StreamIt: A Compiler for Streaming Applications*

William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe

{thies, karczma, mgordon, dmaze, jnwong, hank, morris, saman}@lcs.mit.edu

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

December 19, 2001

ABSTRACT

Streaming programs represent an increasingly important and widespread class of applications that holds unprecedented opportunities for high-impact compiler technology. Unlike sequential programs with obscured dependence information and complex communication patterns, a stream program is naturally written as a set of concurrent filters with regular steady-state communication. The StreamIt language aims to provide a natural, high-level syntax that improves programmer productivity in the streaming domain. At the same time, the language imposes a hierarchical structure on the stream graph that enables novel representations and optimizations within the StreamIt compiler. We define the "stream dependence function", a fundamental relationship between the input channels of two filters in a stream graph. We also describe a suite of stream optimizations, a denotational semantics for validating these optimizations, and a novel phased scheduling algorithm for stream graphs. In addition, we have implemented a prototype of the StreamIt optimizing compiler that is showing promising results.

1. INTRODUCTION

Recent years have seen the proliferation of applications that are based on some notion of a "stream". There is evidence that streaming media applications are already consuming most of the cycles on consumer machines [13], and their use is continuing to grow. The stream abstraction is central to embedded applications for hand-held computers, cell phones, and DSP's, as well as for 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. For example, a number of grid-based architectures have been emerging that are particularly well-suited for stream programming [19, 10, 14], but there is no common machine language that a programmer can use to exploit their common properties while hiding their differences. Thus, most programmers turn to general-purpose languages such as C or C++ to implement stream programs, resorting to low-level assembly codes for loops that require high performance. 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 goal is to raise the abstraction level in the streaming domain, providing a natural, high-level syntax that conceals architectural details without sacrificing performance. To accomplish this goal, the compiler needs to be "stream-aware"–that is, it needs to be able to recognize, analyze, and manipulate data streams as would an expert assembly programmer in lowering an application to a given target. Towards this end, this paper makes the following contributions:

- "Structured" streams as a language construct for enabling novel compiler analyses of stream programs.
- The identification of a fundamental property of a stream graph–the stream dependence function–that establishes a notion of relative time and dependence information.
- A semantic model of structured stream programs that allows one to formulate and validate stream transformations.
- A parallel fusion transformation that collapses several filters into one.
- A suite of optimizations that are specific to the streaming domain.
- A novel phased scheduling algorithm that finds a minimal latency schedule over a structured stream graph.
- A prototype implementation of the StreamIt optimizing compiler that is showing promising results.

2. THE STREAMIT LANGUAGE

In this section we provide a very brief overview of the StreamIt language; please see [17] for a more detailed description. The current version of StreamIt is legal Java syntax to simplify our presentation and implementation, and it is designed to support only streams with static input and output rates. Designing a cleaner syntax and considering dynamically varying rates will be the subject of future work.

*This document is MIT Laboratory for Computer Science Technical Memo LCS-TM-622. A similar version was submitted to PLDI 2002.

```
class Adder extends Filter {
  int N;

  void init (int N) {
    this.N = N;
    input = new Channel(Float.TYPE, N);
    output = new Channel(Float.TYPE, 1);
  }

  void work() {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += input.popFloat();
    }
    output.pushFloat(sum);
  }
}

public class Equalizer extends Pipeline {
  void init(float samplingRate, int N) {
    add(new SplitJoin() {
      void init() {
        int bottom = 2500;
        int top = 5000;
        setSplitter(DUPLICATE());
        for (int i=0; i<N; i++, bottom*=2, top*=2) {
          add(new BandPassFilter(samplingRate, bottom, top));
        }
        setJoiner(ROUND_ROBIN());
    }});
    add(new Adder(N));
  }
}

class FMRadio extends Pipeline {
  void init() {
    add(new DataSource());
    add(new LowPassFilter(samplingRate, cutoffFrequency, numTaps));
    add(new FMDemodulator(samplingRate, maxAmplitude, bandwidth));
    add(new Equalizer(samplingRate, 4));
    add(new Speaker());
  }
}
```

Figure 1: Parts of an FM Radio in StreamIt.

2.1 Filters

The basic unit of computation in StreamIt is the `Filter`. An example of a Filter is the `Adder`, a component of our software radio (see Figure 1). Each `Filter` contains an `init` function that is called at initialization time; in this case, the `Adder` 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 its neighbors using the `input` and `output` channels, which are typed FIFO queues declared within the `init` function. These high-volume channels support the intuitive operations of `push(value)`, `pop()`, and `peek(index)`, where `peek` returns the value at position `index` without dequeuing the item.

2.1.1 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. 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 an object-oriented language, one could implement a stream abstraction as a library. This avoids some of the problems associated with a procedural loop nest, but the programming model is complicated by efficiency concerns–to optimize cache performance, the entire application processes blocks of data that complicate and obscure the underlying algorithm.

In contrast to these alternatives, 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.

2.2 Connecting Filters

The basic construct for composing filters into a communicating network is a `Pipeline`, such as the FM Radio in Figure 1. Like a `Filter`, a `Pipeline` 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`.

There are two other stream constructors besides `Pipeline`: `SplitJoin` and `FeedbackLoop` (see Figure 2). From now on, we use the word *stream* to refer to any instance of a Filter, Pipeline, SplitJoin, or FeedbackLoop.

A SplitJoin is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. There are two kinds of splitters: 1) Duplicate, which replicates each data item and sends a copy to each parallel stream, and 2) RoundRobin($w_1, \ldots, w_n$), which sends the first w_1 items to the first stream, the next w_2 items to the second stream, and so on. RoundRobin is also the only type of joiner that we support; its function is analogous to a round robin splitter. If a RoundRobin is written without any weights, we assume that all $w_i = 1$.

The splitter and joiner type are specified with calls to `setSplitter` and `setJoiner`, respectively (see Figure 1); 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 RoundRobin can function as an exclusive selector if one or more of the weights are zero.

The last control construct provides a way to create cycles in the stream graph: the `FeedbackLoop`. It contains a joiner, a body stream, a splitter, and a loop stream, which are set with calls to `setJoiner`, `setBody`, `setSplitter`, and `setLoop`, respectively.

