Language Support for Stream Processing

Robert Soulé
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Introduction

- Stream processing is becoming popular:
  - Algorithmic trading
  - MPEG encoding/decoding
  - Web page analysis
# Languages, Runtimes, Optimizations

**Every domain has its own language:**
- Algorithmic trading $\rightarrow$ CQL/StreamSQL
- MPEG encoding/decoding $\rightarrow$ StreamIt
- Web page analysis $\rightarrow$ Sawzall

**Every language has its own runtime:**
- CQL/StreamSQL $\rightarrow$ STREAMS
- StreamIt $\rightarrow$ JVM
- Sawzall $\rightarrow$ MapReduce

**Every pair has their own optimizations:**
- CQL/StreamSQL $\rightarrow$ STREAMS $\rightarrow$ Operator Re-ordering
- StreamIt $\rightarrow$ JVM $\rightarrow$ Operator Fusion
- Sawzall $\rightarrow$ MapReduce $\rightarrow$ Data Parallelism
Problem

• Multiple streaming languages:
  – Developers are already trained in one language, but not another.
  – Applications in a domain are easier to express in one language than another.
  – Legacy code is easier to reuse from one language than another.

• Multiple platforms:
  – Different price points.
  – Different performance and reliability trade-offs.

• Organizations are forced to choose
Requirements

Problem:
Given multiple streaming languages
(e.g., StreamSQL, StreamIt, Sawzall),
apply multiple optimizations
(e.g., fusion, fission, reordering),
and run on multiple platforms
(e.g., cluster, supercomputer, internet).

Requirement:
**General** enough to support many languages
**Versatile** enough to allow many optimizations
**Portable** enough to allow for arbitrary runtimes
Requirements

Problem:
Given multiple streaming languages
(e.g., StreamSQL, StreamIt, Sawzall),
apply multiple optimizations
(e.g., fusion, fission, reordering),
and run on multiple platforms
(e.g., cluster, supercomputer, internet).

Requirement:
General enough to support many languages
Versatile enough to allow many optimizations
Portable enough to allow for arbitrary runtimes
Correct we should get the right results
Proposed Solution

Problem:
Given multiple streaming **languages**
(e.g., StreamSQL, StreamIt, Sawzall),
apply multiple **optimizations**
(e.g., fusion, fission, reordering),
and run on multiple **platforms**
(e.g., cluster, supercomputer, internet).

Solution:
Introduce a streaming **VEE**
(virtual execution environment).

[Diagram showing relationships between languages, optimizations, and platforms]
Proposed Solution

Solution:
Translators from languages to IL
Optimizers applied to the IL
Abstract common runtime functionality
Formalize the semantics and translations

Requirement:
General enough to support many languages
Versatile enough to allow many optimizations
Portable enough to allow for arbitrary runtimes
Correct we should get the right results
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Hypothesis

An intermediate language that encodes both state and communication, but is agnostic to the type system and underlying runtime, is sufficient for optimizing streaming languages.
Steps To Complete

1. **Formalize** the semantics for streaming languages and optimizations
2. Implement the language **translators** and runtime **abstractions**
3. Apply the **optimizations**
Outline

• Introduction
• Proposal
• **Work Plan**
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Outline

• Introduction
• Proposal
• Work Plan
  – **Formalism**
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Formalism

**Solution:**
Formalize the semantics and translations

**Requirement:**
Correct we should get the right results

- Brooklet, a core calculus for stream programming languages
- Allows us to reason about:
  - How languages map to runtimes
  - Correctness of streaming optimizations
- To appear in ESOP 2010
Brooklet Syntax

- Queue: FIFO communication channel.
- Variable: Local or shared state.
- Function: One step of local, deterministic computation.
- Operator: Vertex in graph of computations.
### Brooklet Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Expression</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queues</td>
<td>$(sales) \leftarrow \text{SaleJoin}(bids, $ask);$</td>
<td><img src="image" alt="Queues Diagram" /></td>
</tr>
<tr>
<td>Variables</td>
<td>$(sales) \leftarrow \text{SaleJoin}(bid, $ask);$</td>
<td><img src="image" alt="Variables Diagram" /></td>
</tr>
<tr>
<td>Functions</td>
<td>$(sales) \leftarrow \text{SaleJoin}(bids, $ask);$</td>
<td><img src="image" alt="Functions Diagram" /></td>
</tr>
</tbody>
</table>
Brooklet Semantics

