Towards a Universal Stream Processing System

Robert Soulé
Cornell University
Data Crisis

2.5 quintillion bytes every day

90% of the world’s data was created in the last 2 years
How Big is Your Data?

This talk will focus on infinite data

Biggest data set that you can have:

\[ \text{kB} < \ldots < \text{PB} < \text{EB} < \infty \]
Sources of Infinite Data
Sources of Infinite Data

Justin Bieber
@justinbieber
#BELIEVE is on ITUNES and in STORES WORLDWIDE! – SO MUCH LOVE FOR THE

40,077,846 followers
122,642 following
22,353 tweets
Challenges at All Levels of the Software Stack

- Systems challenges at all levels
- Leverage formal techniques from programming languages to build correct and efficient systems
- This talk will focus on two levels
Step 1
Focus on Application Level

How to simplify the development of languages
Stream Processing

- Process data as a sequence of computational steps
  - Data flows in as a stream
  - Each processing step is an operator

- Natural for processing when data does not fit in memory
  - MapReduce is a special case of stream processing
  - Input data set is finite
  - There are only four operators map/split/merge/reduce
Languages are Crucial To Streaming

- Developers are already familiar with a particular syntax
  - Wall Street IT already knows SQL, so CQL is natural

- Compilers can statically enforce safety properties
  - Video playback needs fixed data transfer rate, StreamIt ensures it

- Languages hide system complexity
  - Sawzall expresses business logic, MapReduce handles distribution
Developing Languages Is Too Much Work
Developing Languages Is Too Much Work

Design for an application domain
Developing Languages Is Too Much Work
Developing Languages Is Too Much Work

- Apply optimizations
- Design for an application domain
- Build custom runtime

- Optimizer
- Runtime
- StreamIt
- Optimizer
- Runtime
- Sawzall
- Optimizer

Thursday, June 20, 13
Developing Languages Is Too Much Work

- **Apply optimizations**
  - Optimizer
  - Runtime

- **Design for an application domain**
  - Optimizer
  - Runtime

- **Build custom runtime**
  - Optimizer
  - Runtime
An IL separates language front-ends from optimizers and target

- Allows languages to reuse optimizations
- Simplifies target for language translation
- Traditional examples: UNCOL, CLI, JVM

There does not exist an IL for streaming

- Optimizations are different
- Need to reason across machines
River: an IL for Streaming

- Supports diverse streaming languages
- Enables critical streaming optimizations
- Encourages reuse and portability
IL Needs To Be Correct

- “Hour-glass” interface
- Target of all languages and optimizations
- Start with a formal model
  - Lightweight calculus and semantics
  - Prove correctness of translations
  - Prove correctness of optimizations
Elements of a Streaming App
Elements of a Streaming App
Elements of a Streaming App

Not all operators have state
Elements of a Streaming App

Operators may share state
Calculus Syntax

```
(volume, $total) ← Sum(trades, $total)
```
Function Environment

F: The function implementations
Queue Store

Q: The contents of the queues
Variable Store

V: The contents of the variables
Operational Semantics

\[ F \vdash \langle Q, V \rangle \rightarrow \langle Q', V' \rangle \]
**Complete Calculus**

**Syntax:**
- \( P_b ::= \text{out \ in} \ \overline{\text{op}} \quad \text{Brooklet program} \)
- \( \text{out ::= output} \ \overline{q} \; ; \quad \text{Output declaration} \)
- \( \text{in ::= input} \ \overline{q} \; ; \quad \text{Input declaration} \)
- \( \text{op ::= (} \ \overline{q}, \overline{v} \ \text{)} \leftarrow f(\ \overline{q}, \overline{v}) \quad \text{Operator} \)
- \( q ::= \text{id} \quad \text{Queue identifier} \)
- \( v ::= \$ \text{id} \quad \text{Variable identifier} \)
- \( f ::= \text{id} \quad \text{Function identifier} \)

**Example:** IBM market maker.

```plaintext```
output result;
input bids, asks;
(IBMbids) ← SelectIBM(bids);
(IBMasks) ← SelectIBM(asks);
($lastAsk) ← Window(IBMasks);
(IBMsales) ← SaleJoin(IBMbids,$lastAsk);
(result,$cnt) ← Count(IBMsales,$cnt);
```

**Semantics:** \( F_b \vdash \langle V,Q \rangle \longrightarrow \langle V',Q' \rangle \)

\[
d, b = Q(q_i) 
\]

\[
op = (_\_,\_ \leftarrow f(\overline{q},\overline{v})
\]

\[
(\overline{b}',\overline{d}') = F_b(f)(d,i,V(\overline{v}))
\]

\[
V' = \text{update}_V(op,V,\overline{d}')
\]

\[
Q' = \text{update}_Q(op,Q,q_i,\overline{b}')
\]

(E-FIREQUEUE)

```
\[
op = (_\_,\overline{v} \leftarrow f(\_\_,\_)
\]

\[
\text{update}_V(op,V,\overline{d}) = [\overline{v} \mapsto \overline{d}]V
\]

(E-UPDATEV)

