StreamIt: A Language for Streaming Applications*

William Thies, Michal Karczmarek, and Saman Amarasinghe

Laboratory for Computer Science, Massachusetts Institute of Technology
Cambridge, MA 02139
{thies, karczma,saman}@lcs.mit.edu

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.

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 [1], 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 [3] 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. Consequently, most programmers turn to general-purpose languages such as C or C++ to implement stream programs.

There are two reasons that general-purpose languages are inadequate for stream programming. Firstly, they are a mismatch for the application domain.

* For more information about StreamIt, see http://compiler.lcs.mit.edu/streamit.

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 179–196, 2002.
© Springer-Verlag Berlin Heidelberg 2002

That is, they do not provide a natural or intuitive representation of streams, thereby having a negative effect on readability, robustness, and programmer productivity. Moreover, because the widespread parallelism and regular communication patterns of data streams are left implicit in general-purpose languages, compilers are not stream-conscious and do not perform stream-specific optimizations. As a result, performance-critical loops are often hand-coded in a low-level assembly language and must be re-implemented for each target architecture. This practice is labor-intensive, error-prone, and very costly.

Secondly, general-purpose languages are a mismatch for the emerging class of grid-based architectures [4,5,6] that are especially well-suited for stream processing. Perhaps the primary appeal of C is that it provides a "common machine language" for von-Neumann architectures. That is, it abstracts away the idiosyncratic differences between machines, but encapsulates their common properties: a single program counter, arithmetic operations, and a monolithic memory. However, for grid-based architectures, the von-Neumann model no longer holds, as there are multiple instruction streams and distributed memory banks. Thus, C no longer serves as a common machine language–in fact, it provides the wrong abstraction for the underlying hardware, and architecture-specific directives are often needed to obtain reasonable performance. Again, this greatly complicates the job of the programmer and hampers portability.

StreamIt is a language and compiler specifically designed for modern stream programming. The StreamIt language has two goals: first, to provide high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain, and second, to serve as a common machine language for grid-based processors. At the same time, the StreamIt compiler aims to perform stream-specific optimizations to achieve the performance of an expert programmer.

This paper motivates, describes, and justifies the high-level language features of StreamIt, version 1.0. The major limitation of StreamIt 1.0 is that all flow rates in the streams must be static; applications such as compression that have dynamically varying flow rates will be the subject of future work. A large set of applications can be implemented with static rates, and while dynamic rates will require a different runtime model, it will still be essential to fully analyse and optimize static sub-sections in order to obtain high performance.

The paper is organized as follows. In Section 2, we characterize the domain of streaming programs that motivates the design of StreamIt, and in Section 3 we describe the language features in detail. We present an in-depth example of a software radio in Section 4, preliminary results in Section 5, related work in Section 6, and conclusions in Section 7.

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 servers. Examples include systems such as the Click modular router [7] and the

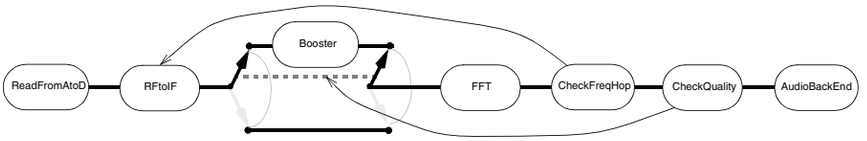


Fig. 1. A block diagram of our frequency-hopping software radio

Spectrumware software radio [8,9]; specifications such as the Bluetooth communications protocol [10], the GSM Vocoder [11], and the AMPS cellular base station[12]; and almost any application developed with Microsoft's DirectShow library [13], Real Network's RealSDK [14] or Lincoln Lab's Polymorphous Computing Architecture [15].