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`; 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 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 indenta-

[1]This design might seem unnatural, but it is necessary to allow inlining (Section 2.2) within a Java-based syntax.

tion level of the code. In our Java syntax, we make use of anonymous classes for inlining [4].

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

A final benefit of stream graph construction in StreamIt is the ability to parameterize graphs. For instance, the Equalizer in Figure 1 inputs a parameter `N` that controls the number of parallel streams that it contains. This further improves readability and decreases code size.

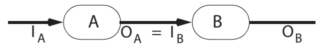
2.3 Messages

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. 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. Due to space constraints, we do not describe the syntax for message statements, but we do consider the semantics of message timing in Section *3.2.2*.

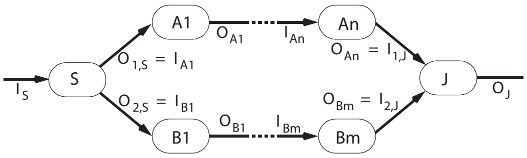
3. STREAMING MODEL OF COMPUTATION

In this section, we develop an abstract model of streaming computation to serve as a basis for reasoning about program transformations and compilation techniques within the streaming domain. A stream graph differs from a traditional, sequential program in that all of the filters of the graph are implicitly running in parallel, with the execution order constrained only by the availability of data on channels between the filters. Further, filters communicate only with their immediate neighbors, thereby removing any notion of global time or non-local dependences of one filter on another. These properties merit the development of a new model of computation, in which the notions of timing, scheduling, and dependence analysis are in terms that are relative to a given filter in the graph, instead of being global characteristics of a program.

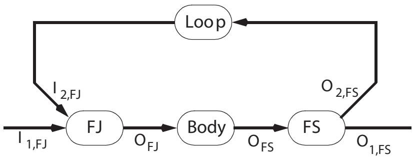
We will arrive at this notion of relative timing and dependence via a **stream dependence function**, SDEP, that is defined for a given stream graph. In Section 3.1 we provide a definition of SDEP along with some notation. We then motivate the SDEP function in 3.2 by deriving a concept of relative time and a meaning for StreamIt's messaging system. Only then, in Section 3.3, do we turn to deriving expressions for the SDEP function itself; Sections 3.4, 3.5 and 4 further employ the function in the respective contexts of program verification, denotational semantics, and program optimization.



(a) A Pipeline.



(b) A SplitJoin.



(c) A FeedbackLoop.

Figure 2: StreamIt structures with labeling.

3.1 Notation

We use the following notation:

- A *tape* is an infinite history of the values that have been pushed onto a channel between two filters. We use I_S and O_S to denote the input and output tapes of stream S, respectively, with numbering used to distinguish between multiple input or output tapes (see Figure 2). Finally, $n(T)$ represents the number of items on tape T at a given point of execution.

- We say that a filter A is *upstream* of filter B (or, equivalently, B is *downstream* of A) if there is a directed path in the stream graph from O_A to I_B. We use this terminology for tapes as well as filters.

- The number of items that are pushed, popped, and peeked by filter A during a single execution of its work function are denoted by $push_A$, pop_A, and $peek_A$, respectively. Note that $peek_A$ includes the items that are popped, such that $pop_A \leq peek_A$.

Now, we are ready to give a definition of the stream dependence function, SDEP:

DEFINITION 1. $\text{SDEP}_{b \to a}(x)$ *is the minimum number of items that must appear on tape a given that there are x items on tape b, where b is downstream of a.*

Thus, one can think of SDEP as an inter-filter data dependence mapping. Though the actual data references in a stream program appear from within the work functions, there is an aggregate dependence that restricts a filter from firing until it has enough items on its input tape to satisfy all its internal references (we assume that each firing is atomic). The SDEP function generalizes this dependence to answer a different question: how many items are needed on *another* filter's input tape before my filter can fire? The following sections will provide some additional intuition as to the meaning and applications of the SDEP function.

3.2 Information Flow

Above, the SDEP function is described in terms of data dependences. However, we can also think of this function as defining a common timing mechanism that asynchronous filters can use to synchronize events. We present this timing mechanism in terms of "information flow", which we believe is a central concept of the streaming domain.

3.2.1 Information Wavefronts

When an item enters a stream, it carries with it some new information. As execution progresses, this information cascades through the stream, affecting the state of filters and the values of new data items which are produced. We refer to an "information wavefront" as the set of filter executions that first sees the effects of a given input item. 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.

The SDEP function can be used to give a precise definition to an information wavefront. One interpretation of $y = \text{SDEP}_{b \to a}(x)$ is that the item at position y of tape a is the 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 needs to pass through a round-robin joiner before some data from another stream can be routed to tape b. This is why we choose "information flow" instead of "data flow" to describe the timing concept.

3.2.2 Message Timing

We can also use the SDEP function to give a precise meaning to the message delivery guarantees in StreamIt. Though we cannot give the details here due to space constraints (see [17] for a careful treatment), the general idea is as follows.

A filter A can send a message to filter B to communicate low-bandwidth, asynchronous data. To send a message, there needs to be an upstream or downstream path from A to B in the stream graph (the filters need not be directly connected.) The message statement appears in A's `work` function and includes a specified latency n that indicates "when" the target filter B should receive the message. The StreamIt language specification measures the latency n in terms of information wavefronts: if A is upstream of B, then B will receive the message immediately preceding the first invocation of its own `work` function which reads items that were affected by some output of the n'th invocation of A's `work` function. That is, the message handler in B is invoked when B sees the information wavefront that A sees in n execution steps.

In some cases, the ability to synchronize a message with an information wavefront can be very useful. For instance, if the input port of a hand-held computer detects a change in the networking protocol, it can send a reconfiguration message to all downstream filters with latency 0. This guarantees that each filter will reconfigure just in time to understand the new protocol, but will still process previous elements in the pipeline according to the old protocol.

3.3 The Stream Dependence Function

We now turn to deriving $\text{SDEP}_{b \to a}$ for all pairs of tapes a and b in a filter graph where a is upstream of b.

