Towards self-managed, re-configurable streaming dataflow systems

Vasia Kalavri
kalavriv@inf.ethz.ch
THE DATAFLOW MODEL

- Computations as **Directed Acyclic Graphs** (DAGs)
  - nodes are operators and edges are data channels
  - operators can accumulate state, have multiple inputs, express event-time custom window-based logic
- Transformations are **data-parallel**
  - distributed workers (threads) execute one parallel instance of one of more operators on *disjoint data partitions*
- Queries are **long-running**
  - input streams are potentially *unbounded*
  - results are *continuously produced*
DATAFLOW COMPUTATIONS

Twitter source → Extract hashtags → Count topics → Trends sink

Logic

Query Plan

Deployment
DATAFLOW WORKER ACTIVITIES

- Parallel workers perform activities
  - receive message
  - deserialize
  - process
  - serialize
  - send message
- Or are **waiting** for
  - input (nothing in the buffer)
  - output (no write buffer available)
STRYMON: ONLINE DATACENTER ANALYTICS AND MANAGEMENT

Datacenter

event streams

traces, configuration, topology updates, ...

Strymon

Datacenter model

policy enforcement, what-if scenarios, ...

queries, complex analytics, simulations, ...

strymon.systems.ethz.ch
RECONFIGURABLE STREAM PROCESSING

streaming application

input streams

output streams

external systems

trace streams

state backend

DS2 auto-scaling controller

scaling decisions

input streams

output streams

external systems

trace streams

state backend

DS2 auto-scaling controller

scaling decisions

streaming application

RECONFIGURABLE STREAM PROCESSING

DS2 auto-scaling controller

OSDI’18

SnailTrail profiler

NSDI’18

online performance metrics

performance summaries

OSDI’18

streaming application

RECONFIGURABLE STREAM PROCESSING

DS2 auto-scaling controller

OSDI’18
Snailtrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows

Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, Timothy Roscoe

Systems Group, ETH Zurich

nsdi'18
PERFORMANCE TROUBLESHOOTING

- long-running, dynamic workloads
- many tasks, activities, operators, dependencies
- conventional profiling tools provide *aggregate* information

Dataflow graph

Aggregate data exchange

Duratio
OPTIMIZING ACTIVITY DURATION

Processing is the most *time-consuming* activity
What if we optimize it?
OPTIMIZING ACTIVITY DURATION

W1

waiting

W2

serializatio

W3

deserialization
OPTIMIZING ACTIVITY DURATION

W1

waiting

W2

waiting increases

serializatio

W3

deserialization

No benefit
CRITICAL PATH ANALYSIS
THE PROGRAM ACTIVITY GRAPH (PAG)

Nodes are timestamped events: start or end of a worker activity.
THE PROGRAM ACTIVITY GRAPH (PAG)

Edges represent activities annotated with a type and duration

(u, v) = {
    type: serialization
    duration: 1
}

15
The **longest** path in the execution history (not considering waiting activities)
CRITICAL PATH

Reduced execution
POST-MORTEM CRITICAL PATH ANALYSIS

1. Collect traces during execution

2. Analyze traces offline

job start → profiler → job end → profiler → database → analyzer → performance summaries
ONLINE CRITICAL PATH ANALYSIS
ONLINE ANALYSIS OF TRACE SNAPSHOTS

- **Input stream**
- **Output stream**
- **Periodic snapshot**
- **Trace snapshot**
- **Performance summaries stream**
- **Analyzer**
All paths have the same length: $t_e - t_s$
All paths have the same length: $t_e - t_s$
All paths have the same length: $t_e - t_s$
- All paths have the same length: \( t_e - t_s \)
- Choosing a random path might miss critical activities
All paths have the same length: $t_e - t_s$

Choosing a random path might miss critical activities
- All paths have the same length: $t_e - t_s$
- Choosing a random path might miss critical activities
- Enumerating all paths is impractical
All paths are potentially part of the **evolving** critical path

How to **rank** activities with regard to **criticality**?

**Intuition:** the more paths an activity appears on the more probable it is that this activity is critical
CRITICAL PARTICIPATION (CP METRIC)

An estimation of the activity’s participation in the critical path

**Centrality**: the number of paths this activity appears on

**Activity duration**: edge weight

\[
CP_a = \frac{C(a) \cdot d_w}{N(t_e - t_s)}
\]

Can be computed without path enumeration!
ONLINE PERFORMANCE ANALYSIS WITH SNAILTRAIL
reference application

Profiling

Trace generation

SnailTrail

Timely

Trace ingestion

PAG construction

CP computation and activity ranking

CP-based performance summaries

Apache Flink, Apache Spark, TensorFlow, Heron, Timely Dataflow, ...
Activity Summary

- Which activity type is a bottleneck?
Apache Flink: Dhalion WordCount Benchmark, 4 workers, 1s snapshots

Optimize serialization
SNAILTRAIL CP-BASED SUMMARIES

- **Activity** Summary
  - *which activity type* is a bottleneck?

