped, and pushed by each filter must be constant from one invocation of the `work` function to the next. We plan to support dynamically changing rates in a future version of StreaMIT.

3.1.2 Rationale

StreaMIT's representation of a filter is an improvement over general-purpose languages. In a procedural language, the analog of a filter is a block of statements in a complicated loop nest (see Figure 3). This representation is unnatural for expressing the feedback and parallelism that is inherent in streaming systems. Also, there is no clear abstraction barrier between one filter and another, and high-volume stream processing is muddled with global variables and control flow. The loop nest must be re-arranged if the input or output ratios of a filter changes, and scheduling optimizations further inhibit the readability of the code. In contrast, StreaMIT places the filter in its own independent unit, making explicit the parallelism and inter-filter communication while hiding the grungy details of scheduling and optimization from the programmer.

```
class FFT extends Stream {
   void init(int N) {
      add(new SplitJoin() {
         void init() {
            setSplitter(WEIGHTED_ROUND_ROBIN(N/2, N/2));
            for (int i=0; i<2; i++)
               add(new SplitJoin() {
                  void init() {
                     setSplitter(ROUND_ROBIN);
                     add(IDENTITY());
                     add(IDENTITY());
                     setJoiner(WEIGHTED_ROUND_ROBIN(N/4, N/4);
               }});
            setJoiner(ROUND_ROBIN);
      }});
      for (int i=2; i<N; i*=2)
        add(new Butterfly(i, N));
}}
```

Figure 5: A Fast Fourier Transform (FFT) in StreaMIT.

One could also use an object-oriented language to implement a stream abstraction (see Figure 4). This avoids some of the problems associated with a procedural loop nest, but the programming model is again complicated by efficiency concerns. That is, a runtime library usually exectutes filters according to a pull model, where a filter operates on a block of data that it retrieves from the input channel. The block size is often optimized for the cache size of a given architecture, which hampers portability. Moreover, operating on large-grained blocks obscures the fundamental fine-grained algorithm that is visible in a StreaMIT filter. Thus, the absence of a runtime model in favor of automated scheduling and optimization again distinguishes StreaMIT.

3.2 Connecting Filters

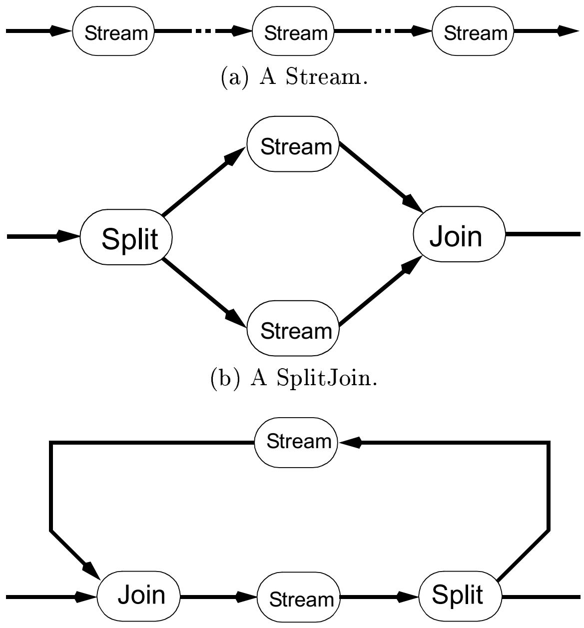
3.2.1 StreaMIT Approach

The basic construct for composing filters into a communicating network is a `Stream`. The FFT in Figure 5 is an example of a `Stream` that appears in our software radio. Like a `Filter`, a `Stream` has an `init` function that is called upon its instantiation. However, there is no `work` function, and all input and output channels are implicit; instead, the stream behaves as the sequential composition of filters that are specified with successive calls to `add` from within `init`. That is, `Stream` creates a single pipeline.

There are two other stream constructors besides `Stream`: `SplitJoin` and `FeedbackLoop`. The former is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. There are three kinds of splitters:

1. WEIGHTED_ROUND_ROBIN(i_1, i_2, ..., i_k), which sends the first i_1 data items to the first stream, the next i_2 data items to the second stream, and so on.
2. ROUND_ROBIN, which is just a weighted round robin where all weights are 1.
3. DUPLICATE, which replicates each data item and sends a copy to each parallel stream.
4. NULL, which means that all of the parallel components are sources and there is no input to split.

Similarly, there are three kinds of joiners:



(a) A Stream.

(b) A SplitJoin.

(c) A FeedbackLoop.

Figure 6: Stream structures supported by StreaMIT.

1. WEIGHTED_ROUND_ROBIN(i_1, i_2, ..., i_k), which reads the first i_1 data items from the first stream, the next i_2 data items to the second stream, and so on.
2. ROUND_ROBIN, which is just a weighted round robin where all weights are 1.
3. COMBINE, which reads from all the streams in parallel and combines the results into a structure. The structure has a separate field for each stream, and the joiner cannot fire until each field is filled by an item from the corresponding output channel.
4. NULL, which means that all of the parallel components are sinks and there is no output to join together.

The splitter and joiner type are specified with calls to `setSplitter` and `setJoiner`, respectively (see Figure 5); the parallel streams are specified by successive calls to `add`, with the i'th call setting the i'th stream in the SplitJoin . Note that a WEIGHTED_ROUND_R can function as an exclusive selector if one or more of the weights are zero. Also, there are additional splitters and joiners that we plan to add when StreaMIT supports filters with dynamically changing rates, including:

1. TYPE_DISPATCH, which sends an item to one of multiple streams depending on its type.
2. ANY, which sends items to any parallel stream that has space on its input, or reads items from any parallel stream that has output available.

The last control construct provides a way to create cycles in the stream graph: the `FeedbackLoop`. It contains a joiner,

```
class Fibonnacci extends FeedbackLoop {
   void init() {
      setDelay(2);
      setJoiner(WEIGHTED_ROUND_ROBIN(0,1));
      setBody(new Filter() {
          Channel input = new IntChannel();
          Channel output = new IntChannel();
          void work() {
              output.push(input.peek(0)+input.peek(1));
              input.pop();
          }
      });
      setSplitter(DUPLICATE);
   }

   int initPath(int index) {
      return index;
   }
}
```

Figure 7: A FeedbackLoop version of Fibonnacci.

a body stream, a splitter, and a loop stream, which are set with calls to `setJoiner`, `setBody`, `setSplitter`, and `setLoop`, respectively (see Figure 7). The splitters and joiners can be any of those for SplitJoin, except for NULL.

The feedback loop has a special semantics when the stream is first starting to run. Since there are no items on the feedback path at first, the stream instead inputs items from an `initPath` function defined by the `FeedbackLoop`; `initPath` is called with the number of the data item that is being fabricated (starting from 0). With a call to `setDelay` from within the `init` function, the user can specify how many items should be calculated with `initPath` before the joiner looks for data items from the feedback channel.

Evident in all of these examples is another feature of the StreaMIT syntax: *inlining*. The definition of any stream or filter can be inlined at the point of its instantiation, thereby preventing the definition of many small classes that are used only once, and, moreover, providing a syntax that reveals the hierarchical structure of the streams from the indentation level of the code. In our Java syntax, we make use of anonymous classes for inlining [5].

3.2.2 *Rationale*

StreaMIT differs from other languages in that it imposes a well-defined structure on the streams; all stream graphs are built out of a hierarchical composition of Streams, SplitJoins, and FeedbackLoops. This is in contrast to other environments, which generally regard a stream as a flat and arbitrary network of filters that are connected by channels. However, arbitrary graphs are very hard for the compiler to analyze, and equally difficult for a programmer to describe. Most programmers either resort to straight-line code that links one filter to another (thereby making it very hard to visualize the stream graph), or using an ad-hoc graphical programming environment that is awkward to use and admits no good textual representation.

