Teleport Messaging for Distributed Stream Programs

William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah and Saman Amarasinghe

Massachusetts Institute of Technology
PPoPP 2005

http://cag.lcs.mit.edu/streamit

Please note: This presentation was updated in September 2006 to simplify the timing of upstream messages. The corresponding update of the paper is available at http://cag.csail.mit.edu/commit/papers/05/thies-ppopp05.pdf
Streaming Application Domain

• Based on a stream of data
  – Radar tracking, microphone arrays, HDTV editing, cell phone base stations
  – Graphics, multimedia, software radio

• Properties of stream programs
  – Regular and repeating computation
  – Parallel, independent actors with explicit communication
  – Data items have short lifetimes

Amenable to aggressive compiler optimization

[ASPLOS ’02, PLDI ’03, LCTES’03, LCTES ’05]
Control Messages

• Occasionally, low-bandwidth control messages are sent between actors
• Often demands precise timing
  – Communications: adjust protocol, amplification, compression
  – Network router: cancel invalid packet
  – Adaptive beamformer: track a target
  – Respond to user input, runtime errors
  – Frequency hopping radio

What is the right programming model?
How to implement efficiently?
Supporting Control Messages

• Option 1: Synchronous method call
  PRO: - delivery transparent to user
  CON: - timing is unclear
  - limits parallelism

• Option 2: Embed message in stream
  PRO: - message arrives with data
  CON: - complicates filter code
  - complicates stream graph
  - runtime overhead
Teleport Messaging

• Looks like method call, but timed relative to data in the stream

```java
TargetFilter x;
if newProtocol(p) {
x.setProtocol(p) @ 2;
}
```

• PRO:
  – simple and precise for user
    • adjustable latency
    • can send upstream or downstream
  – exposes dependences to compiler
Outline

• StreamIt
• Teleport Messaging
• Case Study
• Related Work and Conclusion
Outline

• StreamIt
• Teleport Messaging
• Case Study
• Related Work and Conclusion
Model of Computation

- Synchronous Dataflow [Lee 92]
  - Graph of autonomous filters
  - Communicate via FIFO channels
  - Static I/O rates

- Compiler decides on an order of execution (schedule)
  - Many legal schedules
Example StreamIt Filter

```cpp
float->float filter LowPassFilter (int N, float[N] weights) {
    work peek N push 1 pop 1 { // N
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```
Example StreamIt Filter

