A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters

Miyuru Dayarathna and Toyotaro Suzumura

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

4/25/2013

The 18th International Conference on Database Systems for Advanced Applications (DASFAA 2013)
Wuhan, China
Introduction

More and more data analysis activities are conducted in Online

Streaming Click-Through Rate Computation (Neumeyer et al.)

Weatherbug sensor network (35000 sensors around the world)
(http://developer.weatherbug.com/)

Highway toll Processing
(http://pages.cs.brandeis.edu/~linearroad/)

Visual Sensor Networks (VSNs)
http://vip.bu.edu/projects/vsns/

The Artemis Project (McGregor et al.)

DASFAA 2013
Data Stream Processing (Complex Event Processing)

Famous stream processing systems
- Aurora, Borealis, STREAM, System S, S4

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

- Introduction
- Research Problem & Proposed Solution
  - Automatic code generation for performance maintenance
- Related Work
- Methodology
  - Program selection algorithm
  - Program Switching model
- Evaluation & Conclusion
Highly availability is a key challenge for stream processing systems in providing continuous services.

Crash faults
- Operating system halts
- Power outages
- Virtual machine crashes

Crash faults take more time to recover
Research Problem

- The widely adapted solution – physical replication
  - Requires k replicas to tolerate up to (k-1) simultaneous failures.
- Maintaining large number of backup nodes costs a lot in terms of
  - Electricity
  - rack space
  - ventilation
  - Etc.

A Google Data Center
Balancing local stream processing load with public cloud (i.e., Hybrid cloud)

- Need the access to public clouds
- Makes it impossible to apply such solutions in certain applications
  - Stream application deals with sensitive information (e.g., healthcare, national defense, etc.)
  - Stream processing systems have license and software issues
Our approach – Based on Automatic Code generation

- Applicable to resource limited stream computing clusters
- Approach
  - Generate a variety of data flow graphs with each giving different performance characteristics for each input app.
  - Select sample programs with consistent high throughput performance compared to the input programs and run them in the cluster.
  - During a node failure, evaluate feasibility of performance maintenance using the existing stream applications and introduce different sample applications which produce better performance in the new environment.
- Our approach assigns priority to each input program and tries to maintain performance of high prioritized programs.
Contributions


2. Switching between different versions
   - method for swapping different versions of data stream programs with
     1. minimal effect on their runtime performance
     2. without loss of integrity of data.
Background – Resource Limited Stream Compute Clusters

- A compute cluster with fixed set of nodes which cannot be expanded dynamically.
- Our emphasize is on the number of nodes
  - does not matter how much resources a node has if it crashes suddenly.
- Widely used by financial, academic, health care institutions.
- Takes considerable amount of time for crash recovery.
System S is an operator-based, large-scale, distributed data stream processing middleware.
System S uses an operator-based programming language called SPADE for defining Data flow graphs.
Data flow graphs consisting of operators and streams.

Source: IBM InfoSphere Streams Harnessing Data in Motion
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

EXAMPLE SPADE APPLICATION

[Application]
regex

[Program]
stream Source1(dateTime:String) := Source()"file:///SourceData.dat",nodelays, csvformat{ }

stream Regex1(dateTime:StringList) := Functor(Source1)[]
{ regexMatch(dateTime, "([0-9]*)([0-9]*)([0-9]*).*)" } 

stream Regex2(date:String, ttime:String, seq:Long) := Functor(Regex1)[]
{ dateTime[3]+"-"+
select(toInteger(dateTime[2])-1, "JAN", "FEB", "MAR", "APR", "MAY", "JUN",
"JUL", "AUG", "SEP", "OCT", "NOV"
"DEC")+
"-"+dateTime[1], dateTime[4], seqNum()
}

stream Regex3(date:String, ttime:String, seq:Long) := Functor(Regex2)[]
{ date, regexReplace(ttime,"00","22",true), seq }

Nil := Sink(Regex3)"file:///SinkData.dat"{}
Automatic Sample Program Generation with Hirundo

- Sample Programs (S)
  - Generate similar programs with different flow graph layouts
  - Analyze Program Structure
  - Synthesize sample data

- Select programs $U_1$
  - Compile programs $U_1$ in parallel
  - Run each program in $U_1$ for $W_t$ time
  - Calculate performance details, rank programs

- Select programs $U_2$
  - Select programs $R_1$
  - Compile programs $U_3$ parallel
  - Run each program in $U_3$ for $W_t$ time

- A ranked list of programs (R2)

---


DASFAA 2013

Stream Program Performance Recovery in Resource Limited Clusters
Hirundo introduces techniques for automatically generate data flow graphs with varied performance characteristics for a given stream application.