In contrast, StreaMIT is a clean textual representation that–especially with inlined streams–makes it very easy to see the shape of the computation from the indentation level of the code. The comparison of StreaMIT's structure with arbitrary stream graphs could be likened to the difference between structured control flow and GOTO statements. Though sometimes the structure restricts the expressiveness of the

```
class TrunkedRadio extends Stream {
   RFtoIFPortal freqHop = new RFtoIFPortal();
   ...
   void init() {
      ...
      RFtoIF rf2if;
      add(rf2if = new RFtoIF(STARTFREQ));
      freqHop.register(rf2if);
      ...
      add(new CheckFreqHop(freqHop));
      ...
   }
}

class RFtoIF extends Filter {
   ...
   // sets frequency to <f>
   void setf(float f) {
      ...
   }
}

class CheckFreqHop extends SplitJoin {
   RFtoIFPortal freqHop;

   void init(RFtoIFPortal freqHop) {
      this.freqHop = freqHop;
      ...
      add(new Filter() {
         ...
         void work() {
            ...
            if(val >= MIN_THRESHOLD)
               freqHop.setf(Freq[k], new TimeInterval(4*N, 6*N));
            ...
         }
      });
      ...
   }
}
```

Figure 8: The frequency-hopping of our software radio illustrates StreaMIT's messaging system. In `TrunkedRadio`, a `Portal` is created to hold the message target, `rf2if`. Then, `CheckFreqHop` uses the `Portal` to send a frequency-change message.

programmer, the gains in robustness, readability, and compiler analysis are immense.

A final benefit of stream graph construction in StreaMIT is the ability to do *scripting* to parameterize graphs. For instance, both the FFT stream in Figure 5 inputs a parameter `N` and adjusts the number of butterfly stages appropriately. This further improves readability and decreases code size.

3.3 Messages

3.3.1 *StreaMIT Approach*

StreaMIT provides a dynamic messaging system for passing irregular, low-volume control information between filters and streams. Messages are sent from within the body of a filter's `work` function, perhaps to change a parameter in another filter. For example, in the `CheckFreqHop` stream of our software radio example (Figure 8), a message is sent upstream to change the frequency of the receiver if the downstream component detects that the transmitter is about to change frequencies. The sender can continue to execute while the message is en route, and the `set_freq` method will be invoked in the receiver with argument `Freq[k]` when the message arrives. Since message delivery is asynchronous, there can be no return value; only void methods can be message targets.

Message timing. The central aspect of the messaging system is a sophisticated timing mechanism that allows filters to specify when a message will be received relative to the flow of information between the sender and the receiver.

Recall that each filter executes independently, without any notion of global time. Thus, the only way for two filters to talk about a time that is meaningful for both of them is in terms of the data items that are passed through the streams from one to the other.

In StreaMIT, one can specify a range of latencies for a message to get delivered. This latency is measured in terms of an information "wavefront" from one filter to another. For example, in the `CheckFreqHop` example of Figure 8, the sender indicates an interval of latencies between $4N$ and $6N$. This means that the receiver will receive the message immediately following the last invocation of its own `work` function which produces an item affecting the some output of the *sender's* $4N$'th to $6N$'th work functions, counting the sender's current work function as number 0. Defining this notion precisely is the subject of Section 4, but the general idea is simple: the receiver is invoked when it sees the information wavefront that the sender sees in $4N$ to $6N$ execution steps.

In some cases, the ability to synchronize the arrival of a message with some element of the data stream is very important. For example, `CheckFreqHop` knows that the transmitter will change the frequency between $4N$ and $6N$ steps later, in terms of the frame that `CheckFreqHop` is inputting. To ensure that the radio changes frequencies at the same time–so as not to lose any data at the old or new frequency–`CheckFreqHop` instructs the receiver to switch frequencies when the *receiver* sees one of the last data items at the old frequency.

If the receiver of a message is a stream instead of a filter, then the message delivery is timed with respect to the first (most upstream) filter in the stream. We are still formalizing the message delivery semantics in cases where the receiver is a stream that has no unique first filter (e.g., a SplitJoin with NULL splitter). Note that the stream itself can receive a message even though the timing is in terms of filter-to-filter communication.

Portals for broadcast messaging. StreaMIT also has support for modular broadcast messaging. When a sender wants to send a message that will invoke method M of the receiver R upon arrival, it does not call M on the object R. Rather, it calls M on a *Portal* of which R is a member. Portals are typed containers that forward all messages they receive to the elements of the container. Portals could be useful in cases when a component of a filter library needs to announce a message (e.g., that it is shutting down) but does not know the list of recipients; the user of the library can pass the filter a Portal containing all interested receivers. As for message delivery constraints, the user specifies a single time interval for each message, and that interval is interpreted separately (as described above) for each receiver in the Portal.

In a language with generic data types, a Portal could be implemented as a templated list. However, since Java does not yet support templates, we automatically generate an `<X>Portal` class for every class and interface `<X>`. Our syntax for using Portals is evident in the `TrunkedRadio` class in Figure 8.

3.3.2 Rationale

Stream programs present a challenge in that filters need both regular, high-volume data transfer and irregular, low-volume control communication. Moreover, there is the problem of reasoning about the relative "time" between filters when they are running asynchronously and in parallel.

A different approach to messaging is to embed control messages in the data stream instead of providing a separate mechanism for dynamic message passing. This does have the effect of associating the message time with a data item, but it is complicated, error-prone, and leads to unreadable code. Further, it could hurt performance in the steady state (if each filter has to check whether or not a data item is actual data or control, instead) and complicates compiler analysis, too. Finally, one can't send messages upstream without creating a separate data channel for them to travel in.

Another solution is to treat messages as synchronous method calls. However, this delays the progress of the stream when the message is en route, thereby degrading the performance of the program and restricting the compiler's freedom to reorder filter executions.

We feel that the StreaMIT messaging model is an advance in that it separates the notions of low-volume and high-volume data transfer–both for the programmer and the compiler–without losing a well-defind semantics where messages are *timed* relative to the high-volume data flow. Further, by separating message communication into its own category, fewer connections are needed for steady-state data transfer and the resulting stream graphs are more amenable to structured stream programming.

3.4 Re-Initialization

3.4.1 StreaMIT Approach

One of the characteristics of a streaming application is the need to occaisionally modify the structure of part of the stream graph. StreaMIT allows these changes through a re-initialization mechanism that is integrated with its messaging model. If a sender targets a message at the `init` function of a stream or filter S, then when the message arrives, it re-executes the initialazation code and replaces S with a new version of itself. However, the new version might have a different structure than the original if the arguments to the `init` call on re-initialization were different than during the original initialization.

When an init message arrives, it does not kill all of the data that is in the stream being re-initialized. Rather, it *drains* the stream until the wavefront of information (as defined for the messaging model) from the top of the stream has reached the bottom. The draining occurs without consuming any data from the input channels to the re-initialized region. Instead, a `drain` function of each filter is invoked to provide input when its other input source is frozen. (Each filter can override the `drain` function as part of its definition.) If the programmer prefers to kill the data in a stream segment instead of draining it, this can be indicated by sending an extra argument to the message portal with the re-initialization message.

3.4.2 Rationale

Re-initialization is a headache for stream programmers because–if done manually–the entire runtime system could be put on hold to re-initialize a portion of the stream. The interface to starting and stopping streams could be complicated when there is not an explicit notion of initialization time vs. steady-state execution time, and ad-hoc draining techniques could risk losing data or deadlocking the system.

StreaMIT improves on this situation by abstracting the re-initialization process from the user. That is, no auxillary control program is needed to drain the old streams and create the new structure; the user need only trigger the reinitialization process through a message. Additionally, any hierarchical stream construct automatically becomes a possible candidate for re-initialization, due to the well-defined stream structure and the simple interface with the `init` function. Finally, it is easy for the compiler to recognize stream re-initialization possibilities and to account for all possible configurations of the stream flow graph during analysis and optimization.

3.5 Latency Constraints

Lastly, StreaMIT provides a simple way of restricting the latency of an information wavefront in traveling from the input of one filter to the output of a downstream filter. Issueing the directive `MAX_LATENCY(A, B, n)` from within an `init` means that A can only execute up to the wavefront of information that B will see after n invocations of its own work function.

