Hirundo: A Mechanism for Automated Production of Optimized Data Stream Graphs

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
Japan

Miyuru Dayarathna, Toyotaro Suzumura

25/4/2012
Introduction – Stream Computing

- More and more data analysis activities are conducted in **Online**

![Streaming Click-Through Rate Computation (Neumeyer et al.)](image1)

- **Op1**: Route keyless input events
- **Op2**: Join the serve and click events
- **Op3**: BotFilter
- **Op4**: Compute the correct click throughput rate

![An Underwater Acoustic Sensor Network (D. Pompili et al.)](image2)

- **Weatherbug sensor network (35000 sensors around the world)** ([http://developer.weatherbug.com/](http://developer.weatherbug.com/))
- **Highway toll Processing** ([http://pages.cs.brandeis.edu/~linearroad/](http://pages.cs.brandeis.edu/~linearroad/))
Overview – Stream Processing Systems

- Two famous stream processing system models
  - Relational Model (E.g., Aurora, Borealis, STREAM)
  - Operator-based Model (E.g., System S, S4)

- High performance stream programs
  - Program Structure
  - Topology and performance characteristics of stream processing system

- Target Stream processing Systems Model
  - Operator-based model
Overview – Stream Applications

(Roger Rea et al.)
Presentation Outline

- Introduction
- Research Problem
- Proposed Solution
- Related Work
- Methodology
- Evaluation
- Conclusion
Research Problem

Distributed stream programs written by programmers at large are not optimized.

Possible solutions

- Manual fine tuning
  Tremendous amount of programmer time
- Profile Driven Optimization
  Need to be made at the compiler/scheduler level (Is not generalizable)

(Suzumura et al. [27])

Proposed Solution and Contributions

Automatic Stream Program Performance Tuning

1. Tri-Operator Transformation
2. Transformer Blocks
3. Stream Program Performance Prediction
4. Stream Program Performance Characterization
5. Fault Tolerance
Methodology

Input

Sample Programs (S)

Generate similar programs with different flow graph layouts

Analyze Program Structure

Select programs U₁

Compile programs U₁ in parallel

Run each program in U₁ for W₁ time

Calculate performance details, rank programs

Select programs U₂

Select programs R₁

Compile programs U₃ in parallel

Run each program in U₃ for W₃ time

Output

A ranked list of programs (R₂)

Synthesize sample data

Sample data file

Sample Programs

stream program

Program Structure

Select programs U₁

Compile programs U₁ in parallel

Run each program in U₁ for W₁ time

Calculate performance details, rank programs

Select programs U₂

Select programs R₁

Compile programs U₃ in parallel

Run each program in U₃ for W₃ time

A ranked list of programs (R₂)
Related Work

- Distributed-memory multi computer systems [8][22][6]
  - Static Code Optimizations

- Automatic Composition of Workflows [23]
  - Hirundo address the problems that occur due to node failures in the runtime.

- Recurring Patterns for Optimizing Workflows [29]
  - Similar to the transformer blocks used in Hirundo


Related Work (Contd.)

- Performance prediction of parallel applications [30]
- Empirical cost information for producing optimized query plans [1][17]
  - Hirundo integrates the results of partial execution with empirical data

Relational Databases

- Subquery optimization [7]
- Table Partitioning [14]


Related Work (Contd.)

- **Stream Graph Partitioning [18][28]**
  - Stream program performance optimization at stream processing environment’s level

  Hirundo approaches the solution from the source program level by conducting high-level code transformations

- **Automatic optimization of MapReduce systems [4]**
- **DryadLINQ static optimizations [31]**


SPADE – An Operator-based Stream Processing Language

- Data flow graphs consisting of operators and streams [15]
- Operator – The smallest possible building block
  - Composite Operator – a reusable subgraph that can be invoked to define streams
  - Primitive Operator – basic building blocks of composite operators

SPADE Operators

(Henrique Andrade et al. [Andrade 2011])

- Sort, Barrier, Punctor, Delay [15]


Tri-Operator Transformation

- Based on Parallel Streams Design Pattern
- Transforms data flow graphs by three operator blocks at time

Why Select Three Operator Blocks?
- Simplicity
- Controllability of Code generation process

Key
- S – Source
- Fn – Functor n
- AG – Aggregate
- SI – Sink
Tri-Operator Transformation Algorithm

1. Traverse the flow graph three operator blocks at time
2. Apply Transformation and create operator block
3. Weld the transformed operator blocks
4. Transform operator blocks
5. G – input graph structure
6. Sample Programs
Transformer Patterns of Hirundo (i-j-k)

- A combination of i-j-k (i, j, and k are positive integers)

Input Operator Blocks

Transformed Operator Blocks

1-1-1

1-j-1

i-0-k

i, k > 1

j > 1
Transformer Patterns of Hirundo (Contd.)

D - Transformation depth is the maximum value i, j, k can have
Operator Blocks Fusion

- In most occasions more than one transformed operator blocks are created by `generate()`
- The transformed operator blocks need to be stitched meaningfully

Transformed Operator Blocks
Transformer Blocks

- A set of generic operator blocks that creates coupling between transformed operator blocks

![Diagram of Transformer Blocks]

- MUX-SINK
- DEMUX-SINK
- DEMUX-SOURCE
- MUX-SINK
- MUX-STREAM
- PARALLEL-SOURCE
- MULTI-FUNCTOR
- Transformed Operator Block
Sample Program Performance Prediction

- Large amount of sample programs are generated for a given input program.
  
