Teleport Messaging for Distributed Stream Programs[†]

William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe
{thies, karczma, janiss, rabbah, saman}@csail.mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

ABSTRACT

In this paper, we develop a new language construct to address one of the pitfalls of parallel programming: precise handling of events across parallel components. The construct, termed *teleport messaging*, uses data dependences between components to provide a common notion of time in a parallel system. Our work is done in the context of the Synchronous Dataflow (SDF) model, in which computation is expressed as a graph of independent components (or *actors*) that communicate in regular patterns over data channels. We leverage the static properties of SDF to compute a stream dependence function, SDEP, that compactly describes the ordering constraints between actor executions.

Teleport messaging utilizes SDEP to provide powerful and precise event handling. For example, an actor A can specify that an event should be processed by a downstream actor B as soon as B sees the "effects" of the current execution of A. We argue that teleport messaging improves readability and robustness over existing practices. We have implemented messaging as part of the StreamIt compiler, with a backend for a cluster of workstations. As teleport messaging exposes optimization opportunities to the compiler, it also results in a 49% performance improvement for a software radio benchmark.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages; Dataflow languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design, Performance

Keywords

StreamIt, Synchronous Dataflow, Event Handling, Dependence Analysis, Digital Signal Processing, Embedded

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Algorithms that operate on streams of data are becoming increasingly pervasive across a broad range of applications, and there is evidence that streaming media applications already consume a substantial fraction of the computation cycles on consumer machines [6, 7, 18, 23]. Examples of streaming workloads can be found in embedded systems (e.g., sensor nets and mobile phones), as well as in desktop machines (e.g., networking and multimedia) and high-performance servers (e.g., HDTV editing consoles and hyper-spectral imaging).

As high performance remains a critical factor for many streaming applications, programmers are often forced to sacrifice readability, robustness, and maintainability of their code for the sake of optimization. One notoriously difficult aspect of stream programming, from both a performance and programmability standpoint, is reconciling regular streaming dataflow with irregular control messages. While the high-bandwidth flow of data is very predictable, realistic applications also include unpredictable, low-bandwidth control messages for adjusting system parameters (e.g., filtering coefficients, frame size, compression ratio, network protocol, etc.). Control messages often have strict timing constraints that are difficult to reason about on parallel systems.

For example, consider a frequency hopping radio (FHR), which mirrors how CDMA-based cell phone technology works. In FHR, a transmitter and a receiver switch between a set of known radio frequencies, and they do so in synchrony with respect to a stream boundary. That is, a receiver must switch its frequency at an exact point in the stream (as indicated by the transmitter) in order to follow the incoming signal. Such a receiver is challenging to implement in a distributed environment because different processors might be responsible for the radio frontend and the frequency hop detection. When a hop is detected, the detector must send a message to the frontend that is timed precisely with respect to the data stream, even though the two components are running on different processors with independent clocks.

Other instances of control messaging have a similar flavor. A component in a communications frontend might detect an invalid checksum for a packet, and send a precisely-timed message downstream to invalidate the effects of what has been processed. Or, a downstream component might detect a high signal-to-noise ratio and send a message to the frontend to increase the amplification. In an adaptive beamformer, a set of filtering coefficients is periodically updated to focus the amplification in the direction of a moving target. Additional examples include: periodic channel characteriza-

[†]This is a revised version of the paper, released in September 2006. Compared to the conference version, it simplifies the timing of upstream messages. Changes are limited to Section 4.

tion; initiating a handoff (e.g., to a new network protocol); marking the end of a large data segment; and responding to user inputs, environmental stimuli, or runtime exceptions.

There are two common implementation strategies for control messages using today's languages and compilers. First, the message can be embedded in the high-bandwidth data stream, perhaps as an extra field in a data structure. Application components check for the presence of messages on every iteration, processing any that are found. This scheme offers precise timing across distributed components, as the control message has a well-defined position with respect to the data stream. However, the timing is inflexible: it is impossible for the sender to synchronize the message delivery with a data item that has already been sent, or to send messages upstream, against the flow of data. In addition, this approach adds complexity and runtime overhead to the steady-state data processing, and it requires a direct high-bandwidth connection between sender and receiver.

A second implementation strategy is to perform control messaging "out-of-band", via a new low-bandwidth connection or a remote procedure call. While this avoids the complexity of embedding messages in a high-bandwidth data stream, it falls short in terms of timing guarantees. In a distributed environment, each processor has its own clock and is making independent progress on its part of the application. The only common notion of time between processors is the data stream itself. Though extra synchronization can be imposed to keep processors in check, such synchronization is costly and can needlessly suppress parallelism. Also, the presence of dynamic messaging can invalidate other optimizations which rely on static communication patterns.

This paper presents a new language construct and supporting compiler analysis that allows the programmer to declaratively specify control messages. Termed "teleport messaging", this feature offers the simplicity of a method call while maintaining the precision of embedding messages in the data stream. The idea is to treat control messages as an asynchronous method call with no return value. When the sender calls the method, it has the semantics of embedding a placeholder in the sender's output stream. The method is invoked in the receiver when the receiver would have processed the placeholder. We generalize this concept to allow messages both upstream and downstream, and with variable latency. By exposing the true communication pattern to the compiler, the message can be delivered using whatever mechanism is appropriate for a given architecture. The declarative mechanism also enables the compiler to parallelize and reorder application components so long as it delivers messages on time.

Our formulation of teleport messaging relies on a restricted model of computation known as Synchronous Dataflow, or SDF [20]. As described in Section 1.1, SDF expresses computation as a graph of communicating components, or *actors*. A critical property of SDF is that the input and output rate of each actor is known at compile time. Using this property, we can compute the dependences between actors and automatically calculate when a message should be delivered. We develop a stream dependence function, SDEP, that provides an exact, complete, and compact representation of this dependence information; we use SDEP to specify the semantics of teleport messaging.

Teleport messaging is implemented as part of the StreamIt compiler infrastructure [25]. The implementation computes SDEP information and automatically targets a cluster of workstations. Based on a case study of a frequency hopping radio, we demonstrate a 49% performance improvement due to communication benefits of teleport messaging. As described in Section 4, our implementation limits certain sender-receiver pairs to be in distinct portions of the stream graph; if overlapping messages are sent with conflicting latencies, it may be impossible to schedule the delivery. This constrained scheduling problem is an interesting topic for future work.

This paper is organized as follows. In the rest of this section, we describe our model of computation and give a concrete example of teleport messaging. Section 2 defines the stream dependence function, and Section 3 shows how to calculate it efficiently. Section 4 gives the semantics for teleport messaging, and Section 5 describes our case study and implementation results. Related work appears in Section 6, while conclusions and future work appear in Section 7.

1.1 Model of Computation

Our model of computation is Cyclo-Static Dataflow (CSDF), a generalization [3] of Synchronous Dataflow, or SDF [20]. SDF and its variants are well suited for signal processing applications. Computation is represented as a graph of *actors* connected by FIFO communication channels. In CSDF, each actor follows a set of execution steps, or phases. Each phase consumes a fixed number of items from each input channel and produces a fixed number of items onto each output channel. The number and ordering of phases is known at compile time, and their execution is cyclic (that is, after executing the last phase, the first phase is executed again). If each actor has only one phase, then CSDF is equivalent to SDF. These models are appealing because the fixed input and output rates make the stream graph amenable to static scheduling and optimization [20].

In this paper, we use the StreamIt programming language [25] to describe the connectivity of the dataflow graph as well as the internal functions of each actor. Our technique is general and should apply equally well to other languages and systems based on Synchronous or Cyclo-Static Dataflow. In StreamIt, each actor (called a *filter* in the language) has one input channel and one output channel. An execution step consists of a call to the "work function", which contains general-purpose code. During each invocation, an actor consumes (*pops*) a fixed number of items from the input channel and produces (*pushes*) a fixed number of items on the output channel. It can also *peek* at input items without consuming them from the channel.