```c
float->float filter LowPassFilter (int N, float[N] weights) {
    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
    handler setWeights(float[N] _weights) {
        weights = _weights;
    }
}
```
float->float filter LowPassFilter (int N, float[N] weights, Frontend f) {
    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        if (result == 0) {
            f.increaseGain() @ [2:5];
        }
        push(result);
        pop();
    }

    handler setWeights(float[N] _weights) {
        weights = _weights;
    }
}
StreamIt Language Overview

• StreamIt is a novel language for streaming
  – Exposes parallelism and communication
  – Architecture independent
  – Modular and composable
    • Simple structures composed to create complex graphs
  – Malleable
    • Change program behavior with small modifications

merge

may be any StreamIt language construct

pipeline  

filter

splitjoin  

splitter

joiner

feedback loop  

joiner

splitter
Outline

• StreamIt
• Teleport Messaging
• Case Study
• Related Work and Conclusion
Providing a Common Timeframe

• Control messages need precise timing with respect to data stream

• However, there is no global clock in distributed systems
  – Filters execute independently, whenever input is available

• Idea: define message timing with respect to data dependences
  – Must be robust to multiple datarates
  – Must be robust to splitting, joining
Stream Dependence Function (SDEP)

- Describes data dependences between filters

A → B
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftarrow B}(n) \]: minimum number of times that \( A \) must execute to make it possible for \( B \) to execute \( n \) times
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[
\text{SDEP}_{A \leftarrow B}(n): \text{minimum number of times that } A \text{ must execute to make it possible for } B \text{ to execute } n \text{ times}
\]
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftarrow B}(n) \]: minimum number of times that \( A \) must execute to make it possible for \( B \) to execute \( n \) times

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{SDEP}_{A \leftarrow B}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[
SDEP_{A \leftrightarrow B}(n): \text{minimum number of times that } A \text{ must execute to make it possible for } B \text{ to execute } n \text{ times}
\]
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ SDEP_{A \leftarrow B}(n) \]: minimum number of times that \( A \) must execute to make it possible for \( B \) to execute \( n \) times

<table>
<thead>
<tr>
<th>( n )</th>
<th>( SDEP_{A \leftarrow B}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

\( A \) push 2

\( pop \) 3

\( B \)
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftrightarrow B}(n) : \text{minimum number of times that } A \text{ must execute to make it possible for } B \text{ to execute } n \text{ times} \]

<table>
<thead>
<tr>
<th>n</th>
<th>SDEP(_{A \leftrightarrow B}(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftarrow B}(n) \]: minimum number of times that \( A \) must execute to make it possible for \( B \) to execute \( n \) times

<table>
<thead>
<tr>
<th>( n )</th>
<th>SDEP(_{A \leftarrow B}(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[
\text{SDEP}_{A \leftrightarrow B}(n): \text{minimum number of times that A must execute to make it possible for B to execute n times}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>SDEP_{A \leftrightarrow B}(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[
SDEP_{A \leftrightarrow B}(n) = \text{minimum number of times that A must execute to make it possible for B to execute n times}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>SDEP_{A \leftrightarrow B}(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[
\text{SDEP}_{A \leftarrow B}(n): \text{minimum number of times that } A \text{ must execute to make it possible for } B \text{ to execute } n \text{ times}
\]

<table>
<thead>
<tr>
<th>n</th>
<th>SDEP_{A \leftarrow B}(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftarrow B}(n) = \left\lceil \frac{n \times 3}{2} \right\rceil \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \text{SDEP}_{A \leftarrow B}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Calculating SDEP: General Case

\[ SDEP_{A \leftarrow C}(n) = \max_{i \in [1,m]} \left( SDEP_{A \leftarrow B_i}(SDEP_{B_i \leftarrow C}(n)) \right) \]

\[ SDEP_{A \leftarrow B}(n): \text{minimum number of times that} \ A \ \text{must execute to make it possible for} \ B \ \text{to execute} \ n \ \text{times} \]
Teleport Messaging using SDEP

• SDEP provides precise semantics for message timing

If S sends message to R:
• on the n\textsuperscript{th} execution of S
• with latency range \([k_1, k_2]\)

Then message is delivered to R:
• on any iteration \(m\) such that
\[ n + k_1 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq n + k_2 \]
Teleport Messaging using SDEP

• SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
  • on the $n$th execution of $S$
  • with latency range $[k_1, k_2]$

Then message is delivered to $R$:
  • on any iteration $m$ such that
    $$n + k_1 \leq SDEP_{S \leftarrow R}(m) \leq n + k_2$$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

If S sends message to R:
- on the $n$th execution of S
- with latency range $[k_1, k_2]$

Then message is delivered to R:
- on any iteration $m$ such that $n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
- on the $n$th execution of $S$
- with latency range $[k_1, k_2]$

Then message is delivered to $R$:
- on any iteration $m$ such that $n+k_1 \leq \text{SDEP}_{S\leftarrow R}(m) \leq n+k_2$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
- on the $n$th execution of $S$