We have identified a number of properties that are common to such applications–enough so as to characterize them as belonging to a distinct class of programs, which we will refer to as *streaming applications.* We believe that the salient characteristics of a streaming application are as follows:

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 codes, which manipulate a fixed input set 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

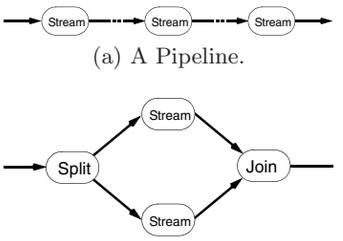
```
class FIRFilter extends Filter {
   float[] weights;
   int N;

   void init(float[] weights) {
      setInput(Float.TYPE); setOutput(Float.TYPE);
      setPush(N); setPop(1); setPeek(N);
      this.weights = weights;
      this.N = weights.length;
   }

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

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

Fig. 2. An FIR filter in StreamIt



(a) A Pipeline.

(b) A SplitJoin.

(c) A FeedbackLoop.

Fig. 3. Stream structures supported by StreamIt

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.

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[1]. Using Java has many advantages, including programmer familiarity, availability of compiler frameworks and a robust language specification. However, the resulting syntax can be cumbersome, and in the future we plan to develop a cleaner and more abstract syntax that is designed specifically for stream programs.

[1] However, for the sake of brevity, the code fragments in this paper are sometimes lacking modifiers or methods that would be needed to make them strictly legal Java.

3.1 Filters

The basic unit of computation in StreamIt is the Filter. An example of a Filter from our software radio (see Figure 1) is the `FIRFilter`, shown in Figure 2. The central aspect of a filter is the `work` function, which describes the filter's most fine grained execution step in the steady state. Within the `work` function, a filter can communicate with neighboring blocks using the `input` and `output` channels, which are FIFO queues declared as fields in the Filter base 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, and 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.

A major restriction of StreamIt 1.0 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. In fact, as described below, the input and output rates must be declared in the filter's `init` function. If a filter violates the declared rates, StreamIt throws a runtime error and the subsequent behavior of the program is undefined. We plan to support dynamically changing rates in a future version of StreamIt.

Each Filter also contains an `init` function, which is called at initialization time. The `init` function serves two purposes. Firstly, it is for the user to establish the initial state of the filter. For example, the FIRFilter records `weights`, the coefficients that it should use for filtering. 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 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[2].

The second purpose of the `init` function is to specify the filter's I/O types and data rates to the StreamIt compiler. The types are specified with calls to `setInput` and `setOutput`, while the rates are specified with calls to `setPush`, `setPop`, and `setPeek`. The `setPeek` call can be ommitted if the peek count is the same as the pop count.

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 4). 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,

[2] 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.

```
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);
  }
}
```

Fig. 4. 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();
}
```

Fig. 5. 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

making explicit the parallelism and inter-filter communication while hiding the grungy details of scheduling and optimization from the programmer.

One could also use an object-oriented language to implement a stream abstraction (see Figure 5). 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 executes 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.

```
class Delay extends Filter {
  void init(int delay) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(1); setPop(1);
    for (int i=0; i<delay; i++)
      output.push(0);
  }
  void work() { output.push(input.pop()); }
}

class EchoEffect extends SplitJoin {
  void init() {
    setSplitter(Duplicate());
    add(new Delay(100));
    add(new Delay(0));
    setJoiner(RoundRobin());
  }
}

class AudioEcho extends Pipeline {
  void init() {
    add(new AudioSource());
    add(new EchoEffect());
    add(new Adder());      <-- Adder is defined in Figure 8.
    add(new Speaker());
  }
}
```