Hirundo analyzes and identifies the structure of a stream program and transforms the program’s dataflow graph into many different versions.

Two example applications – Regex, Apnoea
Experiment Applications

(a) Regex - Data transformation

- **Date & Time values**
  - **S**: Injects input tuples
  - **F1**: Filters the date time tuples it receives
  - **F2**: Conducts date time format conversion
  - **F3**: Replaces “00” with “22”
  - **SI**: Stores the result tuples

(b) VWAP - Volume Weighted Average Price

- **Trades & Quotes**
  - **S**: Injects input tuples
  - **F1**: Filters tuples for valid records
  - **AG**: Finds max/min of the trading prices
  - **F2**: Calculates VWAP
  - **SI**: Stores the result tuples

DASFAA 2013
(c) Apnoea - Detect clinically significant events of premature infants.
Program Transformation – Tri-Operator Transformation

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

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

Introduction > Research Problem > Background > Related Work > Methodology > Implementation > Evaluation > Discussion > Conclusion
Example data flow graph transformation on Hirundo
Example data flow graph transformation on Hirundo (Cont.)

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

Candidate ray identification

Tri-OP Transformation

OPB1 (Transformed operator block)

Operator block fusion
Program Transformation – Tri-Operator Transformation

Input Program

Identify Program Structure

G – input graph structure

Tri-Operator Transform

Transformed operator blocks

Weld the transformed operator blocks

Sample Programs

Traverse the flow graph three operator blocks at time

Apply Transformation and create operator block

Introduction > Research Problem > Background > Related Work > Methodology > Implementation > Evaluation > Discussion > Conclusion
Related Work

- Active Standby (AS) [9]
- Passive Standby (PS) [7]
  - Both approaches involve replication


Hadoop Streaming [16]
- UNIX standard streams are used to interface between Hadoop and user programs.

In-situ MapReduce [14]
- Employ load shedding techniques


Related Work (Cont.)

- Stream Graph Partitioning [12][17]
  - Stream program performance optimization at stream processing environment’s level


## Approach for performance maintenance

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_N$</td>
<td>Stream processing environment with $N$ nodes. ($N &gt; 1$)</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of input stream programs. ($</td>
</tr>
<tr>
<td>$u$</td>
<td>Performance margin for stream processing environment $E$. ($u \in {0, ..., 100}$, $u \in \mathbb{R}^+$). This value is calculated by Albatross using current performance information of $E$.</td>
</tr>
<tr>
<td>$m$</td>
<td>A performance window set by user. ($m \in {0, ..., 100}$, $m \in \mathbb{R}^+$)</td>
</tr>
<tr>
<td>$M_i$</td>
<td>Input stream program priority margin. ($\forall i, j$ where $i, j \in \mathbb{N}$, $M_i, M_j \in \mathbb{R}^+$, $i, j \in {0, ..., (n-1)}$, $i, j \in \mathbb{N}$, $M_i \neq M_j$, $\sum_{i=0}^{n-1} M_i = 100$). Priority margin is used for ranking input programs based on their importance. This value needs to be specified by the user prior to running Albatross.</td>
</tr>
<tr>
<td>$r$</td>
<td>Calibration run. If $r = 0$ it is a normal mode run. When Albatross is deployed in its usual operation, it is called normal mode run.</td>
</tr>
<tr>
<td>$P^{(r)}_S$</td>
<td>Sample program set generated for $S$ during calibration run $r$. ($r \in \mathbb{N}$, $r \neq 0$). A calibration is a running of entire sample program space with the intention of obtaining the performance information.</td>
</tr>
<tr>
<td>$P^{(0)}_S$</td>
<td>Sample program set generated for $S$ during normal mode run. Here $0$ in $P^{(0)}_S$ represents a normal mode run of Albatross.</td>
</tr>
<tr>
<td>$X_N$</td>
<td>Selected sample program set. ($X_{N_i} \in P^{(0)}_{S_i}$, $</td>
</tr>
<tr>
<td>$\text{perf}(x)$</td>
<td>Predicate for performance (e.g., throughput, elapsed time, etc.) of sample program $x$</td>
</tr>
</tbody>
</table>
Approach for performance maintenance (Cont.)

Stream Processing environment

Set of sample programs generated during past calibration sessions

\[ P_s^{(r)} \]

Input programs each with priority Margin \( M_i \)

\[ S \]

\[ P_{si}^{(o)} \text{ for each } Si (i \in \{0, ..., (n-1)\}, i \in N) \]

Instantiate \( E_N \)

Select \( X_N \) Sample programs

Compile \( X_N \) Sample programs

Start \( X_N \)
Stream Program Performance Recovery in Resource Limited Clusters

Approach for performance maintenance (Cont.)