- with latency range $[k_1, k_2]$

Then message is delivered to $R$:
- on any iteration $m$ such that $n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
  - on the $n$th execution of $S$
  - with latency range $[k_1, k_2]$

Then message is delivered to $R$:
  - on any iteration $m$ such that $n + k_1 \leq SDEP_{S \leftarrow R}(m) \leq n + k_2$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
- on the $n$th execution of $S$
- with latency range $[k_1, k_2]$

Then message is delivered to $R$:
- on any iteration $m$ such that $n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$
Teleport Messaging using SDEP

- SDEP provides precise semantics for message timing

**If S sends message to R:**
- on the \( n \)th execution of \( S \)
- with latency range \([k_1, k_2]\)

**Then message is delivered to R:**
- on any iteration \( m \) such that
  \[ n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2 \]
Teleport Messaging using SDEP

• SDEP provides precise semantics for message timing

If $S$ sends message to $R$:
  • on the $n$th execution of $S$
  • with latency range $[k_1, k_2]$  

Then message is delivered to $R$:
  • on any iteration $m$ such that

$$n + k_1 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq n + k_2$$
Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]

If S sends message to R:
  • on the $n$th execution of $S$
  • with latency range $[k_1, k_2]$

Then message is delivered to R:
  • on any iteration $m$ such that
    $n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$
If \( S \) sends message to \( R \):
- on the 4th execution of \( S \)
- with latency range \([k_1, k_2]\)

Then message is delivered to \( R \):
- on any iteration \( m \) such that
  \[
  n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2
  \]

Receiver \( r; \)
\( r.\text{increaseGain()} @ [0:0] \)
If $S$ sends message to $R$:

- on the 4th execution of $S$
- with latency range $[0, 0]$

Then message is delivered to $R$:

- on any iteration $m$ such that $n + k_1 \leq \text{SDEP}_{S \leftarrow R}(m) \leq n + k_2$
**Teleport Messaging using SDEP**

If $S$ sends message to $R$:
- on the 4th execution of $S$
- with latency range $[0, 0]$

Then message is delivered to $R$:
- on any iteration $m$ such that $4 + 0 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq 4 + 0$

Receiver $r$;
$r$.increaseGain() @ [0:0]
Teleport Messaging using SDEP

Receiver r;
r.increaseGain() @ [0:0]

If $S$ sends message to $R$:
- on the 4th execution of $S$
- with latency range $[0, 0]$

Then message is delivered to $R$:
- on any iteration $m$ such that $4+0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4+0$
- $\text{SDEP}_{S \leftarrow R}(m) = 4$
Teleport Messaging using SDEP

If \( S \) sends message to \( R \):

- on the 4th execution of \( S \)
- with latency range \([0, 0]\)

Then message is delivered to \( R \):

- on any iteration \( m \) such that
  \[
  4 + 0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4 + 0
  \]
  \[
  \text{SDEP}_{S \leftarrow R}(m) = 4
  \]
  \[
  m = 4
  \]

Receiver \( r \);
\( r.increaseGain() @ [0:0] \)
If $S$ sends message to $R$:
- on the 4th execution of $S$
- with latency range $[0, 0]$

Then message is delivered to $R$:
- on any iteration $m$ such that $4+0 \leq SDEP_{S \leftarrow R}(m) \leq 4+0$

$SDEP_{S \leftarrow R}(m) = 4$
$m = 4$

Receiver $r$;
r.increaseGain() @ [0:0]
Teleport Messaging using SDEP

If $S$ sends message to $R$:
- on the 4th execution of $S$
- with latency range $[0, 0]$ 

Then message is delivered to $R$:
- on any iteration $m$ such that $4+0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4+0$
  - $\text{SDEP}_{S \leftarrow R}(m) = 4$
  - $m = 4$

Receiver $r$;
$r$.increaseGain() @ $[0:0]$
Receiver r;
r.increaseGain() @ [0:0]

If $S$ sends message to $R$:

- on the 4th execution of $S$
- with latency range $[0, 0]$  

Then message is delivered to $R$:

- on any iteration $m$ such that $4 + 0 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq 4 + 0$
- $\text{SDEP}_{S \leftrightarrow R}(m) = 4$
- $m = 4$
Teleport Messaging using SDEP

If S sends message to R:
  • on the 4th execution of S
  • with latency range [0, 0]

Then message is delivered to R:
  • on any iteration m such that
    \[ 4 + 0 \leq SDEP_{S \leftarrow R}(m) \leq 4 + 0 \]
    \[ SDEP_{S \leftarrow R}(m) = 4 \]
    \[ m = 4 \]
If $S$ sends message to $R$:

- on the 4th execution of $S$
- with latency range $[0, 0]$

Then message is delivered to $R$:

- on any iteration $m$ such that $4 + 0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4 + 0$
  - $\text{SDEP}_{S \leftarrow R}(m) = 4$
  - $m = 4$
Teleport Messaging using SDEP

If $S$ sends message to $R$:
- on the 4th execution of $S$
- with latency range $[0, 0]$ 

Then message is delivered to $R$:
- on any iteration $m$ such that $4+0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4+0$
  
  $\text{SDEP}_{S \leftarrow R}(m) = 4$
  
  $m = 4$

Receiver $r$;
$r$.increaseGain() @ $[0:0]$
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides a unified abstraction
- Intuition:
  - If $S$ sends to $R$ with latency $k$
  - Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides provides a unified abstraction
• Intuition:
  – If $S$ sends to $R$ with latency $k$
  – Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps

Receiver r; r.decimate() @ [3:3]
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides provides a unified abstraction
• Intuition:
  – If $S$ sends to $R$ with latency $k$
  – Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps

Receiver $r$:
$r$.decimate() @ [3:3]
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides a unified abstraction
- Intuition:
  - If $S$ sends to $R$ with latency $k$
  - Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps

```java
Receiver r; r.decimate() @ [3:3]
```
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides a unified abstraction
• Intuition:
  – If $S$ sends to $R$ with latency $k$
  – Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps

Receiver $r$;
r.decimate() @ [3:3]
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow.
- Teleport messaging provides a unified abstraction.
- Intuition:
  - If $S$ sends to $R$ with latency $k$
  - Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps.

Receiver $r$;
r.decimate() @ [3:3]
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides provides a unified abstraction
• Intuition:
  – If S sends to R with latency k
  – Then R receives message after producing item that S sees in k of its own time steps

Receiver r;
r.decimate() @ [3:3]
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides a unified abstraction
- Intuition:
  - If $S$ sends to $R$ with latency $k$
  - Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps
Sending Messages Upstream

- If embedding messages in stream, must send in direction of dataflow
- Teleport messaging provides a unified abstraction
- Intuition:
  - If $S$ sends to $R$ with latency $k$
  - Then $R$ receives message after producing item that $S$ sees in $k$ of its own time steps

```java
Receiver r; r.decimate() @ [3:3]
```
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides provides a unified abstraction
• Intuition:
  – If S sends to R with latency k
  – Then R receives message after producing item that S sees in k of its own time steps

Receiver r;
r.decimate() @ [3:3]
Sending Messages Upstream

• If embedding messages in stream, must send in direction of dataflow
• Teleport messaging provides provides a unified abstraction
• Intuition:
  – If \( S \) sends to \( R \) with latency \( k \)
  – Then \( R \) receives message after producing item that \( S \) sees in \( k \) of its own time steps

\[ R \text{ receives message after iteration 7} \]
## Constraints Imposed on Schedule

<table>
<thead>
<tr>
<th></th>
<th>latency &lt; 0</th>
<th>latency ≥ 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message travels upstream</td>
<td>Illegal</td>
<td>Must not buffer too much data</td>
</tr>
<tr>
<td>Message travels downstream</td>
<td>Must not buffer too little data</td>
<td>No constraint</td>
</tr>
</tbody>
</table>
Finding a Schedule

• Non-overlapping messages: greedy scheduling algorithm

• Overlapping messages: future work
  – Overlapping constraints can be feasible in isolation, but infeasible in combination
Outline

• StreamIt
• Teleport Messaging
• Case Study
• Related Work and Conclusion
Frequency Hopping Radio

- Transmitter and receiver switch between set of known frequencies
- Transmitter indicates timing and target of hop using freq. pulse
- Receiver detects pulse downstream, adjusts RFtoIF with exact timing:
  - Switch at same time as transmitter
  - Switch at FFT frame boundary
Frequency Hopping Radio: Manual Feedback

- Introduce feedback loop with dummy items to indicate presence or absence of message
- To add latency, enqueue 1536 initial items on loop
- Extra changes needed along path of message
  - Interleave messages, data
  - Route messages to loop
  - Adjust I/O rates
- To respect FFT frames, change RFtoIF granularity
Frequency Hopping Radio: Teleport Messaging

- Use message latency of 6
- Modify only RFtoIF, detector
- FFT frame boundaries automatically respected: $SDEP_{RFIF\leftarrow det}(n) = 512*n$

Teleport messaging improves programmability
Preliminary Results

![Graph showing the relationship between throughput and the number of workstations, with two lines: one for Teleport Messaging and one for Manual Feedback.]
Outline

• StreamIt
• Teleport Messaging
• Case Study
• Related Work and Conclusion
Related Work

• Heterogeneous systems modeling
  – Ptolemy project (Lee et al.); scheduling (Bhattacharyya, …)
  – Boolean dataflow: parameterized data rates
  – Teleport messaging allows complete static scheduling

• Program slicing
  – Many researchers; see Tip’95 for survey
  – Like SDEP, find set of dependent operations
  – SDEP is more specialized; can calculate exactly

• Streaming languages
  – Brook, Cg, StreamC/KernelC, Spidle, Occam, Sisal, Parallel Haskell, Lustre, Esterel, Lucid Synchrone
  – Our goal: adding restricted dynamism to static language
Conclusion

Static
Powerful optimizations

Static-rate streaming