Fig. 6. An echo effect in StreamIt

```
class Fibonnacci extends FeedbackLoop {
  void init() {
    setDelay(2);
    setJoiner(RoundRobin(0,1));
    setBody(new Filter() {
       void init() {
         setInput(Integer.TYPE);
         setOutput(Integer.TYPE);
         setPush(1); setPop(1); setPeek(2);
       }
       void work() {
          output.push(input.peek(0)+input.peek(1));
          input.pop();
       }
    });
    setSplitter(Duplicate());
  }

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

Fig. 7. A FeedbackLoop version of Fibonnacci

3.2 Connecting Filters

StreamIt provides three constructs for composing filters into a communicating network: Pipeline, SplitJoin, and FeedbackLoop (see Figure 3). Each structure specifies a pre-defined way of connecting filters into a single-input, single-output block, which we will henceforth refer to as a "stream". That is, a stream is any instance of a Filter, Pipeline, SplitJoin, or FeedbackLoop. Every StreamIt program is a hierarchical composition of these stream structures.

The **Pipeline** construct is for building a sequence of streams. Like a Filter, a Pipeline has an `init` function that is called upon its instantiation. Within `init`, component streams are added to the Pipeline via successive calls to `add`. For example, in the `AudioEcho` in Figure 6, the `init` function adds four streams to the Pipeline: an `AudioSource`, an `EchoEffect`, an `Adder`, and a `Speaker`. This sequence of statements automatically connects these four streams in the order specified. Thus, there is no `work` function in a Pipeline, as the component streams fully specify the behavior. The channel types and data rates are also implicit from the connections.

Each of the stream constructs can either be executed on its own, or embedded in an enclosing stream structure. The `AudioEcho` can execute independently, since the first component consumes no items and the last component produces no items. However, the `EchoEffect` must be used as a component, since the first stream inputs items and the last stream outputs items. When a stream is embedded in another construct, the first and last components of the stream are implicitly connected to the stream's neighbors in the parent construct.

The **SplitJoin** construct is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. As in a Pipeline, the components of a SplitJoin are specified with successive calls to `add` from the

`init` function. For example, the `EchoEffect` in Figure 6 adds two streams that run in parallel, each of which is a `Delay` filter.

The splitter specifies how items from the input of the SplitJoin are distributed to the parallel components. For simplicity, we allow only compiler-defined splitters, of which there are three types: 1) *Duplicate*, which replicates each data item and sends a copy to each parallel stream, 2) *RoundRobin*(i_1, i_2, ..., i_k), which sends the first i_1 data items to the stream that was added first, the next i_2 data items to the stream that was added second, and so on, and 3) *Null*, which means that none of the parallel components require any input, and there are no input items to split. If the weights are ommitted from a RoundRobin, then they are assumed to be equal to one for each stream. Note that RoundRobin can function as an exclusive selector if one or more of the weights are zero.

Likewise, the joiner is used to indicate how the outputs of the parallel streams should be interleaved on the output channel of the SplitJoin. There are two kinds of joiners: 1) *RoundRobin*, whose function is analogous to a RoundRobin splitter, and 2) *Null*, which means that none of the parallel components produce any output, and there are no output items to join. The splitter and joiner types are specified with calls to `setSplitter` and `setJoiner`, respectively. The `EchoEffect` uses a Duplicate splitter so that each item appears both directly and as an echo; it uses a RoundRobin joiner to interleave the immediate signals with the delayed ones. In `AudioEcho`, an `Adder` is used to combine each pair of interleaved signals.

The **FeedbackLoop** construct provides a way to create cycles in the stream graph. The `Fibonacci` stream in Figure 7 illustrates the use of this construct. Each FeedbackLoop contains: 1) a body stream, which is the block around which a backwards "feedback path" is being created, 2) a loop stream, which can perform some computation along the feedback path, 3) a splitter, which distributes data between the feedback path and the output channel at the bottom of the loop, and 4) a joiner, which merges items between the feedback path and the input channel at the top of the loop. These components are specified from within the `init` function via calls to `setBody`, `setLoop`, `setSplitter`, and `setJoiner`, respectively. The splitters and joiners can be any of those for SplitJoin, except for Null. The call to `setLoop` can be ommitted if no computation is performed along the feedback path.

The FeedbackLoop 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; given an index i, `initPath` provides the i'th initial input for the feedback joiner. 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 the `Fibonnacci` example of Figure 7 is another feature of the StreamIt syntax: *inlining*. The definition of any stream 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 [16].

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 Pipelines, 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 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 programmer, the gains in robustness, readability, and compiler analysis are immense. Though graphical programming languages have not gained large-scale acceptance, a graphical editor for StreamIt would have advantages since every stream graph has a precise textual equivalent that could also be edited by the programmer. Further, the hierarchical structure of the stream graph could simplify visualization.

On first glance, the statements within a StreamIt `init` function might appear more like a verbose API than a novel language. However, it was actually a careful design decision to specify all "stream configuration information" via function calls from within the `init` functions. While the current syntax is somewhat tedious, there is great flexibility in this approach, since the user can intermix configuration directives with statements that calculate the configuration parameters. This allows for fully parameterized graph construction–the FFT stream in Figure 8 inputs a parameter `N` and adjusts the number of butterfly stages appropriately. This further improves the modularity and readability of the code.

3.3 Messages

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 our software radio code (see Figure 8), the `CheckFreqHop` stage sends a message upstream to change the frequency of the receiver if it detects that the transmitter is about to change frequencies. The sender can continue to execute while the message is en route, and the `setFreq` 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.


