Opening the Levees for Stream Processing

IBM PL Day 7 May 2009

Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Giuliano Losa, Robert Soulé, Kun-Lung Wu
About this Talk

Technical
• A language for stream computing (work in progress)

Non-technical
• Exercise: pragmatic PL design
• Experience: PL guy in different domain
Continuous high-volume data stream processing

Applications: trading, medical monitoring, fleet management, radio astronomy, production plant control, etc.
SPADE and System S

SPADE compiler

SPADE graph

Native operators

System S Streaming middleware

Cluster
Outline

• Systems Solution
• Language Problem
• Language Design
• Design Process
• Future Work
Problem:
Design SPADE language and compiler.

Priorities:
- Performance: Ultra-fast streaming on a cluster.
- Generality: Support diverse set of applications.
- Usability: Hide complexity of distributed systems.
Problem:

**Co-Design SPADE language and compiler.**

Priorities:

- **Performance:** Ultra-fast streaming on a cluster.
- **Generality:** Support diverse set of applications.
- **Usability:** Hide complexity of distributed systems.

beyond StreamIt, StreamSQL, ...

also, interface with (new and legacy) native code
Outline

• Systems Solution
• Language Problem
• Language Design
• Design Process
• Future Work
Terminology

**Stream**
- Infinite sequence of tuples
- Edge in stream graph

**Operator**
- Reusable stream transformer
- May be primitive or composite

**Operator invocation**
- Defines its output streams
- Vertex in stream graph

**Port**
- Point where streams connect to operator
**SPADE Stream Graph**

```
stream<...> Bid = FileSource() {  //1
    ...
}  //2
stream<...> Quote = FileSource() {  //3
    ...
}  //4
stream<...> Sale = Join(Bid; Quote) {  //5
    ...
}  //6
() = FileSink(Sale) { ... }  //7
```
SPADE Types

```plaintext
stream<string buyer, string item, decimal64 price> Bid = FileSource() {
    ...
} //1

stream<string seller, string item, decimal64 price> Quote = FileSource() {
    ...
} //2

stream<string buyer, string seller, string item> Sale = Join(Bid; Quote) {
    ...
} //3

() = FileSink(Sale) { ... } //4
```

//5

//6

//7

//8

//9

//10

//11

//12

//13
SPADE Operator Customization

```c
stream<string buyer, string item, decimal64 price> Bid = FileSource() {  //1
    param  fileName : "BidSource.dat"; format: csv;  //2
}  //3
stream<string seller, string item, decimal64 price> Quote = FileSource() {  //4
    param  fileName : "SaleSource.dat"; format: csv;  //5
}  //6
stream<string buyer, string seller, string item> Sale = Join(Bid; Quote) {  //7
    window  Bid     : sliding, time(30);  //8
    Quote   : sliding, count(50);  //9
    param   match   : Bid.item == Quote.item && Bid.price >= Quote.price;  //10
    output  Sale    : item = Bid.item;  //11
}  //12
() = FileSink(Sale) { param fileName: "Result.dat"; format: csv; }  //13
```
SPADE Operator Definition

• Previous slides invoke and customize operators, but don’t define them.
• Support for 2 kinds of operator definition
SPADE Operator Definition

• Previous slides invoke and customize operators, but don’t define them.
• Support for 2 kinds of operator definition:

<table>
<thead>
<tr>
<th>Composite operator</th>
<th>Primitive operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulates SPADE stream graph</td>
<td>Encapsulates imperative code</td>
</tr>
<tr>
<td>Written in SPADE</td>
<td>Written in native language (e.g. C++)</td>
</tr>
<tr>
<td>Invoked/customized from SPADE</td>
<td>Invoked/customized from SPADE</td>
</tr>
<tr>
<td>Specialized by compiler</td>
<td>Specialized by compiler</td>
</tr>
</tbody>
</table>
Composite Operator Parameters

Original graph

Composite op. M

Expanded graph
PL Problem Revisited

Problem:
Design SPADE language and compiler.

Priorities:
- **Performance:** Generate specialized operator code.
- **Generality:** Arbitrary graphs, arbitrary C++ code.
- **Usability:** Composite operators, clear syntax.
PL Problem Revisited

Problem: Co-Design SPADE language and compiler.

Priorities:
- Performance: Generate specialized operator code.
- Generality: Arbitrary graphs, arbitrary C++ code.
- Usability: Composite operators, clear syntax.
Outline

• Systems Solution
• Language Problem
• Language Design
• Design Process
• Future Work
Language Design: Reuse+Combine

(Incomplete map of language influences)
Language Design: **Iterate**

<table>
<thead>
<tr>
<th>Sep ‘08</th>
<th>Oct ‘08</th>
<th>Nov ‘08</th>
<th>Dec ‘08</th>
<th>Jan ‘09</th>
<th>Feb ‘09</th>
<th>Mar ‘09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collected and prioritized requirements</td>
<td>Published internally</td>
<td>Wrote language spec</td>
<td>Wrote parser grammar</td>
<td>Feedback on spec from wiki, revised design</td>
<td>Feedback on spec from talk, revised some more</td>
<td>Designed compiler components</td>
</tr>
</tbody>
</table>
Language Design: **Iterate**

• **Meeting preparation**
  – Agenda (which features to discuss)
  – Examples (so everyone can see the issues)

• **During the meeting**
  – Project agenda and examples
  – Project meeting notes (decisions and rationale)
  – Be humble (maybe you are not right)
  – When “stuck” on an item, move on to next item
  – Wrap up meeting after 1 hour max

• **After meeting, send notes to everyone**
Outline

- Systems Solution
- Language Problem
- Language Design
- Design Process
- Future Work
Design Goals and Limits

Performance

Generality

Usability
Opening the Levees

Performance

Language-enabled optimizations

Distributed state

Formalized “correctness”

Generality

Usability
Conclusions

• Emerging technologies are language design opportunities.
• Practical language design: reuse, combine, iterate.
• Language specification available as TR.