3.3.1 Filters

Let us derive $\text{SDEP}_{O_A \to I_A}(x)$, which represents the time shift across a single filter A. Since the filter produces $push_A$ items on every invocation, it must be invoked $\left\lceil \frac{x}{push_A} \right\rceil$ to produce the x'th item. On each invocation, it consumes pop_A items, and peeks at an additional $peek_A - pop_A$ items. Thus, the total number of items that must be present on the input is:

$$\text{SDEP}_{O_A \to I_A}(x) = \left\lceil \frac{x}{push_A} \right\rceil * pop_A + (peek_A - pop_A) \quad (1)$$

3.3.2 Pipelines

Let us now derive an expression for SDEP in the case of a pipeline. In the base case, consider that two filters are connected, with the output of A feeding into the input of B (see Figure 2). We are seeking $\text{SDEP}_{O_B \to I_A}(x)$: the minimum number of items that must appear on tape I_A given that there are x items on tape O_B. Observing that a minimum of $\text{SDEP}_{O_B \to I_B}(x)$ items must appear on tape I_B, and that I_B must equal O_A since the filters are connected, we see that a minimum of $\text{SDEP}_{O_A \to I_A}((\text{SDEP}_{O_B \to I_B})(x))$ items must appear on I_A. Using $\circ$ to denote function composition, we have:

$$\text{SDEP}_{O_B \to I_A} = \text{SDEP}_{O_A \to I_A} \circ \text{SDEP}_{O_B \to I_B}$$

By identical reasoning, this composition law holds for pipelined streams as well as filters. That is, a Pipeline of streams $S1 \ldots Sn$ has the following SDEP function:

$$\text{SDEP}_{Sn \to S1} = \text{SDEP}_{O_{S1} \to I_{S1}} \circ \cdots \circ \text{SDEP}_{O_{Sn} \to I_{Sn}} \quad (2)$$

One might be tempted to define the SDEP function for any pair of connected tapes as the composition of functions for the operators connecting those tapes. However, such a definition turns out to be problematic for the SplitJoin and FeedbackLoop constructs, which require a slightly different composition law for their components (as shown below). Instead, we can further extend our notation to include the *components* of streams that are connected in a pipeline. That is, if tapes t_i and t_j are contained within stream constructs S_i and S_j, respectively, and S_i and S_j belong to a pipeline of streams $S_1 \ldots S_n$, then:

$$\text{SDEP}_{t_j \to t_i} = \text{SDEP}_{O_{S_i} \to t_i} \circ \text{SDEP}_{O_{S_{i+1}} \to I_{S_{i+1}}} \circ \cdots \circ \text{SDEP}_{t_j \to I_{S_j}} \quad (3)$$

3.3.3 SplitJoins

We now derive SDEP expressions for the components of a SplitJoin, and for the SplitJoin construct as a whole. We denote the n output tapes of the splitter S by $O_{1,S} \ldots O_{n,S}$, and the n input tapes of the joiner J by $I_{1,J} \ldots I_{n,J}$ (see Figure 2).

Duplicate splitter. We consider the i'th output tape of an n-way duplicating splitter. Since the splitter duplicates each input item onto each output tape, there must be at least x items on I_S if there are x items on $O_{i,S}$. This yields a simple expression for SDEP:

$$\text{SDEP}_{O_{i,S} \to I_S}(x) = x$$

Round robin splitter. Let us consider an n-way splitter with weights $w_1 \dots w_n$. Observe that if there are $n(O_{n,S})$ items on the n'th output tape, then the splitter must have executed $\left\lfloor \frac{n(O_{n,S})}{w_n} \right\rfloor$ complete cycles in distributing items to the output tapes; each cycle draws $sum_i w_i$ items from the input tape I_S. Further, if there are $n(O_{i,S})$ items on the i'th output tape, then $n(O_{i,S}) mod w_i$ additional items have been deposited on $O_{i,S}$ during the current cycle of the splitter, and $n(O_{i,S})$ mod w_i + $sum_{j=0}^{i-1} w_j$ items have been drawn from the input since the last complete cycle. Summing the item count for the completed cycles and the current cycle gives the following expression for SDEP:

$$\text{SDEP}_{O_{i,S} \to I_S}(x) = \left\lfloor \frac{n(O_{n,S})}{w_n} \right\rfloor * \sum_i w_i + x \ mod \ w_i + \sum_{j=0}^{i-1} w_j$$

Round robin joiner. The reasoning is similar for an n-way joiner with weights $w_1 \dots w_n$. Let us use W to denote the sum of the weights: $W = \sum i = 1^n w_i$. If there are x items on the output tape O_J, then the joiner must have executed $\lfloor \frac{x}{W} \rfloor$ complete cycles, each of which drew w_i items from the i'th input tape. Since the last complete cycle, the joiner has drawn x mod W items from its inputs, and $MIN(0, x \ mod \ W - \sum_{j=0}^{i-1} w_j)$ of these items were taken from input tape j. Thus, the SDEP function from the output of the joiner to the i'th input tape is as follows:

$$\text{SDEP}_{O_J \to I_{i,J}}(x) = w_i * W + MIN(0, x \ mod \ W - \sum_{j=0}^{i-1} w_j)$$

SplitJoin construct. As with the Pipeline construct, we can derive the SDEP function across an entire SplitJoin as a composition of the component functions. However, a SplitJoin differs from a Pipeline in that the joiner imposes a control dependence between the parallel streams. That is, for there to be x items on the output of the joiner, there must be at least $\text{SDEP}_{O_J \to I_{i,J}}(x)$ items on *every* input tape $I_{i,J}$. Applying the composition law for pipelines (Equation 2), it follows that there must be at least at least $\text{SDEP}_{I_{i,J} \to O_{i,S}} \circ \text{SDEP}_{O_J \to I_{i,J}}(x)$ items on every output tape $O_{i,S}$ of the splitter. Finally, the minimum number of items appearing on the input tape I_S of the splitter is the *maximum* of the item requirement from any output tape $O_{i,S}$. By this reasoning, the SDEP function for a SplitJoin is as follows:

$$\text{SDEP}_{O_J \to I_S}(x) = \\ \underset{i \in [1,n]}{MAX}(\text{SDEP}_{O_{i,S} \to I_S} \circ \text{SDEP}_{I_{i,J} \to O_{i,S}} \circ \text{SDEP}_{O_J \to I_{i,J}})(x)$$

3.3.4 FeedbackLoops

The SDEP function for a feedback loop requires extra care. Although the feedback splitter FS serves as a normal splitter, with the same SDEP function as derived above, the feedback joiner FJ is slightly different due to the initialization phase of the loop. Also, the SDEP function does not compose across all components of the loop, since otherwise there would be conflicting definitions for paths that circle the loop several times.