- Failure in runtime?
  - Yes: Shutdown $E_N$
  - No: Instantiate $E_{N'}$
    - Select $X_{N'}$ Sample programs if $M_i \geq (u+m)$
    - Yes: Compile $X_{N'}$ Sample programs
    - No: Cancel $X_{N'}$ Sample programs
    - Start $X_N$ and $X_{N'}$ in $E_{N'}$ for $W_t$ time
    - Yes: Cancel $X_N$ Sample programs
    - No: Cancel $X_N$ Sample programs

- Completed rescheduling?
  - Yes: perf($X_{Ni}$) > perf($X_{Ni'}$)
    - Yes: Cancel $X_N$ Sample programs
    - No: Cancel $X_N$ Sample programs
  - No: perf($X_{Ni}$) > perf($X_{Ni'}$)
    - Yes: Cancel $X_N$ Sample programs
    - No: Cancel $X_N$ Sample programs
Algorithm 1: Selection of Replacement Sample Program

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Get sample program performance details */</td>
<td>21: end for</td>
</tr>
<tr>
<td>/* Aggregate throughput information for each label */</td>
<td>22: sortByRangeAsc(labelPerfStat)</td>
</tr>
<tr>
<td>/* Find range, average, */</td>
<td>23: selectedLabel ← Ø</td>
</tr>
<tr>
<td>/* min, max throughput */</td>
<td>/* Select sample program with higher average */</td>
</tr>
<tr>
<td>/* values for each group */</td>
<td>/* throughput compared to the Input app */</td>
</tr>
<tr>
<td>/* If could not find a suitable app, */</td>
<td>24: for all label in labelPerfStat do:</td>
</tr>
<tr>
<td>/* select using average throughput */</td>
<td>25: if label.min &gt; inputStat.max then</td>
</tr>
<tr>
<td>/* If no suitable app found then return the input app */</td>
<td>26: selectedLabel ← label</td>
</tr>
<tr>
<td>/* Sort labels ascending order using range values */</td>
<td>27: break</td>
</tr>
<tr>
<td>/* end if */</td>
<td>28: end if</td>
</tr>
<tr>
<td>/* end for */</td>
<td>29: end for</td>
</tr>
<tr>
<td>/* end if */</td>
<td>30: if selectedLabel = Ø then</td>
</tr>
<tr>
<td>/* end for */</td>
<td>/* If could not find a suitable app, */</td>
</tr>
<tr>
<td>/* end if */</td>
<td>/* select using average throughput */</td>
</tr>
<tr>
<td>/* end for */</td>
<td>31: for all label in labelPerfStat do:</td>
</tr>
<tr>
<td>/* end if */</td>
<td>32: if label.average &gt; inputStat.average then</td>
</tr>
<tr>
<td>/* end for */</td>
<td>33: selectedLabel ← label</td>
</tr>
<tr>
<td>/* end if */</td>
<td>34: break</td>
</tr>
<tr>
<td>/* end if */</td>
<td>35: end if</td>
</tr>
<tr>
<td>/* end for */</td>
<td>36: end for</td>
</tr>
<tr>
<td>/* end if */</td>
<td>37: end if</td>
</tr>
<tr>
<td>/* end if */</td>
<td>38: if selectedLabel = Ø then</td>
</tr>
<tr>
<td>/* end if */</td>
<td>39: selectedLabel ← appname</td>
</tr>
<tr>
<td>/* end if */</td>
<td>40: end if</td>
</tr>
</tbody>
</table>

**Input**: Input application’s name (appname), structure of the input stream application (G), transformation depth (d), number of nodes (nodes)

**Output**: Replacement sample program (selectedLabel)

**Methodology**

This algorithm is designed to select a suitable replacement sample program for a given input application. It involves several steps:

1. **Step 1**: Retrieve the latest three optrun IDs for the input application.
2. **Step 2**: For each optrun, get the performance details and aggregate the throughput information for each label.
3. **Step 3**: For each label, determine the range, average, minimum, and maximum throughput values.
4. Sort the labels in ascending order using the range values.
5. Select the sample program with a higher average throughput compared to the input application.
6. If no suitable app is found, select using average throughput.
7. Return the selected label.

**Related Work**

This work builds upon previous research in stream program performance recovery, focusing on resource-constrained environments.

**Conclusion**

The proposed method significantly improves the selection of replacement sample programs, enhancing the performance recovery process in resource-limited clusters.
Program Switching Model

- Used to remove low priority programs during drastic node failures to free capacity on the remaining nodes

\[ p(y) = \frac{\text{(current level of resource } y)}{\text{(initial level of resource } y)} \]

\[ \Phi(x) = \begin{cases} 1 & [\text{Mi} - ((p(R) \times p(N) \times 100) + m)] > 0 \\ 0 & \text{otherwise} \end{cases} \]
Implementatio
Implementation (Cont.)