```
class RFtoIF extends Filter {
  int size, count, N;
  float weight[];
  void init(int N, float freq) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(1); setPop(1);
    this.N = N;
    setFreq(freq);
  }
  void work() {
    output.push(input.pop()*weight[count++]);
    if (count==size) count = 0;
  }
  void setFreq(float freq) {
    count = 0;
    size = CARRIER_FREQ/freq*N;
    weight = new float[size];
    for (int i=0; i<size; i++)
      weight[i] = Math.sin(i*PI/size);
  }
}

class CheckFreqHop extends SplitJoin {
  RFtoIFPortal freqPortal;
  void init(RFtoIFPortal freqPortal) {
    this.freqPortal = freqPortal;
    setSplitter(RoundRobin(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() {
            void init() {
              setInput(Float.TYPE);
              setOutput(Float.TYPE);
              setPush(1); setPop(1);
            }
            void work() {
              float val = input.pop();
              if (val >= MIN_THRESHOLD)
                freqPortal.setFreq(
                  FREQ[k], new Latency(4*N,6*N));
              output.push(val);
            }});
          k++;
        }
      } else add(new Identity());
    }
    setJoiner(RoundRobin(N/4-2,1,1,
                         N/2,1,1,N/4-2));
  }
}

class FFT extends Pipeline {
  void init(int N) {
    add(new SplitJoin() {
      void init() {
        setSplitter(RoundRobin(N/2, N/2));
        for (int i=0; i<2; i++)
          add(new SplitJoin() {
            void init() {
              setSplitter(RoundRobin());
              add(new Identity());
              add(new Identity());
              setJoiner(RoundRobin(N/4, N/4));
            }});
        setJoiner(RoundRobin());
      }});
    for (int i=2; i<N; i*=2)
      add(new Butterfly(i, N));
  }
}

class Adder extends Filter {
   void init() {
     setInput(Float.TYPE); setOutput(Float.TYPE);
     setPush(1); setPop(2);
   }
   void work() {
     output.push(input.pop() + input.pop());
   }
}

class CheckQuality extends Filter {
  float aveHi, aveLo;
  BoosterPortal boosterPortal;
  boolean boosterOn;
  void init(BoosterPortal bp, boolean on) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(1); setPop(1);
    aveHi = 0; aveLo = 1;
    this.boosterPortal = bp;  this.boosterOn = on;

  }
  void work() {
    float val = input.pop();
    aveHi = max(0.9*aveHi, val);
    aveLo = min(1.1*aveLo, val);
    if (aveHi - aveLo < FAIL_QUAL && !booosterOn) {
      boosterPortal.init(true, BEST_EFFORT);
      boosterOn = true;
    }
    if (aveHi - aveLo > PASS_QUAL && boosterOn) {
      boosterPortal.init(false, BEST_EFFORT);
      boosterOn = false;
    }
    output.push(val);
  }
}

class Booster extends Pipeline {
  void init(int N, boolean enabled) {
    if (enabled)
      add(new FIRFilter(BOOST_WEIGHTS));
  }
}