Feedback joiner. For a feedback loop with delay d, the feedback joiner must fabricate its first d input values, since no items have yet been pushed onto the loop tape $I_{2,FJ}$. This means that there must be an offset of d in the SDEP function, since the first d items are direct inputs to the joiner instead of appearing as items on the input tape. Using J to denote a round robin joiner as considered above, we thus have the following expression for the SDEP function across the feedback path:

$$\text{SDEP}_{O_{FJ} \to I_{2,FJ}}(x) = \text{SDEP}_{O_J \to I_{2,J}}(x) - d$$

However, the SDEPfunction remains unchanged with respect to the input from the main stream:

$$\text{SDEP}_{O_{FJ} \to I_{1,FJ}}(x) = \text{SDEP}_{O_J \to I_{1,J}}(x)$$

Feedback components. Within a feedback loop, the SDEPfunction between tape a and any downstream tape b can be uniquely defined by composing the SDEPfunctions along the directed acyclic path between a and b. We require an acyclic path to avoid successive passes around the loop, which would prevent a unique definition of the function. Denoting this path of tapes by $(a, t_1, \dots, t_n, b)$, the composition follows the form of Equation 2:

$$\text{SDEP}_{b \to a}(x) = \text{SDEP}_{t_1 \to a} \circ \text{SDEP}_{t_2 \to t_1} \circ \dots \text{SDEP}_{b \to t_n}$$

Note that these functions can then be composed with those of constructs neighboring the feedback loop to obtain, for instance, the relation between the loop tape $I_{2,FJ}$ and a downstream pipeline (by application of Equation 3).

Feedback loop construct. As a special case of the equation above, we can see that the SDEPfunction for the feedback loop as a whole is the composition of the SDEPfunctions along the main path:

$$\text{SDEP}_{O_{FS} \to I_{1,FJ}}(x) = \text{SDEP}_{O_J \to I_{1,J}}(x) \circ \text{SDEP}_{O_J \to I_{1,J}}(x)$$

Intuitively, this is because–in any semantically correct stream program–the loop itself is guaranteed to have enough inputs to feed the joiner, such that the output tape of the feedback loop places a restriction only on the input tape of the feedback loop.

3.3.5 Summary

In the preceding sections, we have derived a SDEPfunction for the components of each stream construct, as well as for the stream construct as a whole. By application of Equation 3, this yields a function $\text{SDEP}_{b \to a}$ for every pair of tapes a and b where b is downstream of a.

3.4 Program Verification

A number of program analysis techniques are immediately afforded by the SDEPfunction. 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 every feedback loop produces enough data to satisfy its own feedback joiner. This can be formulated in terms of the SDEPfunction by considering $\text{SDEP}_{t \to t}$, the data that a tape t in a feedback loop requires of itself. However, since we didn't define SDEPacross circular paths in the stream graph, we will denote this function by LOOPDEPand define it at the loop input to the feedback joiner:

$$\text{LOOPDEP}(x) \equiv \text{SDEP}_{O_{FJ} \to I_{2,FJ}} \circ \text{SDEP}_{I_{2,FJ} \to O_{FJ}}$$

Now, the loop will be deadlock-free if and only if $\forall x \in \mathcal{N}, x - \text{LOOPDEP}(x) > 0$. This condition follows directly from causality–the x'th item can be produced if and only if its production depends only on some subset of the $x-1$ items that are already on the channel.

Overflow detection. There are two places that a buffer can grow to an unbounded size in the stream graph. The first is in a feedback loop, when[2] $x - \text{LOOPDEP}(x) = \omega(1)$. That is, if $\text{LOOPDEP}(x)$ items on the feedback tape enables the production of an additional $x - \text{LOOPDEP}(x)$ items that grows asymptotically with the position x on the tape, then the constant consumption rate will not keep up with the growing production rate, and the buffer will overflow.

The second case of buffer overflow is when the parallel streams of a SplitJoin have asymptotically different production rates. For a given stream i in a SplitJoin construct, the buffer corresponding to the joiner input tape $I_{i,J}$ will overflow if and only if there is a stream j in the SplitJoin for which:

$$(\text{SDEP}_{O_{i,S} \to I_S} \circ \text{SDEP}_{I_{i,J} \to O_{i,S}} \circ \text{SDEP}_{O_J \to I_{i,J}})(x) -$$
$$(\text{SDEP}_{O_{j,S} \to I_S} \circ \text{SDEP}_{I_{j,J} \to O_{j,S}} \circ \text{SDEP}_{O_J \to I_{j,J}})(x) = \omega(1)$$

Both of these cases could be detected by a compiler to verify that no buffers will overflow during steady-state execution.

3.5 Denotational Semantics

In this section, we develop a denotational semantics for obtaining the meaning of an entire stream graph. In Section 4, this semantics is used to show that a optimizing transformation on the stream graph preserves the meaning of the entire program.

Our denotational semantics contains three algebras: one for literal StreamIt syntax, one for an intermediate abstract syntax, and one for the semantic analysis. The purpose of the intermediate algebra is to provide a simplified syntax for developing stream transformations, and to abstract away the StreamIt-specific aspects of the program. We provide an informal description of how to translate back and forth between StreamIt programs and the abstract syntax, and then consider more formal valuation functions for determining the meaning of the abstract syntax within the semantic algebra. Throughout the analysis, we assume that filters are stateless and that the stream program is semantically correct.

3.5.1 Intermediate Algebra

The intermediate algebra provides a common mathematical representation for manipulating stream programs. Though we have referred to this algebra as providing an abstract syntax for stream programs, the representation is strictly a mathematical framework within semantic domains rather than a program that is fit for execution. Nonetheless, the LISP-like syntax allows us to think of the representation as a program that is amenable to straightforward transformation techniques.