Actors are assembled into single-input, single-output stream graphs (or *streams*) using three hierarchical primitives. A *pipeline* arranges a set of streams in sequence, with the output of one stream connected to the input of the next. A *splitjoin* arranges streams in parallel; incoming data can either be duplicated to all streams, or distributed using a round-robin splitter. Likewise, outputs of the parallel streams are serialized using a round-robin joiner. Round-robin splitters (resp. joiners) execute in multiple phases: the ith phase pushes (resp. pops) a known number of items k_i to (resp. from) the ith stream in the splitjoin. Finally, a *feedbackloop* can be used to introduce cycles in the graph.

1.2 Illustrating Example

Figure 1 illustrates a StreamIt version of an FIR (Finite Impulse Response) filter. A common component of digital

```
 1  struct Packet {
 2    float sum;
 3    float val;
 4  }
 5
 6  void->void pipeline FIR {
 7    int N = 64;
 8
 9    add Source(N);
10    for (int i=0; i<N; i++)
11      add Multiply(i);
12    add Printer();
13  }
14
15  void->Packet filter Source(int N) {
16    work push 1 {
17      Packet p;
18      p.sum = 0;
19      p.val = readNewData();
20      push(p);
21    }
22  }
23
24  Packet->Packet filter Multiply(int i,
25                                 int N) {
26    float W = initWeight(i, N);
27    Packet last;
28
29    work pop 1 push 1 {
30      Packet in = pop();
31      last.sum = in.sum + last.val * W;
32      push(last);
33      last = in;
34    }
35  }
36
37  Packet->void filter Printer {
38    work pop 1 { print(pop().sum); }
39  }
```

Figure 1: FIR code.

```
 1    struct Packet<N> {
 2  *   boolean newWeights;
 3  *   float[N] weights;
 4      float sum;
 5      float val;
 6    }
 7
 8    void->void pipeline FIR {
 9      int N = 64;
10
11      add Source(N);
12      for (int i=0; i<N; i++)
13        add Multiply(i, N);
14      add Printer();
15    }
16
17    void->Packet<N> filter Source(int N) {
18      work push 1 {
19        Packet p;
20        p.sum = 0;
21        p.val = readNewData();
22
23  *     if (newConditions()) {
24  *       p.newWeights = true;
25  *       p.weights = calcWeights();
26  *     } else {
27  *       p.newWeights = false;
28  *     }
29
30        push(p);
31      }
32    }
33
34    Packet<N>->
35    Packet<N> filter Multiply(int i, int N) {
36      float W = initWeight(i, N);
37      Packet<N> last;
38
39      work pop 1 push 1 {
40        Packet<N> in = pop();
41  *     if (in.newWeights) {
42  *       W = in.weights[i];
43  *     }
44        last.sum = in.sum + last.val * W;
45        push(last);
46        last = in;
47      }
48    }
49
50    Packet<N>->void filter Printer {
51      work pop 1 { print(pop().sum); }
52    }
```

Figure 2: FIR code with manual event handling. Modified lines are marked with an asterisk.

```
 1    struct Packet {
 2      float sum;
 3      float val;
 4    }
 5
 6    void->void pipeline FIR {
 7      int N = 64;
 8  *   portal<Multiply> teleport;
 9
10  *   add Source(N, teleport);
11      for (int i=0; i<N; i++)
12  *     add Multiply(i, N) to teleport;
13      add Printer();
14    }
15
16    void->Packet filter
17    Source(int N, portal<Multiply> teleport) {
18      work push 1 {
19        Packet p;
20        p.sum = 0;
21        p.val = readNewData();
22        push(p);
23
24  *     if (newConditions())
25  *       teleport.setWeights(calcWeights());
26  *   }
27    }
28
29    Packet->Packet filter Multiply(int i, int N) {
30      float W = initWeight(i, N);
31      Packet last;
32
33      work pop 1 push 1 {
34        Packet in = pop();
35        last.sum = in.sum + last.val * W;
36        push(last);
37        last = in;
38      }
39
40  *   handler setWeights(float[N] weights) {
41  *     W = weights[i]
42  *   }
43    }
44
45    Packet->void filter Printer {
46      work pop 1 { print(pop().sum); }
47    }
```

Figure 3: FIR code with teleport messaging. Modified lines are marked with an asterisk.

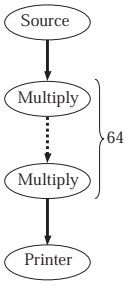

Figure 4: FIR stream graph.

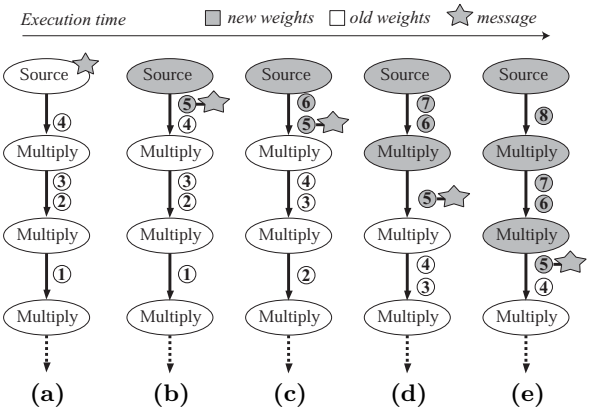

Figure 5: Execution snapshots illustrating manual embedding of control messages in FIR. Channels are annotated with data items present on one possible execution; items are numbered in order of production. (a) Source initiates change of weights, (b) weights are attached to data item #5 and embedded in stream, (c)-(e), actors check each input item, adjusting their own weight when they find a tagged item.

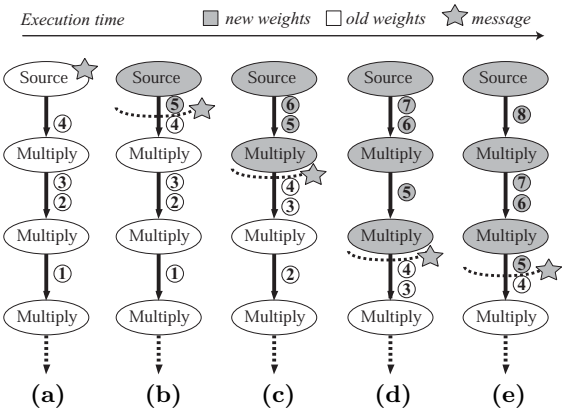

Figure 6: Execution snapshots illustrating teleport messaging in FIR. Channels are annotated with data items present on one possible execution; items are numbered in order of production. (a) Source calls a message handler, passing new weights as argument, (b) message boundary is maintained by compiler, (c)-(e), message handler is automatically invoked in actors immediately before the arrival of affected items.

signal processing applications, FIR filters represent sliding window computations in which a set of coefficients is convolved with the input data. This FIR implementation is very fine-grained; as depicted in Figure 4, the stream graph consists of a single pipeline with a `Source`, a `Printer`, and 64 `Multiply` stages—each of which contains a single coefficient (or *weight*) of the FIR filter. Each `Multiply` actor inputs a `Packet` consisting of an input item and a partial sum; the actor increments the sum by the product of a weight and the *previous* input to the actor. Delaying the inputs by one step ensures that each actor adds a different input to the sum. While we typically advocate a more coarse-grained implementation of FIR filters, this formulation is simple to parallelize (each actor is mapped to a separate processor) and provides a simple illustration of our analysis.

The problem addressed by this paper is as follows. Suppose that the actors in FIR are running in parallel and the `Source` detects that the weights should be adjusted (e.g., to suite the current operating conditions). Further, to guarantee stability, every output from the system must be obtained using either the old weights or the new ones, but not a mixture of the two. This constraint precludes updating all of the weights at the same instant, as the partial sums within the pipeline would retain evidence of the old weights. Rather, the weights must be changed one actor at a time, mirroring the flow of data through the pipeline. What is a simple and efficient way to implement this behavior?