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

    freqPortal.register(rf2if);
    boosterPortal.register(booster);
    MAX_LATENCY(in, out, 10);
  }
}

class Butterfly extends Pipeline {
  void init(int N, int W) {
    add(new SplitJoin() {
      void init() {
        setSplitter(RoundRobin(N, N));
        add(new Filter() {
          float weight[] = new float[W];
          int curr;
          void init() {
            setInput(Float.TYPE);
            setOutput(Float.TYPE);
            setPush(1); setPop(1);
            for (int i=0; i<W; i++)
              weight[i] = calcWeight(i, N, W);
            curr = 0;
          }
          void work() {
            output.push(input.pop()*weight[curr++]);
            if (curr>=W) curr = 0;
          }});
        add(new Identity());
        setJoiner(RoundRobin());
      }});
    add(new SplitJoin() {
      void init() {
        setSplitter(Duplicate());
        add(new Subtractor());
        add(new Adder());
        setJoiner(RoundRobin(N, N));
      }});
}
```

Fig. 8. StreamIt code for a software radio. Arrows denote the paths of messages

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 each message delivery. 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 output of the *sender's* $4N$'th to $6N$'th work functions, counting the sender's current work function as number 0. Due to space limitations, we cannot define this notion precisely in this paper (see [17,18] for a formal semantics), 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.

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 to 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.

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-defined 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

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 initialization 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.

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.

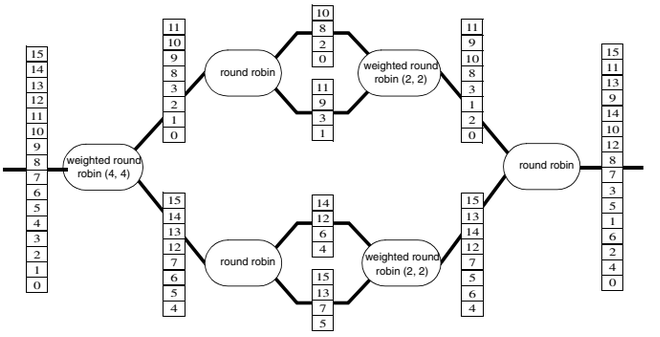


Fig. 9. The bit-reversal phase in the FFT, with N=8. A bit-reversal permutation is one that swaps all elements with indices whose binary representations are the reverse of each other. The Butterfly stage is similar, but ommitted for lack of space

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. Issuing 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.

4 Detailed Example

We now discuss the StreamIt implementation of the Trunked Radio illustrated in Figure 1. The Trunked Radio is a frequency-hopping system in which the receiver switches between a set of known frequencies whenever it hears certain tones from the transmitter.

The toplevel class, `TrunkedRadio`, is implemented as a seven-stage Pipeline (see Figure 8). The `RFtoIF` stage modulates the input signal from RF to a

frequency band around the current IF frequency. To support a change in the IF frequency when frequency hopping occurs, the `RftoIF` filter contains a `setFreq` method that is invoked via a message from the `CheckFreqHop` stage. The message is sent from `CheckFreqHop` with a latency range of $4N$ to $6N$, which means that `RFtoIF` must deliver between $4N$ and $6N$ items using the old modulation scheme before changing to the new frequency.

The optional `Booster` stage provides amplification for weak signals, but is usually turned off to conserve power. The `Booster` is toggled by a re-initialization message from the `CheckQuality` stage, which estimates the signal quality by the shape of the frequency spectrum. If all the frequencies have similar amplitudes, `CheckQuality` 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.

The `FFT` stage converts the signal from the time domain to the frequency domain; please refer to p. 796 of [19] for a diagram of the parallel FFT algorithm. The StreamIt implementation consists of a bit-reversal permutation followed by a series of `Butterfly` stages. The bit-reversal phase illustrates how data can be reshuffled with just a few SplitJoin constructs (see Figure 9). The `Butterfly` stage–which is parameterized to allow for a compact representation of the FFT–also employs SplitJoins to select groups of items for its computation. We believe that the StreamIt version of the FFT is clean and intuitive, as the SplitJoin constructs expose the natural parallelism of the algorithm.

5 Results

We have implemented a fully-functional prototype of the StreamIt compiler as an extension to the Kopi Java Compiler, a component of the open-source Kopi Project [20]. At this time, our compiler is a proof-of-concept and does not yet include the stream-specific optimizations that we are working on; we generate C code that is compiled with a StreamIt runtime library to produce the final executable. We have also developed a library in Java that allows StreamIt code to be executed as pure Java, thereby providing a verification mechanism for the output of the compiler.