In the case that A is a stream instead of a filter, then the latency is with regards to the most upstream filter of A; likewise, if B is a stream, then the latency is with regards to the most downstream filter in B. We are still in the process of formalizing the semantics in cases when there is no unique upstream or downstream filter in these streams.

4. SEMANTICS OF TIME

In this section we develop a more formal semantics for the message delivery guarantees described above. The timing model in StreaMIT is unique in that all time is relative to *information wavefronts*–that is, two independent filters can describe a common time only in terms of when the *effects* of one filter's execution are seen by the other. Thus, although each filter's `work` function is invoked asynchronously without any notion of global time, two invocations of a work function occur at the same "information-relative time" if they operate on the same information wavefront.

To define this notion more precisely, we present transfer functions that describe the flow of information across filters and streams. Using these transfer functions, we translate message delivery constraints into a set of constraints on the execution schedule of the stream graph. Finally, we use these scheduling constraints to formulate an operational semantics for messaging and latency in StreaMIT.

4.1 Information Flow

The concept of information flow is central to the streaming domain. When an item enters a stream, it carries with it some new information. As execution progresses, this information cascades through the stream, effecting the state of

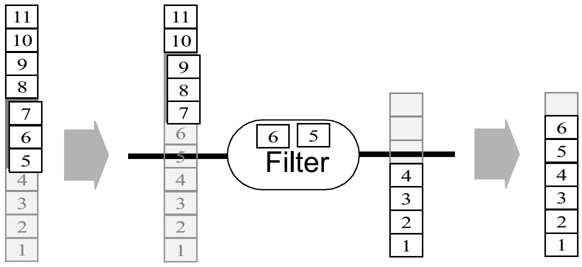


Figure 9: A filter's input and output tapes during an execution step. With each step, the filter pushes two items, pops two items, and peeks at three additional items. The initial state of the input tape is shown at left. The center shows the filter with both input and output tapes during the invocation of `work`. The final state of the output tape is shown at right

filters and the values of new data items which are produced. We refer to an "information wavefront" as the sequence of filter executions that first sees the effects of a given input item. This wavefront is well-defined even in the presence of rate-changing filters that peek or pop a different number of items than they push. To formalize the wavefront, we introduce some new representations for the state of the stream graph. Consider that in place of each data channel there is an infinite "tape" which contains the history of values that have been pushed onto the channel (see Figure 9). Now consider the following functions:

- Given that there are x items on tape a, the maximum number of items that can appear on tape b is $max_{a \to b}(x)$.
- Given that there are x items on tape b, the minimum number of items that must appear on tape a is $min_{a \leftarrow b}(x)$.

Note that these functions are only defined over pairs of tapes (a, b) where a is *upstream* of b–that is, where there is a directed path in the stream graph from the filter following a to the filter preceding b. We will say that the filter b is *downstream* of a under exactly these same conditions.

The max and min functions are related to the information wavefront in the following sense. The item at position $y = min_{a \leftarrow b}(x)$ of tape a is the latest item on tape a to *affect* the item at position x of tape b. This is because item x on tape b can be produced if and only if tape a contains at least y items. Note that this effect might be via a control dependence rather than a data dependence–for instance, if item y needed to pass through a round-robin joiner before some data from another stream could be routed to tape b. However, when speaking of the information wavefront, we only consider information passed through the data streams; if a data item affects another via a low-latency downstream message, then this effect could jump ahead of the wavefront.

We now turn to deriving expressions for $max_{a \to b}$ and $min_{a \leftarrow b}$. Doing so will allow us to formalize the semantics of messaging and latency in StreaMIT, as well as enabling static verification techniques such as deadlock and overflow detection.

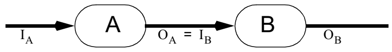


Figure 10: Stream construct with labeling.

4.1.1 Filters

Consider a filter A that peeks $peek_A$, pops pop_A, and pushes $push_A$ data items on every execution step. Further, let us denote the input and output tapes of A by I_A and O_A, respectively. We now turn our attention to finding $max_{I_A \to O_A}$ and $min_{I_A \leftarrow O_A}$, describing the transfer of information across the filter A.

To derive $max_{I_A \to O_A}(x)$, observe that the filter can execute so long as it does not peek beyond the x'th item on the input tape, I_A. After the n'th execution, it has popped $n * pop_A$ items, peeked up to $n * pop_A + (peek_A - pop_A)$, and pushed $n * push_A$ items. Thus, it can execute $n = \lfloor (x - (peek_A - pop_A))/pop_A \rfloor$ times, leaving the following expression for $max_{I_A \to O_A}(x)$:

$$max_{I_A \to O_A}(x) =$$

$$\begin{cases} push_A * \left\lfloor \frac{x-(peek_A - pop_A)}{pop_A} \right\rfloor & if \quad x \geq (peek_A - pop_A) \\ 0 & if \quad x < (peek_A - pop_A) \end{cases}$$

By identical reasoning, the reader can verify the following expression for $min_{I_A \leftarrow O_A}(x)$:

$$min_{I_A \leftarrow O_A}(x) = \left\lceil \frac{x}{push_A} \right\rceil * pop_A + (peek_A - pop_A)$$

4.1.2 Pipelines

Let us now derive expressions for min and max in the case of pipelined filters. In the base case, consider that two filters are connected, with the output of A feeding into the input of B (see Figure 10). We are seeking $max_{I_A \to O_B}(x)$: the maximum number of items that can appear on tape O_B given that there are x items on tape I_A. Observing that a maximum of $max_{I_A \to O_A}(x)$ items can appear on tape O_A, and that O_A must equal I_B since the filters are connected, we see that a maximum of $max_{I_B \to O_B}(max_{I_A \to O_A}(x))$ items can appear on O_B:

$$max_{I_A \to O_B} = max_{I_B \to O_B} \circ max_{I_A \to O_A}$$

In the case of $min_{I_A \leftarrow O_B}(x)$, the order of composition is reversed: given that there are x items on tape O_B, a minimum of $min_{I_B \leftarrow O_B}(x)$ are on tape I_B, and since $O_B = I_A$, we have that a minimum of $min_{I_A \leftarrow O_A}(min_{I_B \leftarrow O_B}(x))$ items appear on I_A, leaving:

$$min_{I_A \leftarrow O_B} = min_{I_A \leftarrow O_A} \circ min_{I_B \leftarrow O_B}$$

By identical reasoning, these composition laws hold for pipelined streams as well as filters. That is, given tapes x, y, and z, we have that:

$$\begin{aligned} max_{x \to z} &= max_{y \to z} \circ max_{x \to y} \\ min_{x \leftarrow z} &= min_{x \leftarrow y} \circ min_{y \leftarrow z} \end{aligned} \tag{1}$$

However, there are some restrictions on these definitions. They only apply when there is a downstream path P_1 from the filter following x to the filter preceding y, a downstream path P_2 from the filter following y to the filter preceding z, and the paths P_1 and P_2 are non-overlapping. This restriction prevents the successive composition of transfer functions around feedback loops, thereby ensuring a unique definition for all pairs (x, z) where there is a downstream path from x to z.

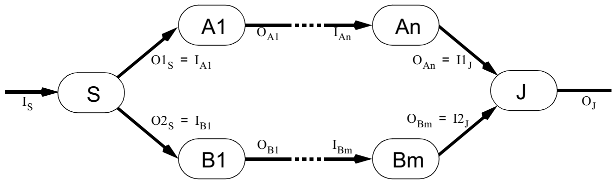


Figure 11: SplitJoin construct with labeling.

4.1.3 SplitJoins

We now derive min and max in the case of a SplitJoin, as pictured in Figure 11. For the splitter S there are two output tapes; let us denote them by $O1_S$ and $O2_S$. Similarly, let us denote the two input tapes of the joiner J by $I1_J$ and $I2_J$. We derive below the transfer functions the round robin and duplicate/combine nodes. Note that the duplicate/combine nodes can be simulated with round robins and duplicating filters, but we provide the transfer functions anyways to simplify the semantic analysis of a program. We have yet to derive these expressions for the weighted round robin nodes.