One way to implement this functionality is by manually tagging each data item with a flag, indicating whether or not it marks the transition to a new set of weights. If it does, then the new set of weights is included with the item itself. While this strategy (shown in Figures 2 and 5) is functional, it complicates the `Packet` structure with two additional fields—a `newWeights` flag and a `weights` array—the latter of which is meaningful only when `newWeights` is true. This scheme muddles steady-state dataflow with event handling by checking the flag on every invocation of `Multiply` (line 41 of Figure 2). It is also very inefficient in StreamIt because arrays are passed by value; though it might be possible to compress each `Packet` when the `weights` field is unused, this would require an aggressive compiler analysis and would also jeopardize other optimizations by introducing an unanalyzable communication rate in the stream graph.

This paper proposes an alternate solution: teleport messaging. The idea behind teleport messaging is for the `Source` to change the weights via an asynchronous method call, where method invocations in the target actors are timed relative to the flow of data in the stream. As shown in Figure 3, the `Multiply` actor declares a message handler that adjusts its own weight (lines 40-42). The `Source` actor calls this handler through a *portal* (line 25), which provides a clean interface for messaging (see Section 4). As depicted in Figure 6, teleport messaging gives the same result as the manual version, but without corrupting the data structures or control flow used in the steady-state. It also exposes the true information flow, allowing the compiler to deliver the message in the most efficient way for a given architecture. Finally, teleport messaging offers powerful control over timing and latency beyond what is utilized in this example.

2. STREAM DEPENDENCE FUNCTION

This section defines a stream dependence function, SDEP, that describes how one actor depends on the execution of another actor in the stream graph. SDEP is meaningful only for pairs of actors that are connected by a directed path in the stream graph. We say that the *upstream* actor is at the start of the path, while the *downstream* actor is at the end. Dependences between parallel actors (e.g., parallel branches of a splitjoin) currently fall outside the scope of this model but could be addressed in future work (see Section 7).

An execution ϕ of a dataflow graph is an ordered sequence of actor firings. Each firing represents the execution of a single phase of the actor. Let $\phi[i]$ denote the ith actor appearing in execution ϕ, and let $|\phi \wedge A|$ denote the number of times that actor A appears in ϕ. An execution is legal if the dataflow requirements are respected; that is, for all i, the sequential firing of actors $\phi[0]$ through $\phi[i-1]$ leaves enough items on the communication channels for $\phi[i]$ to fire its next phase atomically. Let Φ denote the set of legal executions. Note that while Φ is an infinite set, each $\phi \in \Phi$ is a finite sequence.

Informally, $\text{SDEP}_{A \leftarrow B}(n)$ represents the minimum number of times that actor A must execute to make it possible for actor B to execute n times. This dependence is meaningful only if A is upstream of B; otherwise, SDEP assumes a value of zero. Because the I/O rates of each actor are known at compile time, SDEP is a static mapping.

A formal definition of SDEP using the notations introduced above is as follows:

DEFINITION 1. *(SDEP)*

$$\text{SDEP}_{A \leftarrow B}(n) \quad = \quad \min_{\substack{\phi \in \Phi, \\ |\phi \wedge B| = n}} |\phi \wedge A|$$

This equation reads: over all legal executions in which B fires n times, $\text{SDEP}_{A \leftarrow B}(n)$ is the minimum number of times that A fires. Figure 7 illustrates an example of SDEP for the stream graph in Figure 8.

3. CALCULATING SDEP

It is straightforward to calculate $\text{SDEP}_{A \leftarrow B}(n)$ via a fine-grained simulation of the stream graph. Our approach is to construct an execution ϕ that provides the minimum value of $|\phi \wedge A|$ that is selected in Definition 1. We construct ϕ by simulating the stream graph's execution of a "pull schedule" with respect to actor B (see Algorithm 1).

Intuitively, a pull schedule for X is one that executes other nodes as few times as possible for each firing of X. This is achieved by calculating the demand for data items on the input channels of X, and then propagating the demand back through the stream graph via pull scheduling of the actors connected to X. Pull scheduling results in a fine-grained interleaving of actor firings. Some stream graphs admit multiple pull schedules, as actors might be connected to multiple inputs that can be scheduled in any order; however, the set of actor executions remains constant even as the order changes. The following theorem allows us to use a pull schedule to calculate the SDEP function.

THEOREM 1.

$$\text{SDEP}_{A \leftarrow B}(n) = |\textbf{pullSchedule}(B, n) \wedge A|$$

PROOF. By construction, **pullSchedule**(B, n) executes each node in the graph as few times as possible for B to fire n times. Thus, there is no execution containing n executions of B where A executes fewer times. The theorem follows from the definition of SDEP. □

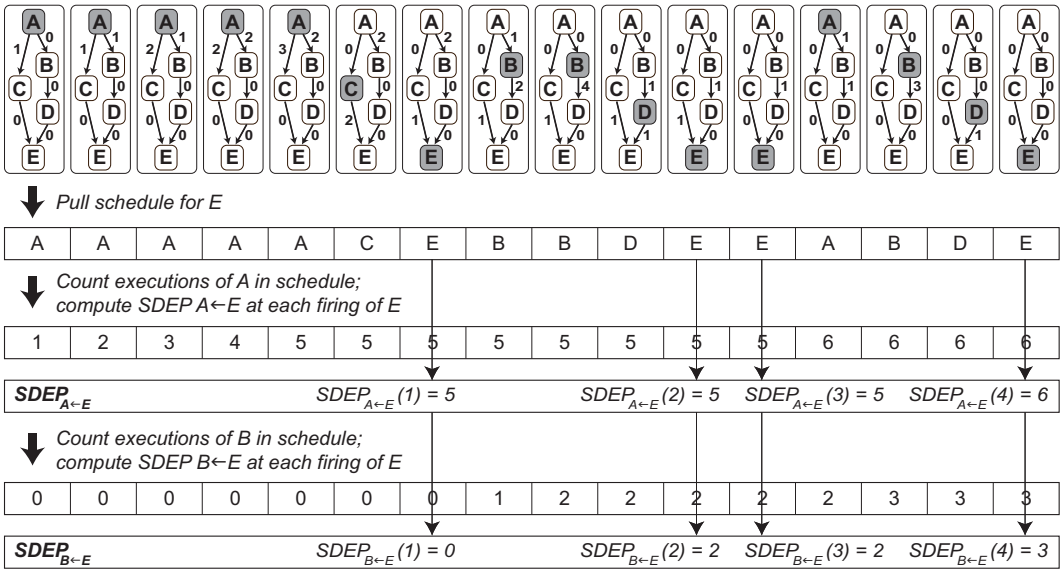

Figure 7: Example *SDEP* calculation for stream graph in Figure 8. The stream graphs illustrate a steady state cycle of a "pull schedule"; execution proceeds from left to right, and channels are annotated with the number of items present. The second line lists the actors that fire in a pull schedule for E. The third line counts the number of times that A executes in the pull schedule, and the fourth line illustrates the computation of $SDEP_{A \leftarrow E}(n)$: the number of times that A executes before the nth execution of E. The last two lines illustrate the computation of $SDEP_{B \leftarrow E}$.