  E.g., 32 sample programs for regex application with $d = 4$

  Running all sample programs took 17 minutes 22 seconds

- We use a performance prediction mechanism that uses,
  
  - graph structure (G)
  - Performance metrics (e.g. Throughput, elapsed time)
  - Optimization run depth ($d$)
  - Input data tuple schema (tschema)
  - Current optimization environment’s profile information
Preservation of Input Program Semantics

- Fusion uses fusions that have the same operation type sequence similar to the input program.
- Hirundo provides the notion of annotations to ensure semantically correctness of sample programs with stateful Operators.
- Froze annotation tags.

Key:
- S – Source
- Fn – Functor n
- AG – Aggregate
- SI – Sink

```
#Hirundo-meta:Froze:Start
```

```
#Hirundo-meta:Froze:End
```
Fault Tolerance of Hirundo

- It is important to have a fault tolerant stream program optimization process
- Unexpected instance failures in System S runtime might change the evaluation results
  - Large sample programs overloads the runtime-instances and they might run out of memory

How to find the sample programs with highest performance without breaking stream processing runtime environment’s stability?

- Hirundo uses *streamtool* of System S periodically to obtain the health information of System S runtime.
Fault Tolerance of Hirundo

1. Use Streamtool to get initial health record
2. Get the health record confirmed by the user
3. Periodically monitor runtime health using streamtool
4. Submit sample program
5. Failure in runtime?
   - Yes: Restart System S runtime
   - No: Three attempts?
     - No: Cancel sample program run
     - Yes: Mark the sample program as failure

Implementation

- Hirundo has been developed using Python
Implementation (Contd.)

- **Program Structure Analyzer**
  - A bespoke grammar written using GOLD parser generator
  - Current version includes 34 rules

- **Data Preparator**
  - Creates required sample data files
  - \([d(d+1)/2 - 1]\) number of files are created for transformation depth \(d\)

- **Parallel Compiler**
  - Sample program compilation on single node is expensive (27 Twitter programs take 50 minutes)
  - Parallel compiler reduces the time (27 Twitter programs take 8 minutes on 8 nodes, 84% reduction in time)
Experiment Applications

Volume Weighted Average Price (VWAP)

\[
\text{VWAP} = \frac{\sum_{i=1}^{n} (P_i \cdot V_i)}{\sum_{i=1}^{n} V_i}
\]

Pi – Traded Price
Vi – Traded Volume

Stream data converter (Regex)

Changes the format of the input data time data tuples

Twitter Hash Tag Counter (Twitter)

Finds the top ten topics from a twitter data stream

Increment the values of a stream of integers (numapp long)

Each operator increments the integer they receive by 100
Evaluation (1) - Sample Program Performance

- Experimental Environment
  - 8 Nodes of Linux CentOS release 5.4
  - IBM Infosphere Stream 1.2
  - Each machine has AMD Opteron (tm) Processor 242, 1MB L2 cache per core
  - 8GB memory, 250 GB hard drive, 1Gigabit Ethernet

- Experiment Applications
  - Volume Weighted Average Price (VWAP)
  - A data converter Application (Regex)
  - Twitter hash tag counter application (Twitter)
Evaluation (VWAP)

Throughput of sample programs generated for VWAP application (d = 8, Nodes = 8)
VWAP (Flow graph of Optimized version)

Input VWAP application

8SCSV_6F_AG_F_SI

8 times
Evaluation (Regex)

Throughput of sample programs generated for regex application (d = 4, Nodes = 8)

Input App

Sample Program Label

- Opt run 1
- Opt run 2
- Opt run 3
- Perf Predict run
Regex (Flow graph of Optimized Version)

Input Regex application

4SCSV_2F_2F_2F_2SI
Evaluation (Twitter)

Throughput of sample programs generated for Twitter application (d = 4, Nodes = 8)

Input App

Sample Program Label

Throughput (B/s)

Opt run 1
Opt run 2
Opt run 3
Perf Predict run
Twitter (Flow graph Optimized Version)

Input Twitter Application

U_4F_2F_F_AG_SI
Evaluation (2) - Performance Prediction

- Each experiment completed in less than 15 minutes time
Performance Clusters

- Objective: Find the characteristics of data flow graphs which leads to higher performance
- K-Means clustering was used to group the performance data points
- The minimum gap between the clusters was set to 100B/s
Performance Clusters (VWAP)

Having large number of source operators may produce high throughput for VWAP application
Performance Clusters (Twitter)

Having more tokenizer functors (F1) has supported for high performance of Twitter sample applications.
Having a variety of middle operators has produced less performance for Regex sample applications.
Regex (Less performance flow graphs)

SCSV_4F_2F_4F_4SI

SCSV_2F_4F_4F_4SI

4SCSV_4F_2F_4F_4SI
Limitations

- Current program structure analyzer’s grammar is limited to Source, Aggregate, Sink, and UDOP operators
- Hirundo assumes that programmer does not manually allocate node pools in his/her program
Conclusions

- We introduce a methodology for automatic production of optimized data stream programs
- Our approach produces higher performance gains
  - Regex (1.7 times), VWAP (2.3 times), Twitter (2.9 times)
- Tri-OP Transformation is able to produce sufficiently large number of sample programs with varied performance levels
  - When d = 4, we get at least 25 sample programs
- Our performance prediction mechanism is able to identify high performance sample programs for a completely new application
- Higher operator density at the middle part of the stream program produces less performance
Further Work

- Different techniques for improving the performance prediction of Hirundo
- Improve program structure analyzer and program generator modules
- An in-depth study on the performance characteristics of transformation blocks.
- Identify important program semantics and use them during program optimization process.
Thank You!

Questions?