The compilation process for streaming programs contains many novel aspects because the basic unit of computation is a stream rather than a procedure. In order to compile stream modules separately, we have developed a runtime interface–analogous to that of a procedure call for traditional codes–that specifies how one can interact with a black box of streaming computation. The stream interface contains separate phases for initialization and steady-state execution; in the execution phase, the interface includes a contract for input items, output items, and possible message production and consumption.

Though we have yet to add optimizations to our compiler, it is nonetheless interesting to evaluate its baseline performance. For this purpose, we developed StreamIt implementations of four applications: 1) A GSM Decoder, which takes GSM-encoded parameters as inputs, and uses these to synthesize audible speech[11], 2) A system from the Polymorphic Computing Architecture

Table 1. Application Characteristics

Benchmark	Lines	Filters	Graph Size
PCA Demo	484	5	7
FM Radio	411	5	27
perftest4	347	5	20
GSM Decoder	3050	11	21

Table 2. Performance Results (in μsec/item)

Benchmark	StreamIt	SpectrumWare	C
PCA Demo	1.3	3.4	N/A
FM Radio	4.9	9.9	N/A
perftest4	330	330	N/A
GSM Decoder	4.88	N/A	.47

(PCA) [15] which encapsulates the core functionality of modern radar, sonar, and communications signal processors, 3) A software-based FM Radio with equalizer, and 4) A performance test from the SpectrumWare system that implements an Orthogonal Frequency Division Multiplexor (OFDM) [8]. Table 1 gives characteristics of the above applications including the number of filters implemented and the size of the stream graph as coded.

Table 2 gives the performance of our compiler by comparing the StreamIt implementation against either the SpectrumWare implementation or (in the case of GSM) a hand-optimized C version. SpectrumWare [8] is a high-performance runtime library for streaming programs, implemented in C++. The StreamIt language offers a higher level of abstraction than SpectrumWare (see Section 3.1), and yet the StreamIt compiler is able to beat the SpectrumWare performance by a factor of two for the PCA Demo and FM Radio.

For the GSM application, the extensively hand-optimized C version incorporates many transformations that rely on a high-level knowledge of the algorithm, and StreamIt performs an order of magnitude slower. However, this version of the compiler is only a prototype, and is not yet intended to compete with hand-coded C. Our code generation strategy currently has many inefficiencies, and in the future we plan to generate optimized assembly code by interfacing with a code generator. We believe that stream-conscious optimizations can improve the performance by an order of magnitude on uniprocessors; moreover, we have yet to consider parallel targets, and this is where we expect to find the most pronounced benefits of the abundant parallelism and regular communication patterns exposed by StreamIt.

6 Related Work

A large number of programming languages have included a concept of a stream; see [3] for a survey. Those that are perhaps most related to StreamIt 1.0 are synchronous dataflow languages such as LUSTRE [21] and ESTEREL [22] which require a fixed number of inputs to arrive simultaneously before firing a stream node. However, most special-purpose stream languages 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 the synchronous dataflow (SDF) domain as considered by the Ptolemy project [23]. 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 and there are a number of nice scheduling results incorporating code size and execution time [24]. 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: Specification and Description Language [25]. 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 and Future Work

This paper presents StreamIt, a novel language for high-performance streaming applications. Stream programs are emerging as a very important class of applications with distinct properties from other recognized application classes. This paper develops fundamental programming constructs for the streaming domain.

The primary goal of StreamIt is to raise the abstraction level in stream programming without sacrificing performance. We have argued that StreamIt's mechanisms for filter definition, filter composition, messaging, and re-initialization will improve programmer productivity and program robustness within the streaming domain.

