StreamIt: A Language for Streaming Applications

William Thies, Michal Karczmarek, Michael Gordon, and Saman Amarasinghe

MIT Laboratory for Computer Science
Streaming Application Domain

- Based on streams of data
- Increasingly prevalent and important
  - Embedded systems
    - Cell phones, handheld computers, DSP’s
  - Desktop applications
    - Streaming media
    - Real-time encryption
    - Software radio
    - Graphics packages
  - High-performance servers
    - Software routers
    - Cell phone base stations
    - HDTV editing consoles
Developing Stream Programs

- C / C++ / Assembly
- Synchronous Dataflow
  - LUSTRE
  - SIGNAL
  - Silage
  - Lucid

Compiler-Conscious Language Design
The StreamIt Language

• Also a synchronous dataflow language  
  - With a few extra features

• Goals:
  - High performance
  - Improved programmer productivity

• Language Contributions:
  - Structured model of streams
  - Messaging system for control
  - Automatic program morphing

ENABLES
  Compiler
  Analysis &
  Optimization
Outline

• Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
• Results
• Conclusions
Outline

- Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
- Results
- Conclusions
Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize
Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize

- Insight: stream programs have structure

unstructured

structured
Structured Streams

- Hierarchical structures:
  - Pipeline
  - SplitJoin
  - Feedback Loop

- Basic programmable unit: Filter
Structured Streams

- Hierarchical structures:
  - Pipeline
  - SplitJoin
  - Feedback Loop

- Basic programmable unit: Filter
- Splits / Joins are compiler-defined
Representing Filters

- Autonomous unit of computation
  - No access to global resources
  - Communicates through FIFO channels
    - input.pop() - input.peek(index) - output.push(value)
  - Peek / pop / push rates must be constant

- Looks like a Java class, with
  - An initialization function
  - A steady-state “work” function
  - Message handler functions

- Implementation has nothing to do with Java
Filter Example: LowPassFilter

class LowPassFilter extends Filter {
    float[] weights;

    void init(int N) {
        weights = calcWeights(N);
        setPush(1); setPop(1); setPeek(N);
        setInput(Float.TYPE); setOutput(Float.TYPE);
    }

    void work() {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * input.peek(i);
        }
        output.push(result);
        input.pop();
    }
}
Filter Example: LowPassFilter

class LowPassFilter extends Filter {
    float[] weights;

    void init(int N) {
        weights = calcWeights(N);
        setPush(1); setPop(1); setPeek(N);
        setInput(Float.TYPE); setOutput(Float.TYPE);
    }

    void work() {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * input.peek(i);
        }
        output.push(result);
        input.pop();
    }
}
class LowPassFilter extends Filter {
    float[] weights;

    void init(int N) {
        weights = calcWeights(N);
        setPush(1); setPop(1); setPeek(N);
        setInput(Float.TYPE); setOutput(Float.TYPE);
    }

    void work() {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * input.peek(i);
        }
        output.push(result);
        input.pop();
    }
}
class LowPassFilter extends Filter {
    float[] weights;

    void init(int N) {
        weights = calcWeights(N);
        setPush(1); setPop(1); setPeek(N);
        setInput(Float.TYPE); setOutput(Float.TYPE);
    }

    void work() {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * input.peek(i);
        }
        output.push(result);
        input.pop();
    }
}
class LowPassFilter extends Filter {
    float[] weights;

    void init(int N) {
        weights = calcWeights(N);
        setPush(1); setPop(1); setPeek(N);
        setInput(Float.TYPE); setOutput(Float.TYPE);
    }

    void work() {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * input.peek(i);
        }
        output.push(result);
        input.pop();
    }
}
Pipeline Example: FM Radio

class FMRadio extends Pipeline {
    void init() {
        add(new DataSource);
        add(new LowPassFilter);
        add(new FMDemodulator);
        add(new Equalizer(8));
        add(new Speaker);
    }
}

Diagram:
- DataSource
  - LowPassFilter
    - FMDemodulator
      - Equalizer
        - Speaker
class FMRadio extends Pipeline {
    void init() {
        add(new DataSource());
        add(new LowPassFilter());
        add(new FMDemodulator());
        add(new Equalizer(8));
        add(new Speaker());
    }
}
class Equalizer extends Pipeline {

    void init(int N) {
        add(new SplitJoin() {
            void init() {
                setSplitter(Duplicate());
                float freq = 10000;
                for (int i = 0; i < N; i ++, freq*=2) {
                    add(new BandPassFilter(freq, 2*freq);
                }
                setJoiner(RoundRobin());
            }
        });
        add(new Adder(N));
    }
}
Why Structured Streams?