The domains of the intermediate algebra are shown in Figures 3 and 4. The algebra represents a tape as an infinite

[2] $f(x) = \omega(g(x))$ if $lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty$

$$
\begin{array}{rcl}
 & & Item = \mathcal{R} \\
i & \in & Index = \mathcal{N} \\
g & \in & IndexTransform = Index \to Index \\
t & \in & Tape = Index \to Item \\
 & & Pop, Peek, Push = \mathcal{N} \\
f & \in & WorkStatement = IndexTransform \to (Tape \to Item) \\
 & & WorkFunction = WorkStatement^{+} \\
S & \in & SplitType = \{Duplicate, RoundRobin\} \\
J & \in & JoinType = \{RoundRobin\}
\end{array}
$$

Figure 3: Semantic domains that are shared between the intermediate and transform algebras.

$$
\begin{array}{l}
s \in Stream = Filter + Pipeline + SplitJoin + FeedbackLoop \\
\quad Filter = Push \times Pop \times Peek \times WorkFunction \\
\quad Pipeline = Stream^{+} \\
\quad SplitJoin = SplitType \times Stream^{+} \times JoinType \\
\quad InitFeedback = Int^{+} \\
\quad BodyStream, LoopStream = Stream \\
\quad FeedbackLoop = JoinType \times BodyStream \times \\
\qquad\qquad LoopStream \times SplitType \times InitFeedback
\end{array}
$$

Figure 4: Semantic domains specific to the intermediate algebra.

$$
\begin{array}{l}
TapeTransform = Tape \to Tape \\
StreamTransform = IndexTransform \to TapeTransform
\end{array}
$$

Figure 5: Semantic domains specific to the transform algebra.

mapping from indices to items. Generally, stream constructs are represented as lists of their component streams, and filters' work functions are encoded as lists of push statements that–given the transform from their local indexing to the global tape position–return a mapping from a tape to an output item.

Converting to the intermediate algebra. It is straightforward to generate an expression in the intermediate algebra that reflects the meaning of a given StreamIt program. Due to space limitations, we consider here only the translation of the work functions.

The translation of a filter's work function contains two steps. First, the function is arranged in a *canonical form*, in which each pushed item is given as a direct function of the peeked items, and all of the pop statements are at the end of the function. Let us consider a work function with I/O rates PUSH, POP, and PEEK. The canonical form of this work function gives us an element `w` of the syntactic domain `StreamItWorkFunction`:

```
w = void work() {
    output.push( (f_1 input.peek(0) ...  input.peek(PEEK-1) )
    ...
    output.push( (f_PUSH input.peek(0) ...  input.peek(PEEK-1) )
    for (int i=0; i<POP; i++) { input.pop(); }
}
```

Above, we model the computation of the work function as pure mathematical functions that can be injected into the semantic domain. To simplify our notation, we define the valuation $\mathcal{W}$: `StreamItWorkFunction` $\to$ *WorkFunction* in terms of `w`, the example syntactic work function from above. The valuation, then, is the alternate application of each push statement's function f, with the index expressions transformed from their local index x to a global index $g(x)$ on the input tape:

$$\mathcal{W}[\mathtt{w}] = [h_1 \ldots h_{push}]$$
$$where\ h_i = [\ \lambda\ g\ t\ .\ f_i(t(g(0)), \ldots, t(g((PEEK-1))))]$$

Converting from the intermediate algebra. To convert back to StreamIt, we can perform the inverse of the translation shown above, with a push statement for each function and a local index expression x in place of the global index $g(x)$. Common sub-expression elimination can be used to eliminate duplicate peek statements or shared portions of the f_i's.

3.5.2 Transform Algebra

The transform algebra is designed to express the meaning of a stream graph as a transformation from an input tape to an output tape. Its semantic domains are given in Figures 3 and 5. Our goal is to express the meaning of a $Stream$ in the intermediate algebra as a $TapeTransform$ in the transform algebra.

To do this, we introduce the $StreamTransform$ domain, each element of which maps an $IndexTransform$ (let us call it g) to a $TapeTransform$. The intuition is that g represents a relative indexing function that is imposed by an enclosing stream construct. For instance, in a two-way SplitJoin with a RoundRobin splitter, the SplitJoin construct imposes a $g = \lambda\ i\ .\ 2*i$ on the second parallel stream component. That is, index i on the component stream's input tape corresponds to index $g(i)$ on the input tape of the SplitJoin. Thus, if the component stream is transforming the input tape of the entire SplitJoin, it must apply g to its original index references.

Let us denote our valuation functions as $\mathcal{M} : Stream \rightarrow TapeTransform$ and $\mathcal{S} : Stream \rightarrow StreamTransform$. Then the meaning of a top-level stream s is as follows:

$$\mathcal{M}[s] = \mathcal{S}[s](\mathcal{I})$$
$$where\ \mathcal{I}\ denotes\ the\ identity\ function$$

That is, the meaning of an entire stream program is simply the $Stream\text{-}Transform$ for that program applied to the identity function as the $Index\text{-}Transform$, since at the top level there is no enclosing stream constructs and the the tape transformation is relative to the input tape of the stream itself. We now turn our attention to deriving $\mathcal{S}$ for Filters, Pipelines, and SplitJoins. Letting % denote the *mod* function, for a filter we have that:

$$\mathcal{S}[(\text{in Filter}\ push\ pop\ peek\ (h_1\ \ldots\ h_{push}))] = \lambda\ g\ t\ i\ .$$
$$(h_{i\ \%\ push})(\lambda\ i_{local}\ .\ g((\text{SDEP}_{O_F \rightarrow I_F}(i) - peek + 1 + i_{local})))(t)$$

That is, the value that a filter pushes onto the i'th position of its output tape is calculated with its function at index $i\ \%\ push$. By the definition of SDEP, the index offset to the last value the filter peeks is $\text{SDEP}_{O_F \rightarrow I_F}(i)$, where I_F and O_F denote the input and output tapes of the filter (as shown in Equation 1, this is a pure function of $push$, pop, and $peek$). Thus, the offset to the first value the filter peeks is $\text{SDEP}_{O_F \rightarrow I_F}(i) - peek + 1$, and we obtain the global index by adding this offset to the local index i_{local}.