Round robin splitter. In the case of a round-robin splitter, the items from the input tape are alternately routed to the output tapes, with the first item going onto tape $O1_S$. By this definition, we can see that the splitter's max is defined as follows:

$$\begin{aligned} max_{I_S \to O1_S}(x) &= \left\lceil \frac{x}{2} \right\rceil \\ max_{I_S \to O2_S}(x) &= \left\lfloor \frac{x}{2} \right\rfloor \end{aligned}$$

To derive the min function across a splitter, observe that the input tape need only progress so far as to produce the items on the emptier output tape. That is, we need to consider the number of items on both of the splitter's output to determine the minimum number of items that are needed at its input. Thus, our min function has two arguments: the first corresponding to $O1_S$ and the second corresponding to $O2_S$. The equation is as follows:

$$min_{I_S \leftarrow (O1_S, O2_S)}(x_1, x_2) = MIN(2 * x_1 - 1, 2 * x_2)$$

Round robin joiner. The rules for a round robin joiner are in some sense dual to those of the round robin splitter. Again assuming that items are alternately drawn from the input tapes, starting with $I1_J$, we have that:

$$\begin{aligned} min_{I1_J \leftarrow O_J}(x) &= \left\lceil \frac{x}{2} \right\rceil \\ min_{I2_J \leftarrow O_J}(x) &= \left\lfloor \frac{x}{2} \right\rfloor \end{aligned}$$

Again, the max function takes two arguments, corresponding to the number of items on $I1_J$ and $I2_J$, respectively:

$$max_{(I1_J, I2_J) \to O_J}(x_1, x_2) = MIN(2 * x_1 - 1, 2 * x_2)$$

Duplicate splitter. Clearly, the max function of a duplicate splitter is simply the identity function, since it maps

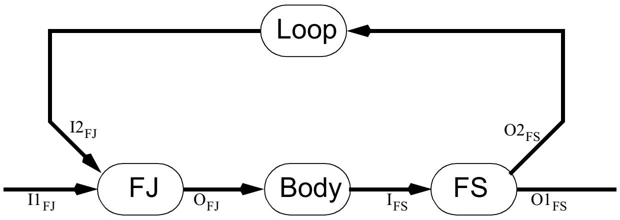


Figure 12: FeedbackLoop construct with labeling.

each element on the input tape to the same location on the output tapes:

$$max_{I_S \to O1_S}(x) = x$$
$$max_{I_S \to O2_S}(x) = x$$

The *min* function is similar, except that–like the round robin split–the input need only progress as far as the lesser output:

$$min_{I_S \leftarrow (O1_S, O2_S)}(x_1, x_2) = MIN(x_1, x_2)$$

Combine joiner. The combine joiner is simply the dual of the duplicate splitter, with transfer functions that the reader can verify as follows:

$$max_{(I1_J, I2_J) \to O_J}(x_1, x_2) = MIN(x_1, x_2)$$
$$min_{I1_J \leftarrow O_J}(x) = x$$
$$min_{I2_J \leftarrow O_J}(x) = x$$

4.1.4 FeedbackLoops

We have to be careful when defining the transfer functions for feedback loops (see Figure 12). The feedback splitter FS serves as a normal splitter, and has the same *min* and *max* functions as defined above. However, the feedback joiner FJ is slightly different than a standard joiner, since during the first few executions it fabricates values from the loop body before they appear on the input tape. The transfer function must take special account of these initial values, since they never appear on $I2_{FJ}$, the input tape from the loop body. This is because we model the initialization of FeedbackLoops by feeding the joiner the initial values directly instead of pushing them onto a channel.

Let n be the number of initial values that are provided to the feedback joiner before values from the feedback loop are read. Let J be a normal COMBINE or ROUND_ROBIN joiner as defined for SplitJoins. Now, let us define the transfer functions for FJ, the feedback joiner.

The *min* function for the main stream is as before:

$$min_{I1_{FJ} \leftarrow O_{FJ}} = min_{I1_J \leftarrow O_J}$$

However, we must offset by n when considering the *min* function that draws from the loop's tape:

$$min_{I2_{FJ} \leftarrow O_{FJ}}(x) = min_{I2_J \leftarrow O_J}(x) - n$$

Finally, the *max* function must be similarly shifted for the input from the loop:

$$max_{(I1_{FJ}, I2_{FJ}) \to O_{FJ}}(x_1, x_2) = max_{(I1_J, I2_J) \to O_J}(x_1, x_2 + n)$$

4.1.5 Summary

We have derived expressions for $max_{a \to b}$ and $min_{a \leftarrow b}$ for when a and b are the respective input and output to 1) a filter, 2) a pipeline, 3) a split or join, and 4) a feedback split or join. By composing these expressions following Equation 1, we can arrive at values of $max_{a \to b}$ and $min_{a \leftarrow b}$ for all pairs of tapes (a, b) where there is some directed path through the stream graph–that is, along the direction of data flow–from the filter reading from tape a to the filter writing to tape b.

Ommitted from the above analysis are: 1) weighted round robin nodes, 2) filters that push or pop items from within their `init` functions, and 3) message handlers that send messages themselves. We hope to address these issues in a future document.

4.2 Semantics

Equipped with definitions of $max_{a \to b}$ and $min_{a \leftarrow b}$, we can now address the semantics of StreaMIT's message delivery and latency guarantees.

4.2.1 Messages

Suppose that filter A sends a message to filter B with latency λ, where λ is any integer. In λ invocations of A's `work` function, A will produce one or more data items d. Now, the messaging system guarantees that:

1. If B is upstream of A, then B will receive the message immediately following the last invocation of its `work` function which produces items that affect d.
2. If B is downstream of A, then B will receive the message immediately preceding the first invocation of its `work` function which produces items that are effected by d.
3. If B runs in parallel with A, then the message timing is in terms of a shared set of splitters and joiners. We are developing semantics for this case, but they are beyond the scope of this paper.

These guarantees can be expressed more formally as a set of constraints on the number of items on certain tapes in the system. We will use $n(t)$ to represent the number of items on tape t at a given point of execution. Again, suppose that filter A sends a message to filter B with latency λ, where λ is any integer. Let s be equal to $n(O_A)$ at the time that the message was sent. We have that:

1. If B is upstream of A, the message will be delivered when:

   $$n(O_B) = min_{O_B \leftarrow O_A}(s + push_A * \lambda) \qquad (2)$$

   That is, $s + push_A * \lambda$ is the number of items on A's output tape after producing the data of interest. Then, $y = min_{O_B \leftarrow O_A}(s + push_A * \lambda)$ is the latest item on B's output tape that affects the data of interest. The message should be delivered immediately after the work function producing this item, which occurs when the item count $n(O_B)$ equals y, as specified by the constraint.

2. If B is downstream of A, the message will be delivered when:

$$n(O_B) = max_{O_A \to O_B}(s + push_A * (\lambda - 1)) \qquad (3)$$

That is, $s + push_A * (\lambda - 1)$ is the number of items on A's output tape before pushing the data of interest, and $y = max_{O_A \to O_B}(s + push_A * (\lambda - 1))$ is the maximum number of items on B's output tape as a result of the outputs of A. Thus, when A pushes the next set of data, it could affect the data that will be pushed next onto the output tape of B. (Note that the next set of data from A might not be sufficient to calculate the next set on B's output, but it could affect it nonetheless.) The message must be delivered immediately before this effected data appears on B's output, so the number of items $n(O_B)$ on B's output must equal y.

4.2.2 Latency

Each directive `MAX_LATENCY(A, B, n)` has the same effect as defining a message from filter B to upstream filter A with latency n.

4.2.3 Scheduling

We can fully define the possible sequences of filter executions as a set of constraints on the number of items on each tape in the stream graph. This is useful not only from the perspective of semantics, but for compiler analysis of the space of valid schedules. To begin the analysis, we formulate the constraints imposed by message delivery guarantees on the number of items on each tape.

