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
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;
}

void setProtocol(int p) {
    reconfig(p);
}
```

- **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

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();
    }
}
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;
    }
}
Example StreamIt Filter

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

![Diagram of StreamIt constructs](image)
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
Stream Dependence Function (SDEP)

• Describes data dependences between filters

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

\[
SDEP_{A \leftrightarrow B}(n)\]: 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 \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

\[ SDEP_{A \leftrightarrow B}(n) \]: 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 \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

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

- Diagram showing push 2 and pop 3 between A and B with a factor of 2.
Stream Dependence Function (SDEP)

- Describes data dependences between filters

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

<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

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

- Describes data dependences between filters

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

<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></td>
</tr>
</tbody>
</table>
Stream Dependence Function (SDEP)

- Describes data dependences between filters

\[ \text{SDEP}_{A \leftrightarrow B}(n) \]: 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

\[ \text{SDEP}_{A\leftarrow B}(n) \]: 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

\[
SDEP_{A \leftarrow B}(n) = \left\lfloor \frac{n \times 3}{2} \right\rfloor
\]

<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>

\(SDEP_{A \leftarrow B}(n)\): minimum number of times that A must execute to make it possible for B to execute n times
Calculating SDEP: General Case

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

SDEP is compositional
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

- 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

- 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

- 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\text{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

- 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

- 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

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

If $S$ sends message to $R$:  
- on the $4$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

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

```
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 $4+0 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq 4+0$

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 $4 + 0 \leq \text{SDEP}_{S \leftarrow R}(m) \leq 4 + 0$
  \[ \text{SDEP}_{S \leftarrow R}(m) = 4 \]
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]
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 $4 + 0 \leq SDEP_{S\leftarrow R}(m) \leq 4 + 0$
  
  $SDEP_{S\leftarrow R}(m) = 4$
  
  $m = 4$
Teleport Messaging using SDEP

If \textbf{S} sends message to \textbf{R}:
  
  - on the \textbf{4th} execution of \textbf{S}
  - with latency range \([0, 0]\)

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

If \textbf{S} sends message to \textbf{R}:
- on the 4th execution of \textbf{S}
- with latency range \([0, 0]\)

Then message is delivered to \textbf{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

Receiver r;
\texttt{r.increaseGain()}  @ [0:0]

If \textbf{S} sends message to \textbf{R}:
- on the \textbf{4th} execution of \textbf{S}
- with latency range \([0, 0]\)

Then message is delivered to \textbf{R}:
- on any iteration \(m\) such that
  \[4+0 \leq \text{SDEP}_{S \leftrightarrow R}(m) \leq 4+0\]
  \[\text{SDEP}_{S \leftrightarrow R}(m) = 4\]
  \[m = 4\]
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 when 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 when 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 when 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 when producing item that \( S \) sees in \( k \) of its own time steps

Receiver \( r; \)
\( r\text{.decimate()} @ [3:3] \)
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 when 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 when producing item that \( S \) sees in \( k \) of its own time steps

Receiver \( r \);
\( r.\text{decimate}() @ [3:3] \)
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 when 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 when 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 when 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 when 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 when producing item that $S$ sees in $k$ of its own time steps.

$R$ receives message on iteration 7.

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

<table>
<thead>
<tr>
<th>Latency Conditions</th>
<th>Message Travels Upstream</th>
<th>Message Travels Downstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency &lt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency &gt; 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Constraints Imposed on Schedule

<table>
<thead>
<tr>
<th></th>
<th>latency &lt; 0</th>
<th>latency = 0</th>
<th>latency &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message travels</td>
<td></td>
<td></td>
<td>Must not</td>
</tr>
<tr>
<td>upstream</td>
<td></td>
<td></td>
<td>buffer too</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>much data</td>
</tr>
<tr>
<td>Message travels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>downstream</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Constraints Imposed on Schedule

<table>
<thead>
<tr>
<th></th>
<th>latency $&lt; 0$</th>
<th>latency $= 0$</th>
<th>latency $&gt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message travels upstream</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Must not buffer too much data</td>
</tr>
<tr>
<td>Message travels downstream</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Constraints Imposed on Schedule

<table>
<thead>
<tr>
<th></th>
<th>latency &lt; 0</th>
<th>latency = 0</th>
<th>latency &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message travels upstream</td>
<td>Illegal</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></td>
<td></td>
</tr>
<tr>
<td></td>
<td>latency $&lt; 0$</td>
<td>latency $= 0$</td>
<td>latency $&gt; 0$</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>Message travels upstream</strong></td>
<td>Illegal</td>
<td>Illegal</td>
<td>Must not buffer too much data</td>
</tr>
<tr>
<td><strong>Message travels downstream</strong></td>
<td>Must not buffer too little data</td>
<td>No constraint</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:
  \[ \text{SDEP}_{RFIF \leftrightarrow \text{det}}(n) = 512 \times n \]

Teleport messaging improves programmability
Preliminary Results

Graph showing throughput vs. number of workstations.

- **Teleport Messaging**
- **Manual Feedback**

Throughput:
- 1.0
- 1.5
- 2.0
- 2.5
- 3.0
- 3.5
- 4.0
- 4.5

Number of Workstations:
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

The graph indicates a significant increase in throughput for Teleport Messaging as the number of workstations increases, while Manual Feedback shows a more steady increase.
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

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

- Direct SDEP formulation:

\[
SDEP_{A \leftarrow C}(n) = \max \left[ \max(0, \frac{n^*o_c - k}{u_b1})^*o_{b1} - k \right), \right.
\]

\[
\max(0, \frac{n^*o_c - k}{u_b2})^*o_{b2} - k \right], \right.
\]

\[
\max(0, \frac{n^*o_c - k}{u_b3})^*o_{b3} - k \right] \right]
\]

Direct calculation could grow unwieldy
Calculating SDEP in Practice

SDEP$_{A \leftrightarrow C}(n)$

• initialization: consumes all initial items

• steady state: repetition of each actor that does not change number of items on channels
Calculating SDEP in Practice

\[ S\text{DEP}_{A\leftrightarrow C}(n) \]

\[ S\text{DEP}(n) = \begin{cases} 
0 & n \in \text{init} \\
\text{lookup\_table}[n] & n \in \text{steady}_0 \\
KSA + S\text{DEP}(n - KSC) & n \in \text{steady}_k 
\end{cases} \]

→ Build small SDEP table statically, use for all n
Sending Messages Upstream

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 $R$:

- on any iteration $m$ such that

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

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

Then message is delivered to \( R \):
- on 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 $R$:

- on 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 \textbf{upstream} message to \( R \):

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

Then message is delivered to \( R \):

- on any iteration \( m \) such that

\[
SDEP_{R \leftrightarrow S}(n+k_1) \leq m \leq SDEP_{R \leftrightarrow S}(n+k_2)
\]

\[\text{Receiver } r; \quad r.\text{decimate()} \at [3:3]\]
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 $R$:
- on any iteration $m$ such that

$SDEP_{R \leftarrow S}(4+3) \leq m \leq SDEP_{R \leftarrow S}(4+3)$

Receiver $r$;
$r$.decimate() @ $[3:3]$
If $S$ sends **upstream** message to $R$:

- with latency range $[3, 3]$
- on the 4th execution of $S$

Then message is delivered to $R$:

- on any iteration $m$ such that

$$SDEP_{R\leftarrow S}(4+3) \leq m \leq SDEP_{R\leftarrow S}(4+3)$$

$$m = SDEP_{R\leftarrow S}(7)$$

Receiver $r$;

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

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

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

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

\[
m = 7
\]

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 0 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