For a pipeline, the transform function is simply the composition of the transforms of component streams. At the internal connections of the pipeline, the index transform is the identity function, but at the start of the pipeline we apply the transform g to interface the pipeline to its outside connection.

$$\mathcal{S}[(\text{in Pipeline}\ s_1\ s_2\ \ldots\ s_n)] =$$
$$\lambda\ g\ .\ (\mathcal{S}[s_n](\mathcal{I}) \circ \cdots \circ \mathcal{S}[s_2](\mathcal{I}) \circ \mathcal{S}[s_1](g))$$
$$where\ \mathcal{I}\ denotes\ the\ identity\ function$$

The valuation function for a SplitJoin follows the same idea, but the notation is slightly heavier. Given that we have a round robin joiner with weights $w_1 \ldots w_n$ and $W = \sum w$, we first represent the parallel stream $p(i)$ which computes the i'th output of the joiner:

$$p(i) = MIN(j\ s.t.\ \sum_{k=0}^{j-1} w_i \leq i\ mod\ W) \tag{4}$$

Now, the i'th tape position assumes the value that is produced along stream $p(i)$ in the SplitJoin, and the value of interest appears at position $\text{SDEP}_{O_J \rightarrow I_{p(i),J}}(i)$ on the output tape of stream $p(i)$. The indexing function transforms the stream's local index i_{local} for its own input tape to the corresponding index $\text{SDEP}_{O_{p(i),S} \rightarrow I_S}(i_{local})$ for the input tape of the splitter:

$$\mathcal{S}[(\text{in SplitJoin}\ S\ s_1\ s_2\ \ldots\ s_n\ J)] = \lambda\ g\ t\ i\ .$$
$$((\mathcal{S}[s_{p(i)}])(\lambda\ i_{local}\ .\ g(\text{SDEP}_{O_{p(i),S} \rightarrow I_S}(i_{local}))))(\text{SDEP}_{O_J \rightarrow I_{p(i),J}}(i))$$

This completes our description of the transform algebra, as we have not yet formulated the valuation function for FeedbackLoops. Given the valuation functions above, however, we express the meaning of any combination of Pipelines and SplitJoins as a mathematical transformation between infinite tapes. We will utilize this formulation to prove that certain transformations of the stream graph preserve the meaning of the program.

4. OPTIMIZATION

We now turn our attention to the problem of optimizing a stream program. Unlike other program domains, where the principle aim of compiler optimization is to shorten the total execution time, there are many distinct optimization metrics for streaming applications, including throughput, latency, data size, and code size. The latter two of these are especially important in embedded domains, where memory is in short supply; latency can be critical for real-time applications, and throughput is always of interest.

In this section we present some transformations that improve a stream program by one or more of these metrics. However, there is often a tradeoff between throughput and latency, or code size and data size, such that the optimality of a stream program depends on the metric of interest.

4.1 Fusion Transformations

A primary stream optimization is the fusion of multiple filters and streams into a single atomic unit. This can be beneficial for throughput, latency, and data size, as data buffers are eliminated in favor of local variables with short live ranges. Fusion is also important for adapting a fine-grained stream program to a coarse-grained target; the programmer benefits from dividing the program into many modular components without losing the performance of a single, integrated procedure.

An algorithm for fusing a pipeline of two filters that contain only `push` and `pop` statements is given in [12]. However, in a stream program, it pays to consider not only vertical fusion of pipeline constructs, but also horizontal fusion of parallel streams in a SplitJoin. Here we present a transformation on the abstract syntax of Section *3.5.1* that collapses a SplitJoin construct containing n parallel filters $s1 \ldots sn$ into a single filter sc. Let us denote the weights of the joiner J by $w_1 \ldots w_n$ with $W = \sum_{i=1}^{n} w_i$:

$$
\begin{aligned}
&Merge[(S\ s1\ \ldots\ sn\ J)] \\
&\quad = (\text{in Filter } push_{sc}\ pop_{sc}\ peek_{sc}\ work_{sc})
\end{aligned}
$$

$$
\begin{aligned}
where: \mathbf{push_{sc}} &= totalRounds * W \\
\mathbf{pop_{sc}} &= totalPop \qquad if\ S\ =\ RoundRobin \\
&= totalPop/n \quad\ if\ S\ =\ Duplicate \\
\mathbf{peek_{sc}} &= MAX_{j \in [1, push_{sc}]}(shift_j(peek_{sj})) \\
\mathbf{work_{sc}} &= h_{sc,1} \ldots h_{sc,push_{sc}} \\
totalRounds &= lcm(lcm(push_{s1}, w_1), \ldots, lcm(push_{sn}, w_n)) \\
totalPop &= \sum_{i=1}^{n} (totalRounds * w_i * pop_{si}/push_{si}) \\
shift_j(x) &= \mathrm{SDEP}_{I_{sj} \rightarrow I_S}(x + \\
&\qquad (\mathrm{SDEP}_{O_{sj} \rightarrow I_{sj}} \circ \mathrm{SDEP}_{O_J \rightarrow O_{sj}})(j) - peek_{sj}) \\
h_{sc,j} &= \lambda\ g\ .\ h_{s,p(j)}(\lambda\ i_{local}\ .\ shift_j(g(i_{local})))
\end{aligned}
$$

We have proven that this transformation preserves the meaning of the program with respect to our transform algebra for the case when $n = 2$, $w_1 = w_2 = 1$, and S is a duplicate splitter. The proof involves only straightforward algebra, but we omit it due to space constraints.

This transformation is very powerful–it allows us to fuse any set of parallel filters in a SplitJoin construct into a single filter, regardless of the splitter/joiner types and the push/pop/peek requirements. We have implemented this transformation in the StreamIt compiler for cases with a duplicate splitter and filters with output rates matching the joiner's weights; performance improves significantly (see Section 6) due to decreased channel operations.

In the sections that follow, we give an overview of other optimizations that we are implementing in the StreamIt compiler. Due to space limitations, we cannot describe them at the above level of detail.