Suppose that a filter A might send a message to filter B with a maximum latency of λ during any invocation of its work function. Then we must constrain the execution of B to make sure that it is not too far ahead to receive the message with the given latency. That is, we can only execute B so long as $n(O_B)$–the item count on its output tape–does not exceed the count when a message would be delivered. Recalling the expression for message delivery time (Equations 2 and 3), this constraint is as follows if B is upstream of A:

$$n(O_B) \leq min_{O_B \leftarrow O_A}(n(O_A) + push_A * \lambda) \qquad (4)$$

and as follows if B is downstream of A:

$$n(O_B) \leq max_{O_A \to O_B}(n(O_A) + push_A * (\lambda - 1)) \qquad (5)$$

The guarantees for latency are treated identically to message guarantees, as fitting with the semantics of latency as described above.

Defining the schedule. We now have a set of constraints expressing whether or not a given set of tapes respects the latency and message delivery guarantees in a program. We will now incorporate these constraints into an operational semantics that defines a legal sequences of filter executions.

We represent a stream graph as a configuration $(\langle p(t_1), n(t_1)\rangle, \langle p(t_2), n(t_2)\rangle, \ldots, \langle p(t_k), n(t_k)\rangle)$, where $p(t)$ represents the number of items that have been popped from tape t, and $t_1 \ldots t_k$ are the tapes in the stream graph. Obviously, we have the constraint that $p(t) \leq n(t)$ for each tape t, since a filter can only pop as many items as have appeared on its input tape.

When the program begins, no items have been pushed or popped from any data channels. Thus, each tape is empty, and the starting configuration C_0 is simply the zero vector. It is possible that the initial configuration violates some of the constraints imposed by the messaging and latency constructs, in which case the compiler can inform the programmer that the delivery constraints requested in the program are unsatisfiable.

Let $\mathcal{P}(C)$ denote whether or not the constraints in Equations 4 and 5 are satisfied for all filters in a stream graph with configuration C. We can then write the transition function between configurations as follows:

$$\frac{\begin{array}{c} n(I_A) - p(I_A) \geq peek_A; \\ (\ldots, \langle p(I_A), n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A)\rangle, \ldots); \\ \mathcal{P}((\ldots, \langle p(I_A) + pop_A, n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A) + push_A\rangle, \ldots)); \end{array}}{(\ldots, \langle p(I_A) + pop_A, n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A) + push_A\rangle, \ldots)}$$

There are two components of this rule. On the first line, we state that, for filter A to fire, there must be at least $peek_A$ items on A's input tape that have not yet been popped. Secondly, we express that once A has fired, the new configuration must satisfy the messaging and latency constraints $\mathcal{P}$. The new configuration differs from the original only in that pop_A items have been popped from A's input and $push_A$ items have been pushed to A's output.

Bounding the buffer size. It is a straightforward extension to incorporate a constraint on the maximum number of live items in the stream. This could be useful both from a language perspective, in which a user might wish to constrain the buffer size, or from a compiler perspective, in which the scheduler is interested in constraining the number of live items.

The live items in a given configuration are those that have been pushed to a channel but have not yet been popped. That is, the buffer size needed for a configuration $(\langle p(t_1), n(t_1)\rangle, \ldots, \langle p(t_k), n(t_k)\rangle)$ is $\sum_{i=0}^{k} n(t_i) - p(t_i)$. If we wish to bound the number of live items to MAXITEMS, then, we need only add one condition to the transition rule:

$$\frac{\begin{array}{c} n(I_A) - p(I_A) \geq peek_A; \\ (\ldots, \langle p(I_A), n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A)\rangle, \ldots); \\ \mathcal{P}((\ldots, \langle p(I_A) + pop_A, n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A) + push_A\rangle, \ldots)); \\ \sum_{i=0}^{k} n(t_i) - p(t_i) \leq MAXITEMS; \end{array}}{(\ldots, \langle p(I_A) + pop_A, n(I_A)\rangle, \ldots, \langle p(O_A), n(O_A) + push_A\rangle, \ldots)}$$

4.3 Program Verification

A number of program analysis techniques are also enabled by the *min* and *max* functions that we have defined. In particular, it is very simple to compute 1) whether or not the program will deadlock as a result of a starved input channel, and 2) whether or not any buffer will grow without bound during the steady-state execution of the program.

Deadlock detection. The deadlock detection algorithm takes advantage of the fact that the only loops in our stream graph are part of a FeedbackLoop construct. A stream graph will be deadlock-free if and only if there is no net change of output rate in the feedback loop. This can be formulated in terms of the *max* function by requiring that the wavefront from the output of the feedback joiner FJ unto itself is the identity function. However, since we were careful to leave the *max* function undefined over cycles in the stream graph,

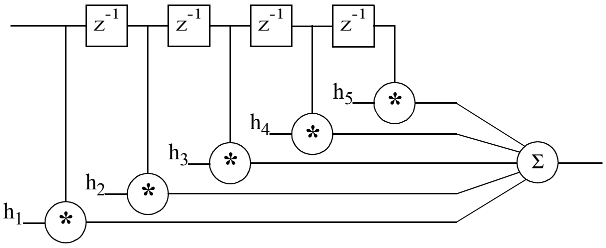


Figure 13: A block diagram of a five tap FIR filter.

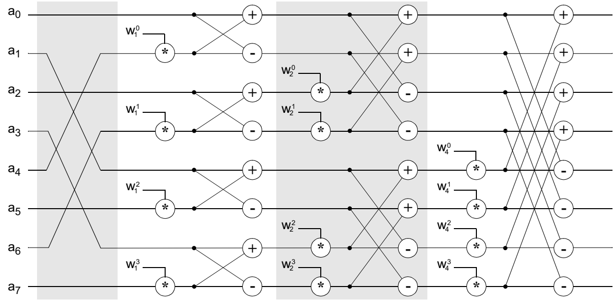


Figure 14: The multi-stage FFT algorithm

we define a new function *maxloop* that maps a given feedback joiner to the information wavefront around the loop:

$$maxloop_{FJ\rightarrow FJ}(x) \equiv max_{I2_{FJ}\rightarrow O_{FJ}} \circ max_{O_{FJ}\rightarrow I2_{FJ}}$$

The order of composition is as in Equation 1 for the composition of pipelines. Also, for the purposes of calculating $maxloop_{FJ\rightarrow FJ}$, one must assume that there are an infinite number of items on tape $I1_{FJ}$; that is, the join from the feedback loop is not limited by the external data source.

Finally, we can state the constraint that the feedback loop must respect. For a loop with declared latency λ, the loop will neither overflow nor deadlock if:

$$maxloop_{FJ\rightarrow FJ}(x) = x + \lambda$$

If $maxloop_{FJ\rightarrow FJ}(x)$ is less than $x + \lambda$, then there will be deadlock in the program.

Overflow detection. There are two places that a buffer can overflow in the stream graph. The first is in a feedback loop, when $maxloop_{FJ\rightarrow FJ}(x)$ (calculated above) is more than $x + \lambda$. The second case is when the parallel streams of a split/join have different production rates. For a splitter S and a joiner J, the production rates will cause an overflow if and only if $max_{O1_S\rightarrow I1_J}(x) - max_{O2_S\rightarrow I2_J}(x)$ is not $O(1)$. This difference could be analyzed by a compiler for every SplitJoin in the stream graph to verify that no buffers will overflow during steady-state execution.

5. DETAILED EXAMPLE

This section describes the Trunked Radio example implementation in Figure 17. The trunked radio is a frequency hopping system. The transition times between the preset frequencies is indicated to the receiver by transmitting a preset tone. Our radio also has an FIR filter to increase the gain of a weak signal. However, in order to save power, this filter is activated only when the signal to noise ratio is low. This implementation relies on transforming the signal into the frequency domain for processing using a FFT. For brevity, we have not shown the RF processing in the front-end and the audio processing in the back-end. The code for the trunked radio demonstrates many features of StreaMIT.