- **F**: the function environment (Select, Window, SaleJoin…)
- **Q**: the state of the queues (asks, bids, …)
- **V**: the state of the variables ($lastAsk, $count)

\begin{align*}
\text{asks} &= \left\{ \langle XYZ, 35 \rangle, \langle IBM, 124 \rangle \right\} \\
\text{bids} &= \left\{ \langle IBM, 119 \rangle, \langle IBM, 124 \rangle \right\} \\
\text{lastAsk} &= \langle IBM, \infty \rangle \\
\text{lastAsk} &= \langle IBM, \infty \rangle \\
\text{count} &= 0
\end{align*}
Brooklet Semantics

• Repeat the following:
  – Selects any non-empty queue.
  – Remove a data item from the queue.
  – Call the function that consumes the data item.
  – Put the results in output queues and variables.

• An execution is a sequence of steps.
  – Transforms Q, V.
  – There are multiple possible executions.
Language Mappings Overview

- CQL → Brooklet
- Sawzall → Brooklet
- StreamIt → Brooklet
Language Mapping Overview

• Exposes implicit state as explicit variables.
• Exposes a mechanism for implementing global determinism.
• Abstracts away local computation with higher-order wrappers.
  – Statically bind the original function.
  – Dynamically adapt the runtime arguments.
CQL and SRA

- **Continuous Query Language (CQL)**
  - SQL + windows, streams
- **Stream Relational Algebra (SRA)**
  - Relational Algebra + streams

```sql
select IStream(*) from quote[Now], history
  where ask <= low quotes.ticker == history.ticker
IStream(BargainJoin(Now(quotes), history)))
```
CQL and SRA

Three types of operators:
- **S2R**: Windows
- **R2S**: Stream of updates, deletions, etc…
- **R2R**: Join, Filter

```plaintext
select IStream(*) from quote[Now], history
where ask <= low quotes.ticker == history.ticker
IStream(BargainJoin(Now(quotes), history)))
```
CQL Translation

IStream(BargainJoin(Now(quotes), history)))
CQL State

- Explicit state in relations.
- Implicit state in operators.

Now

quotes

history

BargainJoin

IStream
CQL Non-Determinism

- Expect language-level determinism.
- Implementation can be parallel.
- Non-determinism in join, resolved with timestamps.
CQL Translation Correctness theorem

- The Brooklet translation of CQL always yields the same output as the original CQL.
- State and determinism are encoded correctly.

For all CQL programs $P_c$ and for all inputs $I_c$:

$$\text{execute}_c(P_c, I_c) = \text{translate}_o(\text{execute}_b(\text{translate}_i(P_c, I_c)))$$
Optimizations

• Data Parallelism
• Operator Fusion
• Operator Reordering
Data Parallelism

- Possible if operator commutes.
- Stateless functions always commute.
- Simply inspect code for variables.
Operator Fusion

- Possible under the conditions that:
  - Operators appear in pipeline.
  - State is not modified elsewhere in the program.
- Simple inspection of topology and variables.
Selection Hoisting

- Leverage prior work on local static analysis.
  - Compute the read-write sets of operator functions.
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Translators & Abstractions

Solution:
Translators from languages to IL
Abstract common runtime functionality

Requirement:
General enough to support many languages
Portable enough to allow for arbitrary runtimes

• Babble, a compiler toolkit based on our formalism
• Allows us to translate from CQL, StreamIt, and Sawzall to IL
• Provides an abstraction for runtime
• Work-in-progress
Babble System Overview

• Translators to IL
  – CQL
  – StreamIt
  – Sawzall

• Specialization Framework
  – Modifies operators for type and use

• Runtime API
  – Abstract runtime

• Translator from IL to Runtime API
Translators

• IL based on Brooklet
• Need types and common operations
• Each application language imposes its own type system
  • CPP for common data types
  • Allows for portability
• Functions implemented in C
  • C is a lingua franca
Specialization Framework

• Operators need to be specialized for use and data
  – Example: Extended projection
    \[
    \text{select } price + 100 \text{ from ticks}
    \]

• Solution:
  – Macro substitution
  – \textit{xtc}'s C module
Runtime API

• Minimal requirements for the runtime
  – Variable initialization
  – Serialization/De-serialization
  – Common types