ALGORITHM 1. *(Pull scheduling)*

```
// Returns a pull schedule for n executions of X
pullSchedule(X, n) {
  φ = {}
  for i = 1 to n {
    // execute predecessors of X until X can execute
    for all input channels c_i of X
      while X needs more items on c_i in order to fire
        // extend schedule (∘ denotes concatenation)
        φ = φ ∘ pullSchedule(source(c_i), 1)
    // add X to schedule
    φ = φ ∘ X
    // update number of items on I/O channels of X
    simulateExecution(X)
  }
  return φ
}
```

Some example SDEP calculations appear in Figure 7. The results are summarized in the following table.

n	$\text{SDEP}_{A \leftarrow E}(n)$	$\text{SDEP}_{B \leftarrow E}(n)$
1	5	0
2	5	2
3	5	2
4	6	3

Note that SDEP is non-linear due to mis-matching I/O rates in the stream graph. However, for longer execution traces, there is a pattern in the marginal growth of SDEP (i.e., in $\text{SDEP}(n) - \text{SDEP}(n-1)$); this quantity follows a cyclic pattern and has the same periodicity as the steady state of the stream graph. A steady state $\mathcal{S} \in \Phi$ is an execution that does not change the buffering in the channels—that is, the number of items on each channel after the execution is the same as it was before the execution. Calculating a steady state is well-understood [20]. The execution simulated in

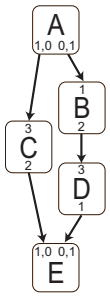

Figure 8: Example stream graph. Nodes are annotated with their I/O rates. For example, node C consumes 3 items and produces 2 items on each execution. Node A is a round-robin splitter that produces one item on its left channel during the first phase, and one item on its right channel during the second phase (similarly for Node E).

Figure 7 is a steady state, meaning that additional entries of the pull schedule will repeat the pattern given in the figure. Thus, SDEP will also grow in the same pattern, and we can calculate $\text{SDEP}_{A \leftarrow E}(n)$ for $n > 4$ as follows[1]:

$$\text{SDEP}_{A \leftarrow E}(n) = p(n) * |\mathcal{S} \wedge A| + \text{SDEP}_{A \leftarrow E}(n - p(n) * |\mathcal{S} \wedge E|) \quad (1)$$

$$p(n) = \lfloor \tfrac{n}{|\mathcal{S} \wedge E|} \rfloor \quad (2)$$

where $\mathcal{S}$ is a steady state and $p(n)$ represents the number of steady states that E has completed by iteration n. The first term of Equation 1 gives the total number of times that A has fired in previous steady states, while the second term counts firings of A in the current steady state.

While Equation 1 works for actors A and E, it fails for certain corner cases in stream graphs. For example, for

[1]Note that for any two actors X and Y, $\text{SDEP}_{Y \leftarrow X}(0) = 0$.

$\text{SDEP}_{A \leftarrow C}(3)$ it detects exactly 3 steady state executions ($p(3) = 3$) and concludes that each requires 6 executions of A ($|\mathcal{S} \wedge A| = 6$). However, as shown in Figure 7, the last firing of C requires only 5 executions of A. C is unusual in that it finishes its steady state before the upstream actor A.

To handle the general case, we simulate *two* executions of the steady state (rather than one) for the base case of SDEP:

$$\text{SDEP}_{Y \leftarrow X}(n) = \tag{3}$$
$$\begin{cases} |\textbf{pullSchedule}(X, n) \wedge Y| & \text{if } n \leq 2 * |\mathcal{S} \wedge X| \\ q(n) * |\mathcal{S} \wedge Y| + & \text{otherwise} \\ \quad \text{SDEP}_{Y \leftarrow X}(n - q(n) * |\mathcal{S} \wedge X|) & \end{cases}$$

$$q(n) = \lfloor \frac{n}{|\mathcal{S} \wedge X|} \rfloor - 1 \tag{4}$$

This formulation increases the size of the base SDEP table by one steady state, and also sets $q(n)$ to be one unit smaller than $p(n)$. The result is that the last complete steady state is counted as part of the "current" iteration rather than a "completed" iteration. For example, Equation 3 evaluates $\text{SDEP}_{A \leftarrow C}(3)$ using $q(3) = 2$, yielding $\text{SDEP}_{A \leftarrow C}(3) = 2 * 6 + \text{SDEP}_{A \leftarrow C}(3 - 2 * 1) = 17$ as desired. Moreover, in complex cases[2], the last steady state adds important context to the SDEP lookup for a given execution.

Thus, to calculate $\text{SDEP}_{Y \leftarrow X}(n)$, it is not necessary to simulate a pull schedule for n iterations of X as described in Algorithm 1. Instead, one can simulate $2 * |\mathcal{S} \wedge X|$ iterations as a pre-processing step and answer all future SDEP queries in constant time, using Equation 3. In addition, the pull schedule for X can be reused to calculate SDEP from X to any other actor (e.g., $\text{SDEP}_{W \leftarrow X}$ in addition to $\text{SDEP}_{Y \leftarrow X}$).

However, note that the pull schedule for X can *not* be used to calculate SDEP from any actor other than X (e.g., $\text{SDEP}_{W \leftarrow Y}$). The guarantee provided by **pullSchedule**(X, n) is only with respect to the base actor X. For other pairs of actors in the graph, one actor might execute more than necessary for n executions of the other. For example, consider what happens if one calculates $\text{SDEP}_{A \leftarrow B}$ using the schedule in Figure 7 (which is a pull schedule for E). In the schedule, A executes 5 times before the first firing of B, so one would conclude that $\text{SDEP}_{A \leftarrow B}(1) = 5$. However, this is incorrect; since B could have fired after only 2 executions of A, the correct value is $\text{SDEP}_{A \leftarrow B}(1) = 2$. Thus, to calculate $\text{SDEP}_{Y \leftarrow X}$, it is essential to calculate **pullSchedule**$(X, |\mathcal{S} \wedge X|)$, that is, a steady state cycle of a pull schedule with respect to X.

It is also possible to calculate SDEP using a compositional approach. For example, $\text{SDEP}_{A \leftarrow E}$ from Figure 7 can be expressed as follows:

$$\text{SDEP}_{A \leftarrow E}(n) = \max \begin{cases} \text{SDEP}_{A \leftarrow B}(\text{SDEP}_{B \leftarrow E}(n)) \\ \text{SDEP}_{A \leftarrow C}(\text{SDEP}_{C \leftarrow E}(n)) \end{cases}$$

That is, to determine the minimum number of times that A must execute to enable n executions of E, first calculate the minimum number of times each of A's successors in the stream graph must execute for n executions of E. Then A must execute enough to enable all of these children to complete the given number of executions, which translates to the max operation shown above. Our implementation exploits this compositional property to tabulate SDEP in a hierarchical manner, rather than simulating a pull schedule.

[2]For example, if within each steady state, the first firing of X does not depend on the first firing of Y, and the last firing of X does not depend on the last firing of Y.

4. TELEPORT MESSAGING

Teleport messaging is a language construct that makes use of SDEP to achieve precise timing of control messages. It is included as part of the StreamIt language [25]. Teleport messaging represents out-of-band communication between two actors, distinct from the high-bandwidth dataflow in the stream graph. Messages are currently supported between any pair of actors with a meaningful SDEP relationship, i.e., wherever there is a directed path in the stream graph from one actor to the other. We say that a *downstream* message travels in the same direction as the steady-state data flow, whereas an *upstream* message travels against it.

Syntax. In order for actor A to send a message to actor B, the following steps need to be taken:

- B declares a message handler that is invoked when a message arrives. For example:

  ```
  handler increaseGain(float amount) {
    this.gain += amount;
  }
  ```

  Message handlers are akin to normal functions, except that they cannot access the input/output channels and they do not return values.

  For another example, see line 40 of Figure 3.

- A parent stream containing A and B declares a variable of type `portal<`T_B`>` that can forward messages to one or more actors of type T_B. The parent adds B to the portal and passes the portal to A during initialization.

  For example, see lines 8, 10 and 12 of Figure 3.