The high-level structure of the radio, graphically shown in Figure 1, is implemented in the class `TrunkedRadio`. The radio has seven stages, where the first three stages operate in the time domain, the last three stages operate in the frequency domain, and there is a conversion phase in between. At this high level, the structure of the system is a pipeline of either six or seven stream stages. The difference is due to the `Booster` stage, which can be active or inactive. The switching on and off of the `Booster` stage, which happens infrequently, is accomplished using a reinitialization message from the `CheckQuality` stage. We also use a message from the `CheckFreqHop` stage to the `RFtoIF` stage to change the baseband when a frequency hop tone is present.

The `RFtoIF` stage modulates the input signal from RF to a frequency band around the current IF frequency. This stage is implemented as a filter that multiplies the current signal with a sine wave at the IF frequency. To support a change in the IF frequency when frequency hopping occurs, the filter contains a `set_freq` method that can be invoked using a message.

The optional `Booster` stage is an FIR filter that is activated when the signal is hard to detect. During normal operation, however, it is deactivated to conserve power. The turning on and off of the filter is controlled by a message. The filter itself, shown in Figure 13, is implemented as a `Filter` that peeks at N elements in the input stream.

The `FFT` stage converts the program from the time domain to the frequency domain using a multi-stage FFT. It is graphically presented in 14. The FFT is composed of a reordering filter and a multi-stage butterfly filter. The StreaMIT representation of the reordering filter (a bit reverse order filter) is given in Figure 15. Note that the complex data re-shuffling is accomplished using a few SplitJoin constructs. A parameterized Butterfly implementation is used to abstract the multi-stage butterfly in the FFT. As shown in Figure 16, the Butterfly filter is also implemented using a combination of SplitJoin constructs.

The StreaMIT implementation of the FFT filter is clean and intuitive. It already has a large amount of pipelined parallelism. Due to the simple and straightforward mapping from the algorithm to the implementation, compiler analyses should be able to extract the parallel structure of the FFT when hardware resources are available.

The next stage, `CheckFreqHop`, checks four different frequencies for the change frequency tone. When the stage detects this tone, it has to change the frequency within a time limit. This task is accomplished by sending a message to the `RFtoIF` stage. The message requires the `RFtoIF` stage to deliver between $4N$ and $6N$ items using the old modulation before changing to the new frequency.

The `CheckQuality` stage checks if the signal has a distinct frequency spectrum. If all the frequencies have similar amplitudes, the stage assumes that the signal-to-noise ratio is low and sends a message to activate the `Booster`. This message is sent using best-effort delivery.

6. RELATED WORK

A large number of programming languages have included a concept of a stream; see [9] for a survey. Those that are perhaps most related to the static-rate version of StreaMIT are synchronous dataflow languages such as LUSTRE [6] and ESTEREL [3] which require a fixed number of inputs to arrive simultaneously before firing a stream node. However, most special-purpose stream languages are functional instead of imperative, and do not contain features such as messaging and support for modular program development that are essential for modern stream applications. Also, most of these languages are so abstract and unstructured that the compiler cannot perform enough analysis and optimization to result in an efficient implementation.

At an abstract level, the stream graphs of StreaMIT share a number of properties with synchronous dataflow (SDF) domain as considered by the Ptolemy project [7]. Each node in an SDF graph produces and consumes a given number of items, and there can be delays along the arcs between nodes (corresponding loosely to items that are peeked in StreaMIT). As in StreaMIT, SDF graphs are guaranteed to have a static schedule, testing for deadlock is decidable, and there are a number of nice scheduling results incorporating code size and execution time [4]. However, previous results on SDF scheduling do not consider constraints imposed by point-to-point messages, and do not include a notion of StreaMIT's information wavefronts, re-initialization, and programming language support.

A specification package used in industry bearing some likeness to StreaMIT is SDL: Specificacation and Description Language [1]. SDL is a formal, object-oriented language for describing the structure and behavior of large, real-time systems, especially for telecommunications applications. It includes a notion of asynchronous messaging based on queues at the receiver, but does not incorporate wavefront semantics as does StreaMIT. Moreover, its focus is on specification and verification whereas StreaMIT aims to produce an efficient implementation.

7. CONCLUSIONS

This paper presents StreaMIT, a novel language for high-performance streaming applications. Streaming programs are an emerging class of very important applications with distinct properties from other recognized application classes.

StreaMIT aims to raise the abstraction level in stream programming, thereby improving programmer productivity and program robustness. This paper presents fundamental programming constructs for the Stream application domain. It identifies information flow as the natural model for reasoning about streaming applications, and presents a formal definition of the language using a semantics of time for StreaMIT.

The second goal of StreaMIT is to provide a compiler that performs stream-specific optimizations to achieve the performance of an expert assembly programmer. Currently, a library-based implementation of StreaMIT is working. We are starting the development of the first compiler implementation of StreaMIT. Another area of future research is to develop a clean high-level syntax for StreaMIT. The Java-based syntax has many advantages, including programmer familiarity, availability of compiler frameworks and a robust language specification. However, the resulting StreaMIT syntax is cumbersome.

8. REFERENCES

[1] Specification and description language. CCITT Recommendation Z.100, ITU, Geneva, 1992.

[2] H. Abelson and G. Sussman. Structure and interpretation of computer programs. MIT Press, Cambridge, MA, 1985.

[3] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996. 189 pages.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] E. A. Lee. Overview of the ptolemy project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[8] S. Rixner, W. J. Dally, U. J. Kapani, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, Dallas, TX, November 1998.

[9] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

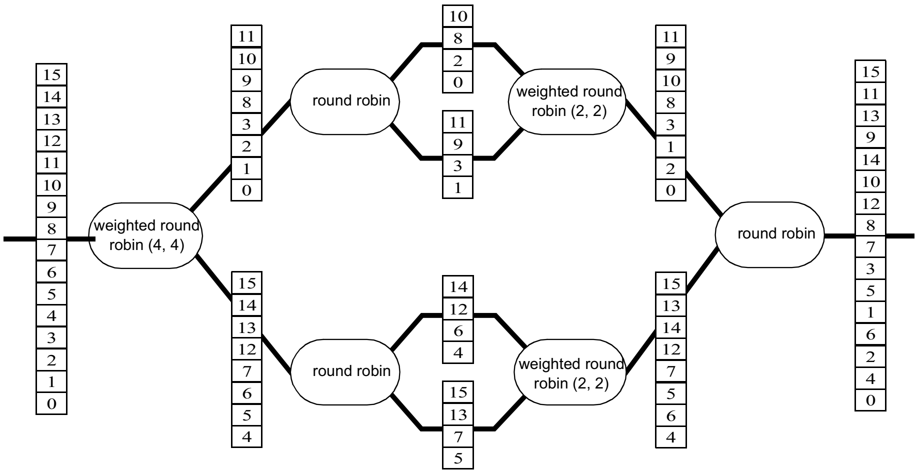


Figure 15: The bit reverse order filter in the FFT, with N=8. The tapes illustrate the data re-shuffling.