(Synchronous dataflow)

Teleport messaging

Dynamic
Expressive behavior

Control messages

StreamIt Language

• Teleport messaging provides precise and flexible event handling while allowing static optimizations
  – Data dependences (SDEP) is natural timing mechanism
  – Messaging exposes true communication to compiler
Extra Slides
Calculating SDEP in Practice

- Direct SDEP formulation:

\[ SDEP_{A\leftarrow C}(n) = \max \left[ \max(0, \frac{n^*o_c - k}{u_{b1}}) \cdot (o_{b1} - k), \max(0, \frac{n^*o_c - k}{u_{b2}}) \cdot (o_{b2} - k), \max(0, \frac{n^*o_c - k}{u_{b3}}) \cdot (o_{b3} - k) \right] \]

Direct calculation could grow unwieldy
Calculating SDEP in Practice

$$SDEP_{A\leftrightarrow C}(n)$$

$\begin{align*}
SDEP(n) &= \begin{cases} 
0 & n \in \text{init} \\
\text{lookup\_table}[n] & n \in \text{steady}_0 \\
k*S_A + SDEP(n - k*S_C) & n \in \text{steady}_k
\end{cases}
\end{align*}$

Build small SDEP table statically, use for all n
If $S$ sends **upstream** message to $R$:  
- with latency range $[k_1, k_2]$  
- on the $n$th execution of $S$

Then message is delivered to $R$:  
- after any iteration $m$ such that  
  
  $$SDEP_{R \leftarrow S}(n+k_1) \leq m \leq SDEP_{R \leftarrow S}(n+k_2)$$
If $\mathbf{S}$ sends \textbf{upstream} message to $\mathbf{R}$:
- with latency range $[k_1, k_2]$
- on the $n$th execution of $\mathbf{S}$

Then message is delivered to $\mathbf{R}$:
- after any iteration $m$ such that

$$SDEP_{R \leftarrow S}(n+k_1) \leq m \leq SDEP_{R \leftarrow S}(n+k_2)$$

Receiver $r$;
$r$.decimate() @ [3:3]
If **S** sends **upstream** message to **R**:
- with latency range \([3, 3]\)
- on the **n**th execution of **S**

Then message is delivered to **R**:
- after any iteration **m** such that
\[
SDEP_{R \leftarrow S}(n + k_1) \leq m \leq SDEP_{R \leftarrow S}(n + k_2)
\]

Receiver **r**; 
\[ r.decimate() @ [3:3] \]
Sending Messages Upstream

If \( S \) sends **upstream** message to \( R \):

- with latency range \([3, 3]\)
- on the \( 4 \text{th} \) execution of \( S \)

Then message is delivered to \( R \):

- after any iteration \( m \) such that

\[
S\text{DEP}_{R \leftarrow S}(n+k_1) \leq m \leq S\text{DEP}_{R \leftarrow S}(n+k_2)
\]

Receiver \( r; \)

\( r.\text{decimate}() \) @ \([3:3]\)
If $S$ sends \textbf{upstream} message to $R$:
- with latency range $[3, 3]$
- on the 4th execution of $S$

Then message is delivered to $R$:
- after any iteration $m$ such that $SDEP_{R\leftarrow S}(4+3) \leq m \leq SDEP_{R\leftarrow S}(4+3)$

Receiver $r$;
$r$.decimate() @ $[3:3]$
Sending Messages Upstream

If $S$ sends **upstream** message to $R$:
- with latency range $[3, 3]$
- on the **4th** execution of $S$

Then message is delivered to $R$:
- after any iteration $m$ such that

$$\text{SDEP}_{R \leftarrow S}(4+3) \leq m \leq \text{SDEP}_{R \leftarrow S}(4+3)$$

$$m = \text{SDEP}_{R \leftarrow S}(7)$$

Receiver $r$; $r$.decimate() @ $[3:3]$
Sending Messages Upstream

If $S$ sends **upstream** message to $R$:
- with latency range $[3, 3]$  
- on the 4th execution of $S$

Then message is delivered to $R$:
- after any iteration $m$ such that $SDEP_{R \leftarrow S}(4+3) \leq m \leq SDEP_{R \leftarrow S}(4+3)$
  
  $$m = SDEP_{R \leftarrow S}(7)$$
  
  $$m = 7$$

Receiver $r$; $r$.decimate() @ $[3:3]$
Constraints Imposed on Schedule

- If $S$ sends on iteration $n$, then $R$ receives on iteration $n+3$
  - Thus, if $S$ is on iteration $n$, then $R$ must not execute past $n+3$
  - Otherwise, $R$ could miss message
- Messages constrain the schedule
- If latency is -1 instead of 3, then no schedule satisfies constraint
- Some latencies are infeasible

Receiver $r$;
$r$.decimate() @ [3:3]
Implementation

• Teleport messaging implemented in cluster backend of StreamIt compiler
  – SDEP calculated at compile-time, stored in table

• Message delivery uses “credit system”
  – Sender sends two types of packets to receiver:
    1. **Credit**: “execute \( n \) times before checking again.”
    2. **Message**: “deliver this message at iteration \( m \).”
  – Frequency of credits depends on SDEP, latency range
  – Credits expose parallelism, reduce communication
Evaluation

• Evaluation platform:
  – Cluster of 16 Pentium III’s (750 Mhz)
  – Fully-switched 100 Mb network

• StreamIt cluster backend
  – Compile to set of parallel threads, expressed in C
  – Threads communicate via TCP/IP
  – Partitioning algorithm creates load-balanced threads