- To send a message, A invokes the handler method on the portal from within its steady-state work function. The handler invocation includes a range of latencies `[min:max]` specifying when the message should be delivered; if no latency is specified, then a default latency of `[0:0]` is used. The following illustrates an example.

  ```
  work pop 1 {
    float val = pop();
    if (val < THRESHOLD) {
      portalToB.increaseGain(0.1) [2:3];
    }
  }
  ```

  This code sends an `increaseGain` message to `portalToB` with minimum latency 2 and maximum latency 3.

  For another example, see line 25 of Figure 3.

Informal Semantics. The most interesting aspect of teleport messaging is the semantics for the message latency. Because there are many legal orderings of actor executions, there does not exist a notion of "global time" in a stream graph. The only common frame of reference between concurrently executing actors is the series of data items that is passed between them.

Intuitively, the message semantics can be thought of in terms of attaching tags to data items. If A sends a message to downstream actor B with a latency k, then this could be

implemented by tagging the items that A outputs k iterations later. These tags propagate through the stream graph; whenever an actor inputs an item that is tagged, all of its subsequent outputs are tagged. Then, the message handler of B is invoked immediately before the first invocation of B that inputs a tagged item. In this sense, the message has the semantics of traveling "with the data" through the stream graph, even though it is not necessarily implemented this way.

The intuition for upstream messages is similar. Consider that B is sending a message with latency k to upstream actor A in the stream graph. This means that A will receive the message immediately after its last invocation that produces an item affecting the output of B's kth firing, counting the current firing as 0. As before, we can also think of this in terms of A tagging items and B observing the tags. In this case, the latency constraint says that B must input a tagged item before it finishes k additional executions. The message is delivered immediately after the latest firing in A during which tagging could start without violating this constraint.

Formal Semantics. The SDEP function captures the data dependences in the graph and provides a natural means of defining a rendezvous point between two actors. The following definition leverages SDEP to give a precise meaning to message timing.

DEFINITION 2. *(Message delivery) Consider that S sends a message to receiver R with latency range $[k_1 : k_2]$ and that the message is sent during the nth execution of S. There are two cases*[3]*:*

1. *If R is downstream of S, then the message handler must be invoked in R immediately before its mth execution, where m is constrained as follows:*

$$n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$$

2. *If R is upstream of S, then the message handler must be invoked in R immediately after its mth execution, where m is constrained as follows:*

$$\text{SDEP}_{R \leftarrow S}(n + k_1) \leq m \leq \text{SDEP}_{R \leftarrow S}(n + k_2)$$

For example, consider the FIR code in Figure 3. On line 25, the `Source` sends a message to the `Multiply` actors with zero latency ($k_1 = k_2 = 0$). Consider that, as illustrated in Figure 6, a message is sent during the fifth execution of `Source` ($n = 5$). Because each `Multiply` is downstream of `Source`, the following equation constrains the iteration m at which the message should be delivered to a given `Multiply`:

$$\begin{gathered} n + k_1 \leq \text{SDEP}_{Source \leftarrow Multiply}(m) \leq n + k_2 \\ 5 \leq \text{SDEP}_{Source \leftarrow Multiply}(m) \leq 5 \\ 5 \leq m \leq 5 \\ m = 5 \end{gathered}$$

To calculate $\text{SDEP}_{Source \leftarrow Multiply}$, observe that `Source` produces one item per iteration, while each `Multiply` produces one item and consumes one item. Thus, the `Source` must fire m times before any given `Multiply` can execute m times, and $\text{SDEP}_{Source \leftarrow Multiply}(m) = m$. Substituting into the above equation yields $m = 5$. That is, the message is delivered to each `Multiply` immediately before its fifth execution. This is illustrated in Figures 6(c) and 6(d) for the first and second `Multiply` in the pipeline, respectively. The message arrives immediately before the fifth data item (which corresponds to the fifth execution).

[3]In a feedback path, both cases might apply. In this event, we assume the message is being sent upstream.

	Latency < 0	**Latency ≥ 0**
Message travels upstream	illegal	buffering and latency in schedule must not be too large
Message travels downstream	buffering and latency in schedule must not be too small	no constraint

Figure 9: Scheduling constraints imposed by messages.

Constraints on the Schedule. It is important to recognize that messaging can place constraints on the execution schedule. The different categories of constraints are illustrated in Figure 9. A negative-latency downstream message has the effect of synchronizing the arrival of the message with some data that was previously output by the sender (e.g., for the checksum example mentioned in the introduction). The latency requires the downstream receiver not to execute too far ahead (i.e., too close to the sender), or else it might process the data before the message arrives. This translates to a constraint on the minimum allowable latency between the sender and receiver actors in the schedule for the program. Intuitively, it also constrains the buffering of data: the data buffers must not grow too small, as otherwise the receiver would be too far ahead.

Similarly, a non-negative-latency upstream message places a constraint on the maximum allowable latency between the sender and receiver. This time the upstream actor must be throttled so that it does not get too far ahead before the message arrives. Intuitively, the amount of data buffered between the actors must not grow too large.

For upstream messages with negative latency, there always exist iterations of the sender during which any messages sent are impossible to deliver. Consider an iteration of the sender that is the first to depend on data propagating from the nth execution of the receiver. A negative-latency message would be delivered immediately after a *previous* iteration of the receiver, but since iteration n has already fired, the message is impossible to deliver. Conversely, a downstream message with positive or zero latency imposes no constraint on the schedule, as the sender has not yet produced the data that is synchronized with the message.

Unsatisfiable Constraints. Messaging constraints can be unsatisfiable—that is, assuming a message is sent on every iteration of the sender's work function, there does not exist a schedule that delivers all of the messages within the desired latency range. Such constraints should result in a compile-time error.

Figure 10 illustrates an example of unsatisfiable constraints. Though each messaging constraint is feasible in isolation, the set of constraints together is unsatisfiable. The unsatisfiability is caused by conflicting demands on the buffering between B and C. The message from B to C constrains this buffer to contain at least 10 items, while the message from D

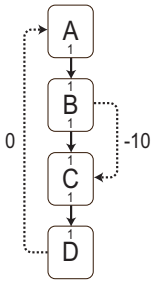

Figure 10: Example of unsatisfiable message constraints. Each node is annotated with its input and output rate. Messages are shown by dotted arrows, drawn from sender to receiver with a given latency. The constraints are satisfiable in isolation, but unsatisfiable in combination.

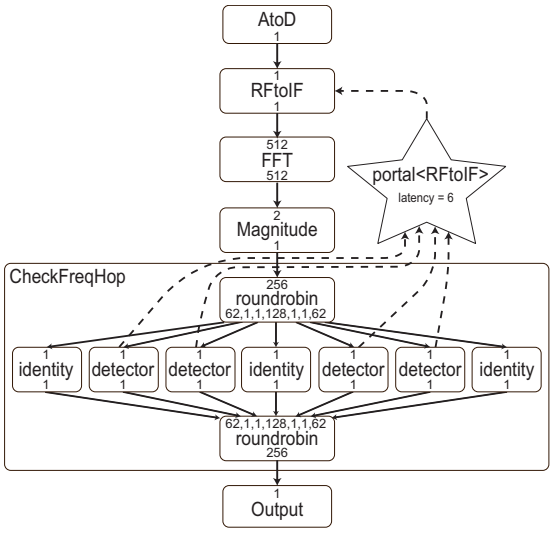

Figure 11: Stream graph of frequency hopping radio with teleport messaging. A portal delivers point-to-point latency-constrained messages from the detectors to the RFtoIF stage.

```
1  float->float filter RFtoIF(int N, float START_FREQ) {
2    float[N] weights;
3    int size, count;
4
5    init { setFrequency(START_FREQ); }
6
7    work pop 1 push 1 {
8      push(pop() * weights[count++]);
9      count = count % size;
10   }
11
12   handler setFrequency(float freq) {
13     count = 0;
14     size  = (int) (N * START_FREQ / freq);
15     for (int i = 0; i < size; i++)
16       weights[i] = sin(i * pi / size);
17   }
18 }
19
20 float->float splitjoin CheckFreqHop(int N,
21                                     float START_FREQ,
22                                     portal<RFtoIF> port) {
23   split roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
24   for (int i=1; i<=7; i++) {
25     if (i==1 || i==4 || i==7) {
26       add Identity<float>;
27     } else {
28       add float->float filter { // detector filter
29         work pop 1 push 1 {
30           float val = pop();
31           push(val);
32           if (val > Constants.HOP_THRESHOLD)
33             port.setFrequency(START_FREQ +
34                               i/7*Constants.BANDWIDTH) [6:6];
35         }
36       }
37     }
38   }
39   join roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
40 }
41
42 void->void pipeline FreqHoppingRadio {
43   int   N          = 256;
44   float START_FREQ = 2402000000;
45   portal <RFtoIF> port;
46
47   add AtoD(N);
48   add RFtoIF(N, START_FREQ) to port;
49   add FFT(N);
50   add Magnitude();
51   add CheckFreqHop(N, START_FREQ, port);
52   add Output()
53 }
```