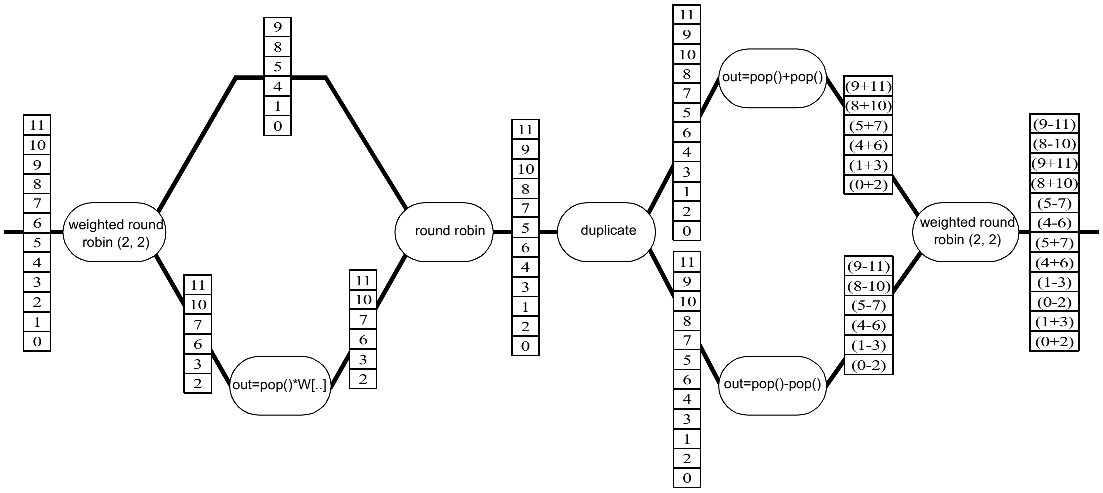


Figure 16: The 4x4 butterfly stage in the FFT. The tapes illustrates the data transformation and computation.

```
class RFtoIF extends Filter {
   Channel input = new FloatChannel();
   Channel output = new FloatChannel();
   int size, count;
   float weight[];
   void init(float f) {
      setf(f);
   }
   void work() {
      output.push(input.pop()*weight[i++]);
      if (count==size) count = 0;
   }
   void setf(float f) {
      count = 0;
      size = CARRIER_FREQ/f*N;
      weight = new float[size];
      for (int i=0; i<size; i++)
         weight[i] = sine(i*PI/size);
   }
}

class FIR extends Filter {
   Channel input = new FloatChannel();
   Channel output = new FloatChannel();
   int N;
   void init(int N) {
      this.N = N;
   }
   void work() {
      float sum = 0;
      for (int i=0; i<N; i++) {
         sum += input.peek(i)*FIR_COEFF[i][N];
      }
      input.pop();
      output.push(sum);
   }
}

class Booster extends Stream {
   void init(int N, boolean adds) {
      if (adds) add(new FIR(N));
   }
}

class Butterfly extends Stream {
   void init(int N, int W) {
      add(new SplitJoin() {
         void init() {
            setSplitter(WEIGHTED_ROUND_ROBIN(N, N));
            add(new Filter() {
               Channel input = new FloatChannel();
               Channel output = new FloatChannel();
               float weights[] = new float[W];
               int curr;
               void init() {
                  for (int i=0; i<W; i++)
                     weights[i] = calcWeight(i, N, W);
                  curr = 0;
               }
               void work() {
                  output.push(input.pop()*weights[curr++]);
                  if(curr>= W) curr = 0;
               }
            });
            add(IDENTITY());
            setJoiner(ROUND_ROBIN);
      }});
      add(new SplitJoin() {
         void init() {
            setSplitter(DUPLICATE);
            add(new Filter() {
               Channel input = new FloatChannel();
               Channel output = new FloatChannel();
               void work() {
                  output.push(input.pop() - input.pop());
               }
            });
            add(new Filter() {
               Channel input = new FloatChannel();
               Channel output = new FloatChannel();
               void work() {
                  output.push(input.pop() + input.pop());
               }
            });
            setJoiner(WEIGHTED_ROUND_ROBIN(N, N));
      }});
}}
```

Figure 17: A Trunked Radio Receiver in StreaMIT.

```
class FFT extends Stream {
   void init(int N) {
      add(new SplitJoin() {
         void init() {
            setSplitter(WEIGHTED_ROUND_ROBIN(N/2, N/2));
            for (int i=0; i<2; i++)
               add(new SplitJoin() {
                  void init() {
                     setSplitter(ROUND_ROBIN);
                     add(IDENTITY());
                     add(IDENTITY());
                     setJoiner(WEIGHTED_ROUND_ROBIN(N/4, N/4));
               }});
            setJoiner(ROUND_ROBIN);
      }});
      for (int i=2; i<N; i*=2)
        add(new Butterfly(i, N));
}}

class CheckFreqHop extends SplitJoin {
   RFtoIFPortal freqHop;
   void init(RFtoIFPortal freqHop) {
      this.freqHop = freqHop;
      setSplitter(WEIGHTED_ROUND_ROBIN(N/4-2,1,1,N/2,1,1,N/4-2));
      int k = 0;
      for (int i=1; i<=5; i++) {
         if ((i==2)||(i==4)) {
            for (int j=0; j<2; j++) {
               add(new Filter() {
                  Channel input = new FloatChannel();
                  Channel output = new FloatChannel();
                  void work() {
                     float val = input.pop();
                     if (val >= MIN_THRESHOLD)
                        freqHop.setf(FREQ[k], new TimeInterval(4*N, 6*N));
                     output.push(val);
                  }
               });
               k++;
            }
         } else add(IDENTITY());
      }
      setJoiner(WEIGHTED_ROUND_ROBIN(N/4-2,1,1,N/2,1,1,N/4-2));
   }
}

class CheckQuality extends Filter {
   Channel input = new FloatChannel();
   Channel output = new FloatChannel();
   float aveHi, aveLo;
   BoosterPortal boosterSwitch;
   boolean boosterOn;
   void init(BoosterPortal boosterSwitch, boolean boosterOn) {
      aveHi = 0; aveLo = 1;
      this.boosterSwitch = boosterSwitch;
      this.boosterOn = boosterOn;
   }
   void work() {
      float val = input.pop();
      aveHi = max(0.9*aveHi, val);
      aveLo = min(1.1*aveLo, val);
      if (aveHi - aveLo < QUAL_BAD_THRESHOLD && !booosterOn) {
         boosterSwitch.init(true, BEST_EFFORT);
         boosterOn = true;
      }
      if(aveHi - aveLo > QUAL_GOOD_THRESHOLD & boosterOn) {
         boosterSwitch.init(false, BEST_EFFORT);
         boosterOn = false;
      }
      output.push(val);
   }
}

class TrunkedRadio extends Stream {
   int N = 64;
   RFtoIFPortal freqHop = new RFtoIFPortal();
   BoosterPortal onOff = new BoosterPortal().
   void init() {
      ReadFromAtoD in = add(new ReadFromAtoD());
      RFtoIF rf2if = add(new RFtoIF(STARTFREQ));
      Booster iss = add(new Booster(N, false));
      add(new FFT(N));
      add(new CheckFreqHop(freqHop));
      add(new CheckQuality(onOff, false));
      AudioBackEnd out = add(new AudioBackEnd());

      freqHop.register(rf2if);
      onOff.register(iss);
      MAX_LATENCY(in, out, 10);
   }
}
```

APPENDIX: Details on Java Syntax
DRAFT

1. JAVA CLASSES

A diagram of the Java class hierarchy for StreaMIT is shown in Figure 18. A summary of the methods in each class is as follows.

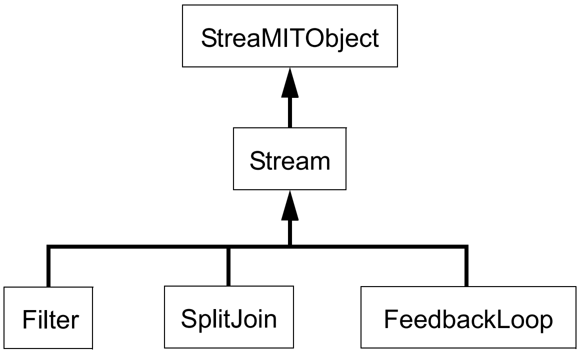


Figure 18: The StreaMIT class hierarchy. Other StreaMIT classes unrelated to this hierarchy are `SplitJoinType`, `Channel`, `Portal`, and `TimeInterval`.

1.1 StreaMITObject

A `StreaMITObject` contains static fields and methods that are useful for all classes in the stream. The other classes extend this type simply so that they can share the same namespace as the constants and methods that it defines.

1.1.1 Fields

`TimeInterval BEST_EFFORT`

This pre-defined time interval indicates that a message should be delivered on a "best-effort" basis, without strict timing guarantees.

`SplitJoinType ROUND_ROBIN`

This is used to specify a round robin splitter or joiner.

`SplitJoinType NULL`

This is used to specify a splitter or joiner that is null (it processes no items).

`SplitJoinType DUPLICATE`

This specifies a duplicating splitter.

`SplitJoinType COMBINE`

This specifies a combining joiner.