- **Straggler** Summary
  - *which worker* is a bottleneck?
Straggler Summary

Apache Flink: Dhalion WordCount Benchmark, 4 workers, 1s snapshots

Steal work from W1
SNAILTRAIL CP-BASED SUMMARIES

- Activity Summary
  - which activity type is a bottleneck?

- Straggler Summary
  - which worker is a bottleneck?

- Operator Summary
  - which operator is a bottleneck?
Operator Summary

Increase flatmap's parallelism

Apache Flink: Dhalion WordCount Benchmark, 10 workers, 1s snapshots
SNAILTRAIL CP-BASED SUMMARIES

- **Activity Summary**
  - *which activity type* is a bottleneck?

- **Straggler Summary**
  - *which worker* is a bottleneck?

- **Operator Summary**
  - *which operator* is a bottleneck?

- **Communication Summary**
  - *which communication channels* are bottlenecks?
Collocate worker 5 with 2, 9, 10, 11
github.com/strymon-system/snailtrail
Three steps is all you need:
fast, accurate, automatic scaling decisions
for distributed streaming dataflows

Vasiliki Kalavri, John Liagouris, Moritz Hoffmann,
Desislava Dimitrova, Matthew Forshaw, Timothy Roscoe

Systems Group, ETH Zurich
Any streaming job will inevitably become over- or under-provisioned in the future
THE SCALING PROBLEM

Given a logical dataflow with sources $S_1, S_2, \ldots S_n$ and rates $\lambda_1, \lambda_2, \ldots \lambda_n$ identify the minimum parallelism $\pi_i$ per operator $i$, such that the physical dataflow can sustain all source rates.

logical dataflow

physical dataflow

Given a logical dataflow with sources $S_1, S_2, \ldots S_n$ and rates $\lambda_1, \lambda_2, \ldots \lambda_n$ identify the minimum parallelism $\pi_i$ per operator $i$, such that the physical dataflow can sustain all source rates.
AUTOMATIC SCALING OVERVIEW

- detect symptoms
- metrics
- scaling controller
- policy
- decide whether to scale
- decide how much to scale
- scaling action
EXISTING SCALING MODELS: QUEUING THEORY

- **Metrics**
  - *service* time and *waiting* time per *tuple* and per task
  - total time spent processing a tuple and all its *derived results*

- **Policy**
  - each operator as a single-server queuing system
  - generalized Jackson networks

- **Action**
  - predictive, at-once for all operators

Too fine-grained, **impractical** for high-rate streams

Sampling **degrades** accuracy

Simplified models make **strong assumptions**

**Unsuitable** for complex operators, e.g. sliding windows, joins
EXISTING SCALING MODELS: CONTROL THEORY

- **Metrics**
  - input and output signals
  - delay of tuples that have just entered the system

- **Policy**
  - dataflow as a black-box
  - SISO models - MIMO too complex

- **Action**
  - predictive, dataflow-wide

The output signal is the **delay** time.

Performance depends on **parameter selection**, e.g. poles placement, sampling period, damping.

Cannot identify individual **bottlenecks** neither model 2-input operators.
EXISTING SCALING MODELS: RULES AND THRESHOLDS

- **Metrics**
  - externally observed coarse-grained and aggregates
  - CPU utilization, throughput, back-pressure signal

- **Policy**
  - rule-based
  - *If CPU utilization > 70% and back-pressure then scale up*

- **Action**
  - speculative, one operator at-a-time

- **Noisy, sensitive to interference, misleading**
- **Easy-to-obtain**
- **Sensitive to thresholds and require **manual tuning****
- **Oscillations, slow convergence, black-listing**
effect of Dhalion’s scaling actions
in an initially under-provisioned wordcount dataflow
Which operator is the bottleneck?

What if we scale $o_1 \times 4$?

How much to scale $o_2$?
$o_1$ is the bottleneck

2 $o_2$ instances can keep up with the rate of 4 $o_1$ instances.

true rate = 200 rec/s
THE DS2 MODEL: INSTRUMENTATION AND DATAFLOW DEPENDENCIES

- Collect metrics per configurable `observation` window $W$
  - `activity durations` per worker
  - records processed $R_{prc}$ and records pushed to output $R_{psd}$
- Capture `dependencies` through the dataflow graph itself
  - assign an increasing `sequential id` to all operators in topological order, starting from the sources
  - represent as an `adjacency` matrix $A$
    - $A_{ij} = 1$ iff operator $i$ is upstream neighbor of $j$
THE DS2 MODEL: USEFUL TIME

Useful time \( W_u \)

The time spent by an operator instance in **deserialization**, **processing**, and **serialization** activities.