Figure 12: Frequency hopping radio with teleport messaging. Arrows depict the path of messages from the sender to the receiver, via a portal declared in the top-level stream.

to A constrains it to be empty. We say that these two constraints *overlap* because the paths from sender to receiver intersect a common actor in the stream graph.

Finding a Schedule. In the presence of overlapping constraints, we leave to future work the problem of finding a legal execution schedule (if one exists). Because overlapping constraints can be detected statically, a given compiler may choose to prohibit overlapping constraints altogether.

For the case of non-overlapping constraints, a simple modification to pull scheduling will always result in a legal schedule (if one exists). First, note that a pull schedule always satisfies constraints imposed by upstream messages; because upstream (receiving) actors execute as little as possible per execution of the downstream (sending) actor, a message can be forwarded to the receiver immediately after sending. The receiver can then store the message and process it at the appropriate iteration. For downstream messages, the pull scheduler is modified to always execute one iteration of the upstream (sending) actor before any execution of the downstream (receiving) actor that would exceed the latency range. If the upstream actor needs more inputs to fire, then they can always be generated by actors that are further upstream (via a recursive call to the pull scheduling algorithm).

As described in Section 5.2, our compiler uses a simple implementation of messaging in which each sender or receiver executes in its own thread and waits for possible messages at appropriate iterations. This approach does not depend on producing a serial ordering of the actors at compile time.

5. CASE STUDY

To illustrate the pros and cons of teleport messaging, we implemented a spread-spectrum frequency hopping radio frontend [12] as shown in Figure 11. A frequency hopping radio is one in which the receiver switches between a set of known frequencies whenever it detects certain tones from the transmitter. The frequency hopping is a good match for control messages because the hopping interval is dynamic (based on data in the stream); it spans a large section of the stream graph (there is a Fast Fourier Transform (FFT) with 15 child actors, not shown, between the demodulator and the hop detector); and it requires precise message delivery. The delivery must be precise both to meet real-time require-

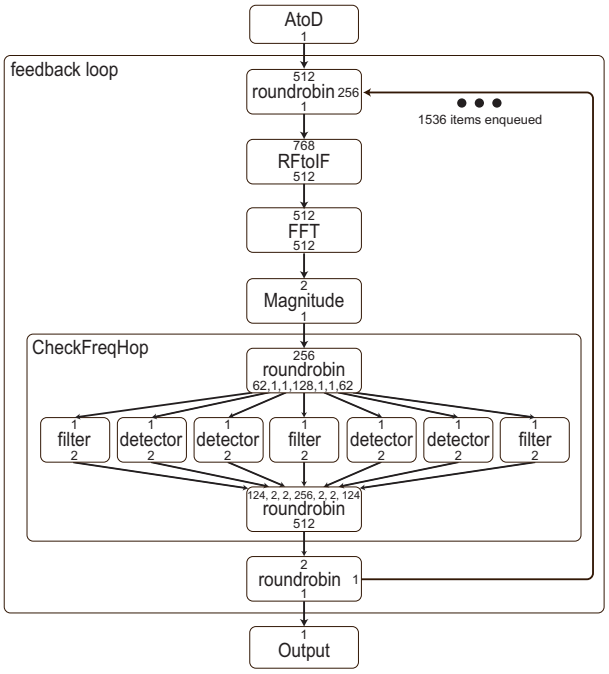

Figure 13: Stream graph of frequency hopping radio with control messages implemented manually. A feedback loop connects the detectors with the RFtoIF stage, and an item is sent on every invocation to indicate whether or not a message is present. The latency and periodicity of message delivery are governed by the data rates and the number of items on the feedback path.

ments (as the transmitter will leave the current frequency soon), and to ensure that the message falls at a logical frame boundary; if the frequency change is out of sync with the FFT, then the FFT will muddle the spectrum of the old and new frequency bands.

A StreamIt version of the radio frontend with teleport messaging appears in Figure 12. The FreqHoppingRadio pipeline creates a portal and adds the RFtoIF actor as a receiver (lines 45 and 48 respectively). The portal is passed to the CheckFreqHop stage, where four parallel detectors send messages into the portal if they detect a hop in the frequency they are monitoring (lines 32-35). The messages are sent with a latency of 6 to ensure a timely transition. To make sense of the latency, note that $\text{SDEP}_{RFtoIF \leftarrow D}(n) = 512 * n$ for each of the detector actors D. This comes about because the FFT stage consumes and produces 512 items[4]; each detector fires once per set of outputs from the FFT, but RFtoIF fires 512 times to fill the FFT input. Because of this SDEP relationship, messages sent from the detectors to RFtoIF are guaranteed to arrive only at iterations that are a multiple of 512. This satisfies the design criterion that a given FFT stage will not operate on data that were demodulated at two separate frequencies.

Another version of the frequency hopping radio appears in Figures 13 and 14. This version is functionally equivalent to the first, except that the control messages are implemented manually by embedding them in the data stream and in-

[4]Though the FFT is 256-way, the real and imaginary parts are interleaved on the tape, leading to an I/O rate of 512.

```
1   float->float filter RFtoIF(int N, float START_FREQ) {
2     float[N] weights;
3     int size, count;
4
5     init { setFrequency(START_FREQ); }
6
7   *  work pop 3*N push 2*N {
8   *    // manual loop to 2*N.  Factor of N because messages
9   *    // for given time slice come in groups of N; factor
10  *    // of 2 for data-rate conversion of Magnitude filter
11  *    for (int i=0; i<2*N; i++) {
12  *      push(pop() * weights[count++]);
13  *      count = count % size;
14  *    }
15  *    // manually check for messages;
16  *    // special value of 0 encodes no message
17  *    for (int i=0; i<N; i++) {
18  *      float freqHop = pop();
19  *      if (freqHop!=0)
20  *        setFrequency(freqHop);
21  *    }
22  *  }
23
24    handler setFrequency(float freq) {
25      count  = 0;
26      size   = (int) (N * START_FREQ / freq);
27      for (int i = 0; i < size; i++)
28        weights[i] = sin(i * pi / size);
29    }
30  }
31
32  float->float splitjoin CheckFreqHop(int N,
33                                      float START_FREQ) {
34    split roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
35    for (int i=1; i<=7; i++) {
36      if (i==1 || i==4 || i==7) {
37        add float->float filter {
38  *       work pop 1 push 2 {
39            push(pop());
40  *         push(0);
41          }
42        }
43      } else {
44        add float->float filter { // detector filter
45  *       work pop 1 push 2 {
46            float val = pop();
47            push(val);
48  *         if (val > Constants.HOP_THRESHOLD) {
49  *           push(START_FREQ + i/7*Constants.BANDWIDTH);
50  *         } else {
51  *           push(0);
52  *         }
53          }
54        }
55      }
56    }
57  *  join roundrobin(2*(N/4-2), 2, 2, 2*(N/2), 2, 2, 2*(N/4-2));
58  }
59
60  void->void pipeline FreqHoppingRadio {
61    int   N             = 256;
62    float START_FREQ    = 2402000000;
63
64    add AtoD(N);
65  *  add float->float feedbackloop {
66  *    // adjust joiner rates to match data rates in loop
67  *    join roundrobin(2*N,N);
68  *    body pipeline {
69  *      add RFtoIF(N, START_FREQ);
70  *      add FFT(N);
71  *      add Magnitude();
72  *      add CheckFreqHop(N, START_FREQ);
73  *    }
74  *    split roundrobin();
75  *    // number of items on loop path = latency * N
76  *    for (int i=0; i<6*N; i++)
77  *      enqueue(0);
78  *  }
79    add Output()
80  }
```