4.2 Fission Transformations

When the machine target is more fine-grained than the stream graph, it is advantageous to break filters up into smaller pieces so that more hardware resources can be utilized. We propose three fission transformations:

1. **Parallelizing stateless filters.** If a filter has no state, then we can gain data parallelism by duplicating the filter n times and embedding it in an n-way SplitJoin with a round robin splitter and joiner.

2. **Parallelizing stateless feedback loops.** If the body of a feedback loop is stateless and its input/output rates evenly divide the delay of the loop, then the entire loop can be replicated and parallelized as in (1), with the quantity and delay of the new loops being (approximately) equal to the quotient of the old delay and the body stream's I/O rates. This exploits the fact that for certain feedback loops there are interleaved subsequences of the input stream that are transformed completely independently by the loop.

3. **Splitting stateful filters.** If a filter has persistent state, we can still gain pipeline parallelism by breaking the the filter into an n-stage pipeline in which the state is communicated through the data channels.

4.3 Steady-State Invariant Code Motion

In the streaming domain, the analog of loop-invariant code motion is the motion of code from the steady-state `work` function to the `init` function if it does not depend on any quantity that is changing during the steady-state execution of a filter. Quantities that the compiler detects to be constant during the execution of `work` can be assigned to fields in the `init` function and then referenced from `work`.

4.4 Induction Variable Detection

The `work` function can also be analyzed as would the body of a loop to see if there are induction variables from one steady-state execution to the next. This analysis is useful both for strength reduction, which *adds* a dependence between invocations by converting an expensive operation to a cheaper, incremental one, as well as for data parallelization, which *removes* a dependence from one invocation to the next by changing incremental operations on filter state to equivalent operations on privatized variables.

4.5 Decimation Propagation

Decimation refers to the regular discarding of a fraction of a filter's input items, perhaps to reduce the sampling rate in a stream. In the streaming domain, the analog of dead code elimination is the propagation of this decimation up through the stream graph, thereby eliminating the computations that produce the unused items.

4.6 Synchronization Removal

In a StreamIt graph, the SplitJoin construct provides a way to define independent units of parallel computation. However, when two SplitJoins s_1 and s_2 are connected in a pipeline, there is a joiner/splitter pair that serializes all of the items passing from s_1 to s_2. If the joiner of s_1 and the splitter of s_2 are both round robins with equal weights, then this node can be eliminated in favor of a single SplitJoin s_c with the i'th parallel stream in s_c being a pipeline of the corresponding streams in s_1 and s_2.

5. SCHEDULING

The tradeoffs between different optimization criteria are particularly pronounced in the scheduling stage of a streaming compiler. As shown in Figure 6, at the extreme ends of the optimization space are schedules which minimize code size (at the expense of latency and buffer size) and which minimize buffer size and latency (at the expense of code size). We give an overview of this scheduling space, and present a new phased scheduling technique that takes advantage of the structured streams in StreamIt to obtain a minimum latency schedule without a large increase in code size.

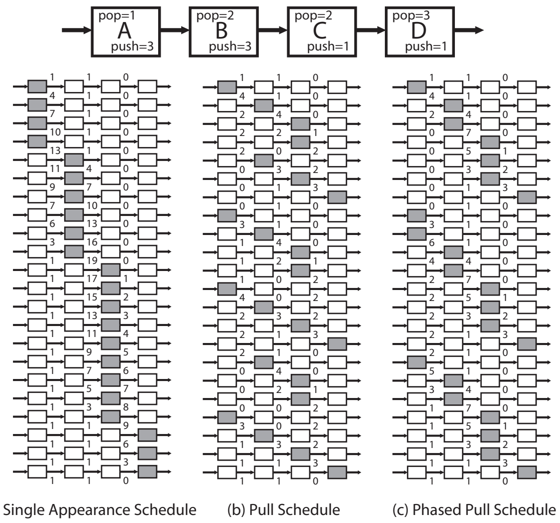


Figure 6: The three different scheduling schemes. The channels are labeled with the number of live data items they contain.

5.1 Initialization vs. Steady State

Firstly, one must note that StreamIt programs can require a separate schedule for initialization and for the steady state. The steady state schedule must be periodic–that is, its execution must preserve the number of live items on each channel in the graph. We need a separate initialization schedule if there is a filter with $peek > pop$, since no periodic schedule could eliminate all of the live items on the filter's input channel (which would be needed to return the graph to its initial configuration). In the StreamIt compiler, this initialization schedule is constructed via symbolic execution of the stream graph, until each filter has $peek - pop$ items on its input channel.

For graphs without peeking, one can find a unique and minimal set of multiplicities for a periodic schedule, and all other periodic schedules will be a multiple of these [2]. Thus, the challenge in scheduling is to impart an *order* on the steady state execution set so that a given metric is optimized. In what follows, we consider three approaches to this problem.

5.2 Minimizing Code Size

A schedule with minimal code size is a Single Appearance Schedule (SAS): one where each node appears exactly once in the loop nest denoting the schedule (e.g., (4A)(6B)(9C)(3D) in Figure 6). There has been a lot of attention (e.g., [2]) on SAS's because their minimal code size allows extensive function inlining, which enables compiler optimizations and improves performance. In the StreamIt compiler, we compute a simple SAS with hierarchical ordering according to the original stream structure. The problem with this and other SAS's is that the data buffer size can grow quite large, which motivates other techniques. Moreover, the inlining benefits afforded by SAS's are less important in StreamIt, where the compiler itself can consider inter-procedural optimizations.

5.3 Minimizing Buffer Size

On the other end of the spectrum, one can minimize buffer size by implementing a "pull schedule", in which filters are executed in demand-driven order to fire the output node of the stream. A pull schedule guarantees the minimal static buffer size (assuming each filter has its own input buffer), with each channel not exceeding

$$\left\lceil \frac{peek_B}{\gcd(push_A, pop_B)} - 1 \right\rceil \gcd(push_A, pop_B) + push_A$$

However, a pull schedule is very irregular, and could require an exponential number of instructions to encode.