• Macros (babble.h) provide abstraction
  – Better performance than wrappers
  – We don’t re-invent the wheel
Generality Evaluation

• Provide translations from three languages:
  – CQL
  – StreamIt
  – Sawzall

• Execute three sample applications:
  – Linear Road
  – MPEG decoder
  – PageRank
Generality Challenges

• Minimal language kernels are not sufficient
  – No type system
  – No data definitions
  – No teleport messaging
  – No out-of-line declarations

• Translations are not “optimal”
  – Need incremental operators
Portability Evaluation

• Three sample applications:
  – Linear Road
  – MPEG decoder
  – PageRank

• Multiple Runtimes:
  – System S
  – ???
Portability Challenges

• What should the other runtime be?
• Runtime must be distributed:
  – StreamIt, STREAM are not suitable
• Runtime must be available:
  – MapReduce is proprietary
• Hadoop is a good option:
  – Open source, similar to MapReduce, distributed, runs the streaming language Pig
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Optimizations

**Solution:** Optimizers applied to the IL

**Requirement:** Versatile enough to allow many optimizations

- Open question:
  - Is an intermediate language that encodes state and communication sufficient for applying optimizations?

- Future work
Versatility Evaluation

• We have already formalized three optimizations:
  – Operator Re-ordering
  – Fusion
  – Data-Parallelism

• Implement these optimizations, apply them to our applications, demonstrate performance improvements.
Versatility Challenges

• These optimizations are “low hanging fruit”
• Is our formalism sufficient for others?
  – Sharing redundant queries
  – Pre-aggregating data in Map phase
  – Eliminate spurious synchronization in StreamIt
  – Stronger variants of the optimizations mentioned:
    • Data parallelism for statefull operators
Dynamic Optimizations

• Move an operator to a new machine
  – Load balancing
• Assign workers dynamically
  – Similar to MapReduce
• Dynamically configure operators
  – Add/Remove queues
• Need a runtime that supports this
Distributed CQL

• How do we partition data in relations?
  – Vertical partitioning
  – Horizontal partitioning

• Do the semantics need to change if we only send updates?
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Related Work

Stream processing

SVM
Labonte et al.
PACT’04

Runtime for executing IL on platforms

This Thesis

CQL
Arasu et al.
VLDB J.’06

P-Code
Nelson
CC’79

Translators from languages to IL
Comparison to Traditional VEEs

<table>
<thead>
<tr>
<th>Streaming VEE</th>
<th>Traditional VEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>For StreamSQL, Sawzall, StreamIt, ...</td>
<td>For Pascal, Java, C#, ...</td>
</tr>
<tr>
<td>IL for explicit streaming topology</td>
<td>IL is lower-level</td>
</tr>
<tr>
<td>Data in motion (queues)</td>
<td>Data at rest (registers)</td>
</tr>
<tr>
<td>Functions that run in parallel,</td>
<td>Instructions that run in sequence, one after the</td>
</tr>
<tr>
<td>continuously</td>
<td>other</td>
</tr>
</tbody>
</table>

Stream processing + Runtime for executing IL on platforms

P-Code Nelson CC’79

Translators from languages to IL
Comparison to CQL

CQL (Arasu et al. VLDB J.’06) | Streaming VEE
---|---
Described in terms of SRA (stream-relational algebra) | Uses more general streaming IL (not restricted to relational)
Inter-dependent with a single runtime | Virtual, independent of any particular runtime
Comparison to SVM

Stream processing

SVM Labonte et al. PACT’04

Runtime for executing IL on platforms

Translations from languages to IL

SVM (Labonte et al. PACT’04)

<table>
<thead>
<tr>
<th>Missing translators from any languages</th>
<th>Translation by recursion over syntax, making state explicit, encapsulating computation in functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous, assumes centralized controller</td>
<td>Asynchronous, no centralized controller</td>
</tr>
<tr>
<td>Assumes machine model with shared memory and CPUs</td>
<td>Abstracts away streaming runtime (may even be distributed cluster)</td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Proposal
• Work Plan
  – Formalism
  – Translators & Abstractions
  – Optimizations
• Related Work
• Conclusion
Conclusion

1. **Formalize** the semantics for streaming languages and optimizations
2. Implement the language **translators** and runtime **abstractions**
3. Apply the **optimizations**
Questions?

• soule@cs.nyu.edu