Figure 14: Frequency hopping radio with manual feedback loop for event handling. Lines that differ from Figure 12 are marked with an asterisk.

troducing a feedback loop. Because the number of items transfered around the loop must be constant from one iteration to the next, a data item is sent whether or not there is a message as part of the algorithm. The RFtoIF filter checks the values from the loop on every iteration; if the value is non-zero, it is treated as a message (the new frequency), while a value of zero is ignored (no message). The I/O rate of the RFtoIF filter has been scaled up to ensure that the messaging information is received at intervals of 512 iterations (as in the version with portals). To achieve the desired messaging latency of 6 frames, $6 * 256 = 1536$ items are enqueued on the feedback path prior to execution.

5.1 Discussion

Teleport messaging offers several benefits compared to a manual implementation of equivalent functionality. While embedding messages in the data stream is equally precise, it involves several tedious and error-prone changes, not only to the stream graph but also to the steady-state execution code within the actors. In particular, the manual derivation of the loop delay, adjustment of the actor I/O rates, and implicit interleaving of data items with control messages has a negative impact on the readability and maintainability of the code. Teleport messaging provides the same level of precision, but with the simplicity of a method call.

Teleport messaging also has advantages from a compiler standpoint. By separating the data-intensive code from the control-oriented code, the common case of steady-state execution is not sacrificed for the uncommon case of message processing. There are no "dummy items" serving as placeholders in the static-rate channels. In addition, by exposing the message latency as part of the language, the compiler can infer the true dependences between actor firings and reorder the execution so long as the message constraints are respected. The actual message delivery can be implemented in the most efficient way for a given architecture.

A final benefit of teleport messaging is the clean interface provided by the portals. Since a portal can have multiple receivers, it is straightforward to send a message that is delivered synchronously to two actors in parallel streams. For example, consider a vocoder (an encoder for voice signals) that is separately manipulating the magnitude and phase components of a signal. If something triggers an adjustment to the speech transformation (e.g., the speaker requests a change of pitch) then the mask needs to be updated at the same time relative to data in both parallel streams. A portal that contains both components seamlessly provides this functionality. Finally, portals are useful as an external programming interface; an application can export a portal based on an interface type without exposing the underlying actor implementation.

One aspect of teleport messaging might be considered unusual: the granularity of message delivery can be affected by changes in granularity elsewhere in the stream graph. This is evident in the frequency hopping radio, as the I/O rate of 512 on the FFT implies that the RFToIF stage will receive messages from CheckFreqHop at most once every 512 iterations. (If the FFT were coarsened to 1024-way, the granularity of messages in RFToIF would increase accordingly.) In this case the behavior is desirable, as messages should not interrupt frame boundaries. It seems that in many cases, the I/O rates are meaningful aspects of the program and their influence on message granularity is appropriate. Nonetheless, this non-local influence might come as a surprise to programmers. If the FFT granularity is scaled up for a different reason (e.g., caching behavior), the effects on message granularity might be unwanted.

This suggests that it might be worthwhile, in future work, to investigate additional mechanisms for programmers to specify the messaging contract independently of the declared I/O rates. For example, a parent stream could override the I/O rates of a child for the sake of a given SDEP calculation. The scheduler would deliver messages according to the parent's expectation of SDEP, or report an error if such delivery is incompatible with the actual I/O rates.

5.2 Experimental Evaluation

We have implemented teleport messaging in the StreamIt compiler infrastructure [10], with a backend that targets a cluster of workstations. A StreamIt program is compiled to a set of parallel threads; if two threads are allocated to different machines, they communicate via dedicated TCP/IP connections. Messages are supported via auxiliary communication channels that transmit two kinds of signals from senders to receivers: 1) the contents of a control message, or 2) a *credit* that indicates the receiver can execute some number of iterations before checking for a message again.

Each actor alternates between normal execution and checking for the exchange of credits. This serves to throttle the message receiver in accordance with the constraints (Section 4), as an actor will block waiting for credits until the sender has reached a given point in its execution. The compiler calculates the SDEP information and schedules the exchange of credits to make sure that the timing constraints are respected. When a message is sent, it is tagged with the iteration number during which the receiver should process it; this is also calculated using SDEP in the compiler.

We chose a cluster-based evaluation for two reasons. First, many streaming applications run on the server side (e.g., cell phone base stations, radar processing, HDTV editing) and require large computational resources. Second, clusters provide a simple abstraction for distributed and parallel computing—multiple program counters, and distributed memories—which is at the heart of emerging multi-core architectures for embedded, desktop, and server computing.

The teleport implementation of the frequency hopping radio was compiled into 29 threads whereas the alternate version using a feedback loop results in 33 threads. Each thread corresponds to a single actor (there are more threads than appear in Figures 11 and 13 because the FFT stage is a pipeline composed of several actors). The thread mapping is done using a dynamic programming algorithm that aims to reduce the overall bottleneck, thereby maximizing throughput (outputs per unit time). Threads are assigned to one of sixteen 750Mhz Pentium III workstations, each with a 256Kb cache. The machines are interconnected using a fully switched 100Mb network.

Figure 15 shows the measured throughput (y-axis) for various cluster sizes. Note that due to the limited parallelism in the two implementations of the frequency hopper, cluster configurations with more than five workstations lead to negligible performance gains. From the data, we can observe that teleport messaging achieves a maximal throughput that is 49% better than its counterpart. We attribute this speedup primarily to reduced communication overhead. A detailed analysis of the results indicates that teleport mes-

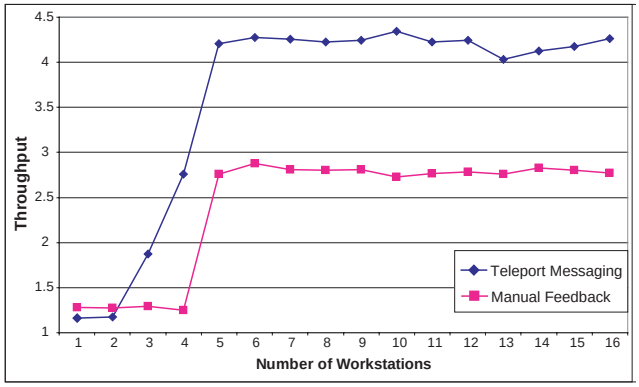

Figure 15: Throughput as a function of the number of workstations in the cluster.

saging reduces the number of items communicated by 35%. While the feedback loop version sends a message placeholder on every iteration, teleport messaging uses credits to allow the receiver to execute several iterations at a time without checking for messages. The amount of communications savings is dictated by the message latency, as larger latencies allow for a less frequent exchange of credits.

6. RELATED WORK

The work most closely related to ours comes from the fields of heterogeneous modeling, program slicing, and domain-specific languages.