Also, we believe that StreamIt is a viable common machine language for grid-based architectures (e.g., [4,5,6]), just as C is a common machine language for von-Neumann machines. StreamIt abstracts away the target's granularity, memory layout, and network interconnect, while capturing the notion of independent processors that communicate in regular patterns. We are developing fission and fusion algorithms that can automatically adjust the granularity of a stream graph to match that of a given target.

We have a number of extensions planned for the next version of the StreamIt language. The current version is designed primarily for uniform one-dimensional data processing, but constructs for hierarchical frames of data would be useful for image processing. Moreover, a future version will support dynamically varying I/O rates of the filters in the stream. We expect that such support will require new language constructs–for instance, a type-dispatch splitter that routes items to the components of a SplitJoin based on their type, and a fall-through joiner that pulls items from any stream in a SplitJoin as soon as they are produced.

Our immediate focus is on developing a high-performance optimizing compiler for StreamIt 1.0. As described in [18], the structure of StreamIt can be exploited by the compiler to perform a wide range of stream-specific optimizations. Our goal is to match the performance of hand-coded applications, such that the abstraction benefits of StreamIt come with no performance penalty.

Acknowledgements

The StreamIt compiler was implemented with Michael Gordon and David Maze, with applications support of Jeremy Wong, Henry Hoffman, and Matthew Brown; we also thank Matt Frank for many helpful comments. This work was supported in part by the MIT Oxygen Project and DARPA Grant DBT6396-C-0036.

References

1. Rixner, S., et al: A Bandwidth-Efficient Architecture for Media Processing. In: HPCA, Dallas, TX (1998) 179
2. Abelson, H., Sussman, G.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA (1985) 179
3. Stephens, R.: A Survey of Stream Processing. Acta Informatica **34** (1997) 491–541 179, 193
4. Mai, K., Paaske, T., Jayasena, N., Ho, R., Dally, W., Horowitz, M.: Smart memories: A modular recongurable architecture (2000) 180, 194
5. Waingold, E., et al.: Baring it all to Software: The Raw Machine. MIT-LCS Technical Report 709, Cambridge, MA (1997) 180, 194
6. Sankaralingam, K., Nagarajan, R., Keckler, S., Burger, D.: A Technology-Scalable Architecture for Fast Clocks and High ILP. UT Austin Tech Report 01-02 (2001) 180, 194
7. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M. F.: The click modular router. ACM Trans. on Computer Systems **18** (2000) 263–297 180
8. Tennenhouse, D., Bose, V.: The SpectrumWare Approach to Wireless Signal Processing. Wireless Networks (1999) 181, 193
9. Bose, V., Ismert, M., Welborn, M., Guttag, J.: Virtual radios. IEEE/JSAC, Special Issue on Software Radios (April 1999) 181
10. B. Volume and B. July: Bluetooth Spec. Vol. 1. Bluetooth Consortium (1999) 181
11. Mouly, M., Pautet, M.: The GSM System for Mobile Communications. Cell&Sys, Palaiseau, France (1992) 181, 192
12. EIA/TIA: Mobile station-land station compatibility spec. Tech. Rep. 553 (1989) 181
13. Microsoft Corporation: Microsoft directshow. Online Documentation (2001) 181
14. RealNetworks: Software Developer's Kit. Online Documentation (2001) 181
15. Lebak, J.: Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, MIT Lincoln Laboratory (August 2001) 181, 193
16. Gosling, Joy, Steele: The Java Language Specification. Addison Wesley (1997) 187

17. Thies, B., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. MIT-LCS Technical Memo TM-620, Cambridge, MA (August, 2001) 189
18. Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffmann, H., Brown, M., Amarasinghe, S.: StreamIt: A Compiler for Streaming Applications. MIT-LCS Technical Memo TM-622, Cambridge, MA (December, 2001) 189, 195
19. Cormen, T. H., Leiserson, C. E., Rivest, R. L.: Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill (1990) 192
20. Vincent Gay-Para, Thomas Graf, A. G. L., Wais, E.: Kopi Reference manual. http://www.dms.at/kopi/docs/kopi.html (2001) 192
21. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320 193
22. Berry, G., Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming **19** (1992) 87–152 193
23. Lee, E. A.: Overview of the Ptolemy Project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA (2001) 194
24. Bhattacharyya, S. S., Murthy, P. K., Lee, E. A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers (1996) 189 pages. 194
25. CCITT Recommendation Z.100: Specification and Description Language. ITU, Geneva (1992) 194