- Compare to structured control flow

  GOTO statements  If / else / for statements

- Tradeoff:
  PRO:    - more robust       - more analyzable
  CON:    - “restricted” style of programming
Structure Helps Programmers

- Modules are hierarchical and composable
  - Each structure is single-input, single-output

- Encapsulates common idioms
- Good textual representation
  - Enables parameterizable graphs
N-Element Merge Sort (3-level)
N-Element Merge Sort (K-level)

class MergeSort extends Pipeline {
    void init(int N, int K) {
        if (K==1) {
            add(new Sort(N));
        } else {
            add(new SplitJoin() {
                void init() {
                    setSplitter(RoundRobin());
                    add(new MergeSort(N/2, K-1));
                    add(new MergeSort(N/2, K-1));
                    setJoiner(RoundRobin());
                }});
            }
        add(new Merge(N));
    }
}
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

- Examples:
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

- Examples:

  ![Diagram](image-url)
Structure Helps Compilers

- Enables local, hierarchical analyses
  - Scheduling
  - Optimization
  - Parallelization
  - Load balancing

- Disallows non-sensical graphs
- Simplifies separate compilation
  - All blocks single-input, single-output
CON: Restricts Coding Style

- Some graphs need to be re-arranged
- Example: FFT
Outline

• Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
• Results
• Conclusions
Control Messages

- Structures for regular, high-bandwidth data
- But also need a control mechanism for irregular, low-bandwidth events

- Change volume on a cell phone
- Initiate handoff of stream
- Adjust network protocol
Supporting Control Messages

- Option 1: Embed message in stream
  PRO:  - message arrives with data
  CON:  - complicates filter code
         - complicates structure
         - runtime overhead

- Option 2: Synchronous method call
  PRO:  - delivery transparent to user
  CON:  - timing is unclear
         - limits parallelism
StreamIt Messaging System

- Looks like method call, but semantics differ
  
  ```java
  void raiseVolume(int v)
  myVolume += v;
  }
  ```

- No return value
- Asynchronous delivery
- Can broadcast to multiple targets
StreamIt Messaging System

- Looks like method call, but semantics differ
  
  ```java
  void work () {
    TargetFilter x;
    ...
    if (lowVolume())
      x.raiseVolume(10);
  }
  ```

  - No return value
  - Asynchronous delivery
  - Can broadcast to multiple targets

- Timed relative to data
  - User gains precision; compiler gains flexibility
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency
Message Timing

- A sends message to B with zero latency

Distance between wavefronts might have changed
Message Timing

- A sends message to B with zero latency
General Message Timing

- Latency of N means:
  - Message attached to wavefront that \textit{sender} sees in N executions
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions

- Examples:
  - A → B, latency 1
General Message Timing

- Latency of N means:
  - Message attached to wavefront that \textit{sender} sees in N executions

- Examples:
  - A $\rightarrow$ B, latency 1
General Message Timing

- Latency of N means:
  - Message attached to wavefront that sender sees in N executions

- Examples:
  - A → B, latency 1
  - B → A, latency 25
General Message Timing

- Latency of N means:
  - Message attached to wavefront that *sender* sees in N executions

- Examples:
  - A → B, latency 1
  - B → A, latency 25
Rationale

• Better for the programmer
  - Simplicity of method call
  - Precision of embedding in stream

• Better for the compiler
  - Program is easier to analyze
    • No code for timing / embedding
    • No control channels in stream graph
  - Can reorder filter firings, respecting constraints
  - Implement in most efficient way
Outline

• Design of StreamIt
  - Structured Streams
  - Messaging
    - Morphing
• Results
• Conclusions
Dynamic Changes to Stream

• Stream structure needs to change

• Examples
  - Switch radio from AM to FM
  - Change from Bluetooth to 802.11

Program “Morphing”
Dynamic Changes to Stream

- Stream structure needs to change

- Challenges for programmer:
  - Synchronizing the beginning, end of morphing
  - Preserving live data in the system
  - Efficiency
Morphing in StreamIt

- Send message to “init” to morph a structure

```java
class Equalizer extends Pipeline {
    void init (int N) {
        ...
    }
}
```
Morphing in StreamIt

- Send message to “init” to morph a structure

```java
class Equalizer extends Pipeline {
    void init (int N) {
        ...
    }
}
```
```
myEqualizer.init (6);
```
Morphing in StreamIt

- Send message to “init” to morph a structure

```java
myEqualizer.init (6);
```

```java
class Equalizer extends Pipeline {
    void init (int N) {
    ...
    }
}
```

- When message arrives, structure is replaced
- Live data is automatically drained
Rationale

• Programmer writes “init” only once
  - No need for complicated transitions
• Compiler optimizes each phase separately
  - Benefits from anticipation of phase changes
Outline

• Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
• Results
• Conclusions
Implementation

- Prototype StreamIt compiler complete
- Backends:
  - Uniprocessor
  - RAW: A tiled architecture with fine-grained, programmable communication
- Extended KOPI, open-source Java compiler
Results

- Developed applications in StreamIt
  - GSM Decoder
  - FM Radio
  - BeamFormer
  - FFT
  - Matrix multiply
  - CRC Encoder/Decoder

- Load-balancing transformations improve performance on RAW
Example: BeamFormer (Original)
Example: BeamFormer (Original)

- Useful work
- Blocked on send or receive
- Unused Tile

570 MFLOPS

110,000 cycles/item
Example: BeamFormer (Original)
Example: BeamFormer (Original)
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer
Example: BeamFormer (Balanced)
Example: BeamFormer (Balanced)

- Useful work
- Blocked on send or receive
- Unused Tile

1120 MFLOPS

55,000 cycles/item
Outline

• Design of StreamIt
  - Structured Streams
  - Messaging
  - Morphing
• Results
• Conclusions
Conclusions

- Compiler-conscious language design can improve both programmability and performance
  - Structure enables local, hierarchical analyses
  - Messaging simplifies code, exposes parallelism
  - Morphing allows optimization across phases

- Goal: Stream programming at high level of abstraction without sacrificing performance
For More Information

StreamIt Homepage

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