1.1.2 Methods

`SplitJoinType WEIGHTED_ROUND_ROBIN(int w1, int w2, ...)`

This specifies a weighted round robin with the given weights. This function does not take a variable number of arguments, but rather is defined for all numbers of arguments that would be likely to occur in a StreaMIT program.

`Filter IDENTITY()`

This returns a `Filter` that outputs exactly the items that it inputs.

`MAX_LATENCY(Stream a, Stream b, int n)`

This directive constrains the schedule such that, at any given time, a can only progress up to the wavefront of information that b will see after n invocations of its own work function.

1.2 Stream extends StreaMITObject

The `Stream` represents a portion of the stream graph that inputs has exactly one input channel and exactly one output channel.

1.2.1 Methods

`void init(user-defined arguments)`

The `init` function is called automatically when the `Stream` is first instantiated; it receives as its arguments the same arguments that were passed to the constructor. Additionally, the `init` function can be called again with a message at runtime to trigger a re-initialization of this stream. The purpose of the function is to initialize child streams and to set parameters used with this stream. The filter can also push, pop, and peek items from its channels from within the `init` function, although this usually isn't necessary.

`Stream add(Stream child)`

The `add` function appends `child` to the current pipeline of blocks comprising this stream and returns `child`. It can only be called from within the `init` function.

`void run()`

The `run` function provides an outside interface for starting the stream. No component of any stream may call `run`.

1.3 Filter extends Stream

The `Filter` is the most basic kind of stream. It contains no child streams, and thus calling `add` is forbidden from within its `init` function. Instead, the `Filter` defines a `work` function that explicitly describes the transfer of input items to output items. A filter has some input and output type, hereafter referred to as `<input-type>` and `<output-type>`, respectively.

1.3.1 Fields

`Channel input`

`Channel output`

These input and output channels must be the first two fields declared in the class. At the line of their declaration, they should be initialized to be a new `<input-type>Channel` and `<output-type>Channel`, respectively. These `Channel` types will be auto-generated.

1.3.2 Methods

`void work()`

The `work` function represents the most fine-grained execution step of the filter. It can read from the input channel, write to the output channel, modify the state of the filter, and send messages.

<input-type> drain(int index)

The drain function specifies what values should be output from this filter if it lies on the boundary of a region that is being re-initialized. For the information in the re-initialized region to drain out, downstream filters will need to input data from the upstream edge of the region. However, we do not want to pull fresh information from outside of the region into the drain, so the drain function is invoked instead to fabricate data. The drain function is successively called with indices 0, 1, 2, ... until the downstream region has drained.

1.4 SplitJoin extends Stream

1.4.1 Methods

A SplitJoin is a set of independent, parallel streams that are contained between a splitter and a joiner.

Splitter setSplitter(SplitJoinType splitter)

This command sets the splitter within a SplitJoin and returns its argument. It must be called in the init function of the SplitJoin.

Joiner setJoiner(SplitJoinType joiner)

This command sets the joiner within a SplitJoin and returns its argument. It must be called in the init function of the SplitJoin.

Stream add(Stream child)

This add function overrides the add function of Stream to append child as a parallel component within the SplitJoin. The first stream to be added is connected to the first port of the splitter and joiner, and likewise with the rest of the streams. This function returns its argument.

1.5 FeedbackLoop extends Stream

The FeedbackLoop provides the means for creating cycles in the stream graph.

1.5.1 Methods

Joiner setJoiner(SplitJoinType joiner)

Stream setBody(Stream stream)

Splitter setSplitter(SplitJoinType splitter)

Stream setLoop(Stream stream)

These methods set the joiner, body stream, splitter, and loop stream for the feedback loop; they each return their argument. Each of them must be called from within the init function.

<varying type> initPath(int index)

The initPath function provides inputs to the joiner at the head of the feedback loop during the initialization period when there are no items on the channels around the loop. The function is called with the number of the item that is being requested, starting from 0.

void setDelay(int delay)

The setDelay function specifies how many times the initPath function is invoked before the joiner starts drawing input items from the feedback channel.

1.6 SplitJoinType

A SplitJoinType represents a compiler-defined configuration for the splitter or joiner in a SplitJoin. For now, the user cannot define custom SplitJoinType's, and the only ones available are those that are constant fields in StreaMITObject.

1.7 Channel

Channels are of a given type <channel-type>, and are auto-generated classes. Their full Java class name is <channel-type>Channel, e.g., IntChannel. They provide typed FIFO queues communicating steady-state data between filters.

1.7.1 Methods

<type> pop()

The pop function removes the item from the end of the channel and returns it.

<type> peek(int index)

The peek function returns the value at index slots from the end of the channel, where peek(0) = pop(). Unlike pop, peek does not remove any items from the channel.

void push(<type> item)

The push function enqueues item onto the front of the channel.

1.8 Portal

Portals provide a means for broadcast messaging within StreaMIT. They are of a given type <portal-type>, and are auto-generated classes. Note that <portal-type> can be either a class or an interface. Their full Java class name is <portal-type>Portal, e.g., MyFilterPortal.

1.8.1 Methods

void register(<portal-type> receiver)

The register method adds receiver to this portal as an object that will be the target of all messages passed to the portal.

all void methods of <portal-type>

A portal automatically defines each of the void methods that is implemented by <portal-type>. Since these methods have no return value, their invocation can act as a message to the receiver object. However, the signature of these methods is modified to accept an extra argument of type TimeInterval, which specifies the timing of the message delivery. When a method is called on the Portal, it is treated as a message and is forwarded to all registered receivers within the given time interval.

1.9 TimeInterval

The TimeInterval class simply provides a wrapper for specifying the upper and lower time limits for a message delivery.

1.9.1 Methods

TimeInterval(int maxTime)

This constructs a time interval with maximum delivery time maxTime. The units of time are according to relative information wavefronts as described in the paper.

`TimeInterval(int minTime, int maxTime)`

This constructs a time interval with minimum delivery time `minTime` and maximum delivery time `maxTime`. The units of time are according to relative information wavefronts as described in the paper.

2. SEMANTIC CHECKING

2.1 Java restrictions

Although this version of StreaMIT is expressed as legal Java syntax, it allows only a small subset of the features of Java. Here we list some of the syntactical elements of Java that fall outside the domain of legal StreaMIT programs.

1. StreaMIT disallows any instantiation, subclassing, or method call to any object from the Java class libraries. The only exception is `Object` itself, which may be subclassed as the basic means of abstraction; however, no member functions of `Object` may be called. Note that this eliminates threads and exceptions from consideration because they require the instantiation of an object from the class library.
2. StreaMIT does not support native method calls.

2.2 StreaMIT restrictions

Though every legal StreaMIT program is a legal Java program, there are legal Java programs–even with the constraints of Section 2.1–that violate higher-level semantic requirements of StreaMIT. We outline these constraints as follows:

1. In this version of StreaMIT, each invocation of a filter's work function must peek, pop, and push a constant number of items. Dynamic rates will be the subject of future work.
2. If two filters are connected, then their corresponding input and output types must match. We postpone a formal treatment of types until a future paper.
3. A given instance of a stream or filter must not appear more than once in the stream graph.
4. A message handler cannot push, pop, or peek items from the input and output channels of a filter. However, a message handler can send another message.
5. There must be no deadlock or buffer overflow in the program. We have developed a simple algorithm to verify that feedback loops and simple round-robin SplitJoins neither deadlock nor overflow.
6. For weighted round robin SplitJoins, we are still developing our analysis. For now, we can at least verify that if the first filter on a branch of a SplitJoin inputs zero items and the splitter is a weighted round robin, then the splitter must have a weight of 0 assigned to the branch. Similarly, if the last filter on a branch output zero items and the joiner is a weighted round robin, then the joiner must assign a weight of 0 to the branch.
7. The numer of inputs and outputs on weighted round robin joiners and splitters must match the number of parallel streams in a SplitJoin.
8. The splitter and joiner in a feedback loop must have two outputs and two inputs, respectively, and must be something other than NULL.