The models of computation in our system are closely related to those explored in the Ptolemy project for heterogeneous design [19]. As part of this effort, Lee et al. have established the Synchronous Dataflow (SDF) paradigm [20] and have developed hybrid models that incorporate Dynamic Dataflow (DDF, in which the I/O rates of actors are fully dynamic). Boolean Dataflow (BDF) [11] is a compromise between these two extremes; it computes a parameterized schedule of the graph at compile time, and substitutes runtime conditions to decide which paths are taken. The performance is nearly that of SDF while keeping some flexibility of DDF.

Teleport messaging shares the motivation of BDF, but is different in its approach. We believe that control messages represent a distinct and well-behaved class of dynamic communication in which a parameter is "pushed" into the receiving actor in an asynchronous way. Because the message handlers do not access the I/O channels of the receiving actor, their irregular invocations do not interfere with a given static schedule. Instead, the schedule is constrained only by the latency of control messages; if a message does not show up in the allotted window, then the receiving actor can go ahead with its high-bandwidth schedule. This is the distinction in the computational model. In addition, the static/dynamic integration offered by our system is integrated with language features that support the model.

Program slicing identifies the set of statements in a program that a given statement might depend on. There is a rich history of work in program slicing; see Tip [26] for a comprehensive review. Many program slicing techniques rely on the Program Dependence Graph as described by Horwitz et al. [13]. Program slicing has been applied for debugging, testing, and program analysis. In many respects, SDEP analysis can be thought of as a slicing technique for Synchronous Dataflow graphs. Because the input domain is restricted (in particular, because of the absence of control flow and recursion), the SDEP calculation can make stronger guarantees than slicing analyses for general procedural languages; SDEP is decidable, exact, and admits a compact representation in terms of the steady state schedule.

Pugh and Rosser present an iteration-based slicing algorithm [22] to identify the dynamic instances of statements (in terms of their loop iteration) that effect a given value. This bears some similarity to stream dependence analysis, as $\text{SDEP}_{A \leftarrow B}(n)$ represents the last iteration of actor A that affected the nth iteration of actor B. However, [22] focuses on the problem of computing the transitive closure of dependences in loops, in which some iterations do not depend on others. We are not interested in this question, as we assume that all actor invocations depend on their previous invocations; SDEP addresses the question of finding only the most recent invocation that is relevant. Moreover, our motivation differs from the slicing community, as we apply SDEP to enrich the semantics of language features. To the best of our knowledge, slicing has not been applied in this way before.

There are many domain-specific stream languages in addition to StreamIt. Streams have a long history in the programming languages community, with influences from dataflow, CSP, synchronous and functional languages; see Stephens [24] for a review. Languages of recent interest include Brook [4], Cg [21], StreamC/KernelC [16], Spidle [5], Occam [14], Sisal [9], and Parallel Haskell [1]. The principle differences between StreamIt and these languages are (*i*) StreamIt adopts the SDF model of computation, which narrows the application class but enables powerful optimizations, (*ii*) StreamIt's support for a "peek" construct that inspects an item without consuming it from the channel, (*iii*) the single-input, single-output hierarchical structure that StreamIt imposes on the stream graph, and (*iv*) teleport messaging as described in this paper.

7. CONCLUSIONS AND FUTURE WORK

This paper makes two contributions. First, it introduces teleport messaging: a powerful language construct enabling precise message delivery between nodes of a distributed stream program. In comparison with other methods to implement messaging functionality in a Synchronous Dataflow model, teleport messaging is arguably more readable, more robust, and easier to maintain. In addition, our implementation of teleport messaging in the StreamIt compiler results in a 49% performance improvement for a frequency hopping radio running on a cluster of workstations. Like several other declarative language constructs, teleport messaging improves performance by exposing the true dependences to the compiler and allowing it to optimize the communication.

Second, this paper formulates SDEP, a natural and useful dependence representation for the streaming domain. While this paper applies SDEP to a new language construct, we envision other applications as well. For example, SDEP could be used in a debugger to identify which iterations of one actor are affecting a given iteration of another. In a software-based speculation system [8], SDEP could be applied to trace the effects of a failed prediction and to roll back the appropriate actor executions. Analogous to representations such as dependence levels [2], direction vectors [27], and dependence polyhedra [15] for scientific programs, SDEP provides

dependence information that could be used to test or verify program transformations. Also, SDEP offers a new method for measuring latency in a stream graph.

There are some limitations in the current study that are fertile grounds for future research. First, our formulation of SDEP requires a directed path in the stream graph between the actors in question. We are generalizing SDEP to actors that run in parallel by leveraging their data dependences with common predecessors (upstream) or successors (downstream). Second, as detailed in Section 4, we do not solve the general scheduling problem that incorporates overlapping constraints from teleport messaging; even determining whether or not a set of constraints is feasible (especially during the initialization schedule [17]) seems to be an interesting question. Third, in the current model only actors can send and receive messages. We are extending this into a hierarchical model where stream containers (such as pipelines) can also receive events and dispatch them precisely to other streams. Finally, our approach relies on the static communication rates present in SDF. It is interesting to consider teleport messaging in a more dynamic context; for example, downstream non-negative latency messages could be supported by embedding messages in data items, while other messages might require speculative delivery or modified timing contracts.

Our work can be viewed as integrating dynamic behavior into a static dataflow language. Our insight is that there is a class of control messages that only adjust parameters in the target actor; they do not otherwise affect the input or output channels upon delivery. This model enables a hybrid scheduling scheme in which the steady-state dataflow is exactly orchestrated at compile time, but there are windows in which a message could adjust an internal field of an actor between its execution steps. We consider this to be a promising avenue for creating a unified development environment that captures all aspects of stream application development without sacrificing either performance or programmability.

8. ACKNOWLEDGMENTS

We are very grateful to Jasper Lin for insightful comments on this paper. We also thank David Maze, Jasper Lin, and Michael Gordon for assistance with the StreamIt implementation, and the anonymous reviewers for their helpful comments. The StreamIt project is supported by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890; NSF awards CNS-0305453 and EIA-0071841; and the MIT Oxygen Alliance.

9. REFERENCES

[1] S. Aidtya, Arvind, L. Augustsson, J.-W. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Haskell Workshop*, 1995.

[2] J. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. *Proc. of the IBM Conf. on Parallel Computing and Scientific Computations*, 1982.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on Signal Proc.*, 1996.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[5] C. Consel, H. Hamdi, L. Reveillere, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Applications. Technical report, LABRI Research Report 1282-02, 2002.

[6] T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe. Challenges to combining general-purpose and multimedia processors. *IEEE Computer*, 30(12), 1997.

[7] K. Diefendorff and P. K. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9), 1997.

[8] M. Frank. *SUDS: Automatic Parallelization for Raw Processors*. PhD thesis, MIT, 2003.

[9] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille. The Sisal Model of Functional Programming and its Implementation. In *Proc. of the 2nd Aizu Int. Symp. on Parallel Algorithms/Architecture Synthesis*, 1997.

[10] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. ASPLOS, 2002.

[11] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Trans. on Computers*, 6, 1997.

[12] H. Harada and R. Prasad. *Simulation and Software Radio for Mobile Communications*. Artech House, 2002.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. 1988.

[14] Inmos Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[15] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL*, 1988.

[16] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 2003.

[17] M. A. Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for StreamIt language. Master's thesis, MIT, 2002.

[18] S. Kirkpatrick. Multimedia's impact on computer architecture. Presentation, Harvard University Colloquium Series, May 1997.

[19] E. A. Lee. Overview of the Ptolemy Project. Technical report, Tech Memo UCB/ERL M03/25, UC Berkeley, 2003.

[20] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, January 1987.

[21] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

[22] W. Pugh and E. Rosser. Iteration based slicing and its application to communication optimization. In *Proc. of the Int. Conf. on Supercomputing*, 1997.

[23] 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.

[24] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[25] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the Int. Conf. on Compiler Construction (CC)*, 2002.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[27] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois, Urbana-Champaign, 1982.