- excludes any time spent waiting on input or on output
- amounts to the time an operator instance runs for if executed in an *ideal* setting
- when there is no waiting the useful time is equal to the **observed time**
THE DS2 MODEL: **TRUE RATES**

**True processing / output rates**

\[ \lambda_p = \frac{R_{prc}}{W_u} \]

\[ \lambda_o = \frac{R_{psd}}{W_u} \]

**Aggregated true processing / output rates**

\[ o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \]

\[ o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k \]
**THE DS2 MODEL: OPTIMAL PARALLELISM**

Optimal parallelism per operator

\[ \pi_i = \left[ \sum_{\forall j: j < i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left( \frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right], n \leq i < m \]
Recursively computed as:

\[
o_j[\lambda_o]^* = \begin{cases} 
    o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \leq j < n \\
    \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{u : u < j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m
\end{cases}
\]

It can be computed for all operators by traversing the dataflow from left to right once.

True output rate of source j
DS2 MODEL PROPERTIES

If operator scaling is **linear**, then:

- scale-up does not cause over-provisioning (**no overshoot**)
- scale-down does not cause under-provisioning (**no undershoot**)

Ideal scaling acts as an **upper bound** when scaling up and as a **lower bound** when scaling down:

- DS2 will **converge monotonically** to the target rate
If the actual scaling is **linear**, convergence takes **one** step.
CONVERGENCE STEPS

when the actual scaling is sub-linear, convergence takes more than one steps
In our experiments, DS2 took **up to three steps** to converge for complex queries.
DS2 operates online in a reactive setting

Apache Flink

Instrumented stream processor

Scaling Manager

Scaling Policy

Metrics Repository

Timely dataflow

Apache Flink

Heron

re-scale job

report
metrics

monitor

invoke

pull
metrics

decision
DS2 converges in a **single step** for both operators.

DS2 converges in **60s**, i.e. as soon as it receives the Heron metrics.

Dhalion scales one operator at a time, resulting to a total of **six steps**.

Dhalion converges in **2000s**.

Target rate: **16,700 rec/s**
DS2 ON APACHE FLINK

Apache Flink savepoint and reconfiguration takes ~30s

DS2 converges in two steps for both operators

DS2 reacts 3s after the target rate has changed

Target rate: 2,000,000 rec/s
### CONVERGENCE - NEXMARK

<table>
<thead>
<tr>
<th>initial parallelism</th>
<th>Q1: flatmap</th>
<th>Q2: filter</th>
<th>Q3: incremental join</th>
<th>Q5: tumbling window join</th>
<th>Q8: sliding window</th>
<th>Q11: session window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scale-up</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 =&gt;</td>
<td>12 =&gt; 16</td>
<td>11 =&gt; 13 =&gt; 14</td>
<td>16 =&gt; 20</td>
<td>14 =&gt; 15 =&gt; 16</td>
<td>10</td>
<td>12 =&gt; 22 =&gt; 28</td>
</tr>
<tr>
<td>12 =&gt;</td>
<td>16</td>
<td>14</td>
<td>18 =&gt; 20</td>
<td>16</td>
<td>10</td>
<td>22 =&gt; 28</td>
</tr>
<tr>
<td>16 =&gt;</td>
<td>16</td>
<td>12 =&gt; 14</td>
<td>20</td>
<td>16</td>
<td>8 =&gt; 10</td>
<td>26 =&gt; 28</td>
</tr>
<tr>
<td>20 =&gt;</td>
<td>16</td>
<td>13 =&gt; 14</td>
<td>20</td>
<td>14 =&gt; 16</td>
<td>8 =&gt; 10</td>
<td>28</td>
</tr>
<tr>
<td>24 =&gt;</td>
<td>16</td>
<td>14</td>
<td>20</td>
<td>14 =&gt; 16</td>
<td>8 =&gt; 10</td>
<td>28</td>
</tr>
<tr>
<td><strong>scale-down</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28 =&gt;</td>
<td>16</td>
<td>14</td>
<td>20</td>
<td>13 =&gt; 16</td>
<td>8 =&gt; 10</td>
<td>28</td>
</tr>
</tbody>
</table>

*at most 3 steps*

*a single step* for many queries and initial configurations

=> : scaling action
no oscillations
true rates as bounds to avoid over/under-shoot
fast convergence
github.com/strymon-system/ds2
streaming (Strymon) application

**Megaphone: Live state migration**

- Progressive, fine-grained state copies
- No stop-and-restart, no downtime
- Minimal latency spikes

**Strymon integration with FASTER**

- New, fast key-value store from Microsoft research
- Support for larger-than-memory state
- Fast in-place updates

ongoing work
Towards self-managed, re-configurable streaming dataflow systems

Vasia Kalavri
kalavriv@inf.ethz.ch