- **Albatross**
  - targeted for System S and SPADE
  - Does not depend on shared file systems
  - Utilizes an SQLite database to store information

- **StreamFarm**
  - Stream workload synthesis tool

- **Runtime Orchestrator module**
  - Monitors health of the cluster using periodic heartbeat messages.

- **Gateway**
  - All the data streams that go into/come out of the sample applications travel through the Gateway module.
  - Used to measure the data rates of the streams.
  - Conducts the tuple buffering process during crash recovery

- **Sample Program Version Selector**
  - Does switching between different versions of sample programs
Assumptions for fault tolerance model

• The stream data is not bigger than what could be buffered in the memory of a node
• The node hosting Albatross’s gateway component does not crash.
## Experimental Evaluation - Environment

<table>
<thead>
<tr>
<th>Cluster</th>
<th>A (For System S and Albatross) - 8 nodes</th>
<th>B (For StreamFarm) – 5 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Quad-Core AMD Phenom(tm) 9850 processor</td>
<td>dual core AMD Opteron(tm) Processor 242</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512KB</td>
<td>1024KB</td>
</tr>
<tr>
<td>RAM</td>
<td>8GB</td>
<td>8GB</td>
</tr>
<tr>
<td>HDD</td>
<td>160GB</td>
<td>250</td>
</tr>
<tr>
<td>Network</td>
<td></td>
<td>1 Gigabit Ethernet</td>
</tr>
<tr>
<td>OS</td>
<td></td>
<td>Linux Cent OS release 5.4</td>
</tr>
<tr>
<td>System S</td>
<td>IBM Infosphere Streams Version 1.2</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td>SQLite version 3</td>
</tr>
<tr>
<td>JRE</td>
<td>1.6.0 (with JVM heap sizes -Xms2048m and -Xmx2048m)</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation 1 - Evaluation of Stream Program Performance Variation

- Modified Albatross not to respond to crash failures and ran two sample applications of Regex and VWAP on Cluster A.
- Then we crashed two nodes which left only 6 operational nodes.

Throughput (B/s) vs. Time

Change of performance for Regex and VWAP applications when two nodes were crashed in a cluster of 8 nodes

- 8 Nodes
- 6 Nodes
- Two Nodes Crashed

DASFAA 2013
Stream Program Performance Recovery in Resource Limited Clusters
Ran Albatross in calibration mode. Used VWAP application in cluster A with 8 nodes.

Throughput of sample programs generated for VWAP application over Six Calibration runs (d = 8, Nodes = 6)
VWAP, Regex, and Apnoea applications were calibrated three times prior the experiment of each node configuration.

Albatross and System S were run in the cluster A, while StreamFarm was ran on cluster B to avoid potential interferences.

In the middle of the experiment, two nodes were crashed.
Performance recovery during a node crash of VWAP

Performance recovery during two consecutive node crashes of Regex
Evaluation 3 - Evaluation of performance recovery process (Cont.)

Performance Recovery process for Apnoea application during two consecutive Node Crashes

- Two Nodes Crashed
- 6 Nodes
- 8 Nodes
- Recovery Completed
- 4 Nodes
- Two Nodes Crashed
- Recovery Completed

(b) Time (Time of the day)

DASFAA 2013
Discussion

- Albatross is able to restore the operations back to the normal level
  - VWAP on 6 nodes, Regex on 4 nodes, Apnoea on 4 nodes

- Albatross was able to restore performance even when half of the nodes in the cluster were crashed.
  - Apnoea and Regex each on 4 nodes.
Limitations

- Albatross’s grammar represents a subset of the constructs of the SPADE language.
- Large number of applications each demanding maintenance of higher performance.
- Data rate might not produce the correct picture of performance of certain stream applications.
- It takes considerable amount of time to restore the normal operations of all the stream jobs.
  - Most of this time is spent on orchestrating the System S runtime.
- Possibility of single point of failure
Conclusions

- We introduced a technique for maintaining performance during crash failures of stream computing systems.
  - widely applicable to resource limited stream processing clusters
  - based on automatic code generation.
- Albatross is a middleware that monitors the status of the node clusters and strives to maintain the performance via swapping the sample programs generated for each input program.
- We observed that Albatross maintains the same performance of the Regex and VWAP stream jobs despite loss of nodes from the stream processing environment.
Future Work

- Create a sophisticated scheduling algorithm for Albatross’s stream job control process.
- Study techniques of predicting sample program performance to improve Albatross’s calibration process.
- Eliminate the single point of failure of Albatross middleware.