5.4 Minimizing Latency

The pull schedule also minimizes the average latency of the stream, which could be important for real-time applications. We define the latency of an output item as the number of work functions that were executed within the stream before the item was output; the stream's average latency is taken over all of its output items. While the pull schedule is sufficient to minimize latency, it is possible to factor more of the schedule into shared loop nests. For this we present the notion of a "phased schedule".

5.5 Phased Schedules

We invented phased schedules–which rely heavily on the structured streams of StreamIt–to achieve a minimum-latency schedule without risking the code explosion of a pull schedule (see Figure 6). A phase is a (possibly non-periodic) schedule for a stream structure in which the bottom-most filter in that structure fires exactly once. There could be several phases for a given stream component, and each phase has an associated push, pop, and peek count. In the base case, a filter has just one phase with its own push, pop, and peek. For stream constructs, the list of phases is determined by simulating a "phased pull"–that is, just like a pull schedule, except that child streams must execute in steps of their own phases.

Due to space limitations, we cannot give a more detailed description of the phased scheduling algorithm. However, it is the case that phased schedules have minimum latency because they invoke the same set of filters as the pull model for a given output item; only the *ordering* of those filter executions can be rearranged to improve the code size.

5.6 Respecting Message Constraints

Another responsibility of the scheduler in StreamIt is to satisfy the message delivery guarantees. Each downstream message with a negative latency imposes a lower bound on the buffer size between the source and target filter. Likewise, an upstream message with a positive latency imposes an upper bound on this buffer size.

6. IMPLEMENTATION AND EVALUATION

We have implemented a fully-functional prototype of the StreamIt optimizing compiler as an extension to the Kopi Java Compiler, a component of the open-source Kopi Project [18]. Our compiler generates 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

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 1: Application Characteristics

	StreamIt		Hand Coded	
Benchmark	Baseline	Fusion	Spectra	C
PCA Demo	1.3	-	3.4	N/A
FM Radio	5.8	4.9	9.9	N/A
perftest4	330	-	330	N/A
GSM Decoder	4.88	-	N/A	.47

Table 2: Performance Results (in μsec/output)

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. The interface relies on the SDEP function to specify message timing in terms of a stream's input tape.

We have evaluated our compiler with StreamIt versions of the following applications: 1) A GSM Decoder, which takes GSM-encoded parameters as inputs, and uses these to synthesize audible speech, 2) A system from the Polymorphic Computing Architecture (PCA) [8] 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) [16]. Table 1 gives characteristics of the above applications including the number of filters implemented and the size of the stream graph as coded.

In the Table 2, we evaluate 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 [16] 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 *2.1.1*), 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 the high-level knowledge of the algorithm, and the StreamIt performs an order of magnitude slower.

The StreamIt compiler infrastructure is far from complete. We are in the process of discovering all the optimization possibilities in this new domain. 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 strongly believe that we can improve the current performance by at least an order of magnitude on uniprocessors, and we have yet to take advantage of the inherent data and pipeline parallelism in StreamIt programs for parallel execution.

7. RELATED WORK

A large number of programming languages have included a concept of a stream, with various semantic formalisms; see [15] 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 [1] 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, these languages lack the structured streams of StreamIt, which enable a suite of hierarchical compiler optimizations and a clean semantics for verifying program transformations.

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 [9]. 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 have been many efforts to minimize their memory requirements [2, 11, 5, 3]. However, nodes such as round robins that have a cyclic pattern of I/O rates fall outside of SDF and within the Cyclo-Static domain [7] where there are fewer scheduling results. To the best of our knowledge, the phased scheduling algorithm for minimal latency is novel.

8. CONCLUSION

We have implemented a prototype optimizing compiler for StreamIt: a high-level stream language that aims to raise the abstraction level of stream programming without sacrificing performance. We have demonstrated that the hierarchical structure imposed by the language enables new compiler analyses and optimizations for the streaming domain. In particular, we believe that the stream dependence function is a critical compiler representation for streaming applications, comparable to distance and direction vectors for scientific applications.

In all, we believe that optimizing compilers will be of immense importance in the streaming domain. Though our compiler cannot yet match the performance of hand-coded applications, there is a ripe field of optimizations that are enabled by the structured nature of the stream programming model. Moreover, it is a young domain where languages and tools are lacking, but performance is very critical; the developers that we have interacted with have been very eager to explore new language and compiler solutions. In an age when many are skeptical of the utility of traditional compiler optimization, we hope that the streaming domain proves to be an important frontier.

9. REFERENCES

[1] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

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

[3] S. Goddard and K. Jeffay. Managing Memory

Requirements in the Synthesis of Real-Time Systems from Processing Graphs. pages 59–70, 1998.

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

[5] R. Govindarajan, G. Gao, and P. Desai. Minimizing Memory Requirements in Rate-Optimal Schedules. Proc. of the Intl. Conf. on Application Specific Array Processors, San Francisco, 1994.

[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] R. Lauwereins, P. Wauters, M. Ade, and J. Peperstraete. Geometric parallelism and cyclo-static data flow in Grape-II, 1994.

[8] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Massachusetts Institute of Technology, August 2001.

[9] 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.

[10] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture, 2000.

[11] P. K. Murthy and S. S. Bhattacharyya. Shared Buffer Implementations of Signal Processing Systems using Lifetime Analysis Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, February 2001.

[12] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *Symposium on Principles of Programming Languages*, pages 119–130, 1996.

[13] 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 *HPCA*, Dallas, TX, November 1998.

[14] K. Sankaralingam, R. Nagarajan, S. Keckler, and D. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. The University of Texas at Austin, Department of Computer Sciences Technical Report TR-01-02, 2001.

[15] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[16] D. Tennenhouse and V. Bose. The SpectrumWare Approach to Wireless Signal Processing. Wireless Networks, 1999.

[17] B. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. MIT-LCS Technical Memo LCS-TM-620, Cambridge, MA, 2001.

[18] A.-G. L. Vincent Gay-Para, Thomas Graf and E. Wais. Kopi Reference manual. http://www.dms.at/kopi/docs/kopi.html, 2001.

[19] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to Software: The Raw Machine. MIT/LCS Technical Report TR-709, Cambridge, MA, 1997.