```

\[
d_f, b_f = Q(q_f)
\]

\[
Q' = [q_f \mapsto b_f]Q
\]

\[
Q'' = [\forall q_i \in \overline{q} : q_i \mapsto Q(q_i), b_i]Q'
\]

(updateQ(op,Q,q_f,$b) = Q'')

(E-UPDATEQ)
River as the Target
For Diverse Languages

We built Source-to-River translators for 3 languages

CQL ↘ StreamIt ↘ Sawzall

River

Optimizer

System S
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Wrap original operators in higher-order functions

Functions → < Queues , Variables >
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Wrap original operators in higher-order functions
- Expose implicit and explicit state
- Make queues explicit

Functions < Queues , Variables >
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Make queues explicit
- Make state explicit

Wrap original operators in higher-order functions

Functions < Queues , Variables >
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Make queues explicit
- Make state explicit

- Re-use operator implementations
- Wrap original operators in higher-order functions

Functions < Queue, Variables >
Benefits of Formalism

River translations are provably correct

Results under CQL and StreamIt semantics are the same as the results under River semantics after translation

First formal semantics for Sawzall
River Optimizations

- Decouples languages and optimizers
- Increases portability and reuse
- Optimizations are provably correct
Operator Fusion: Eliminate Queueing Delays

Look for connected operators, whose state is not used anywhere else.
Operator Fission: Process More Data in Parallel

Look for stateless operators

before

after
Operator Reordering: Filter Data Early

Look for operators whose read/write sets don’t overlap
Runtime Translation

- CQL
- StreamIt
- Sawzall

River

System S

- River-to-System S translator
- Runtime provides data-transport, process management, distribution, etc.
Immediate Benefits

First distributed CQL implementation for free!
Evaluation

- 4 benchmark applications
  - CQL Linear Road
  - StreamIt FM Radio
  - Sawzall Batch Web Log Analyzer
  - CQL Continuous Web Log Analyzer

- 3 optimizations
  - Placement
  - Fission
  - Fusion
Fuse operators into 4 sets, distributed each set to a machine

4x speedup on 4 machines
Sawzall Fission

- Speedup of 50.32x on 64 cores across 16 machines
- Scales almost linearly
River Summary

River calculus:
- Provides a formal model for reasoning about correctness

River IL:
- Supports a diverse set of languages
- Supports crucial optimizations
- Increases portability and re-use
- Simplifies the development of streaming languages
Step 2
Focus on the Network Level

Application Level
Operating System
Cloud Infrastructure
Virtual Machine
Network Software

How to simplify network management
Networking is Streaming

- Packets are the data streams
- Middleboxes, routers, and switches are operators

Different set of challenges and requirements
- Data is unstructured
- Data rates are orders of magnitude faster
- Contention on shared resources
Recent History

- To configure network
  - SSH to router or switch
  - Define routing rules in DSL
- To process packets
  - Route traffic to middlebox
Network Configuration Challenges

- Campus network has 100s of switches
- Data center network has 1000s of switches
- Difficult to configure
- Difficult to verify
Software Defined Networking (SDN)

- Separate the control plane from the data plane

- Controller software provides:
  - Centralized view of the network
  - Rules to match and forward packets
  - Methods for handling exceptions
SDNs Are A Huge Success

- Embraced by vendors (Cisco, Juniper, VMWare)
- Adopted by customers (Google, Yahoo)
- Dramatically simplifies routing
Functionality Beyond Routing

- **Accounting**: track customer usage inside and outside the network
- **Traffic Filtering**: disable SSH traffic to port 22 by default
- **Provisioning**: share bandwidth fairly amongst tenants
SDNs Provide the Wrong Abstraction

What operators want:  
Drop all HTTP traffic that contains the pattern “evil.com”

What SDN provides:  
Match all HTTP traffic and forward it out port 4
Other SDN Problems

- Hard to scale
- Middlebox is a bottleneck
- Controller is a bottleneck
- Hard to verify
- Multiple pieces of software
- Multiple languages
- Asynchronous events
Merlin Raises Level of Abstraction

- **Policy Language**
  - Declaratively specify global network policy

- **Transformer**
  - Transforms global policies into locally enforceable policies and mediates access to resources

- **Enforcer**
  - Interposes on network traffic to ensure compliance
Start with a Formalism

| Numbers   | n ::= 0 | 1 | 2 | ... |
| Protocol  | r ::= ether | ip | tcp | ... |
| Fields    | f ::= src | dst | ... |
| Time      | t ::= sliding(n) | Sliding |
|           |         | tumbling(n) | Tumbling |
| Locations | l ::= {host} | Location |
|           |         | l \cup l | Union |

| Predicates | p ::= r.f \sim n | Match |
|            |               | Location |
|            | & p_1 p_2 | Conjunction |
|            | p_1 \lor p_2 | Disjunction |
|            | \neg p_1 | Negation |

| Packet Set | s ::= \{p,t\} |
| Aggregates | a ::= counts(s) | Total count |
|            | | Total size |
|            | | Average size |
| Expression | e ::= a | Aggregate |
|            | | Sum |
|            | | Difference |
| Policies   | P ::= e \leq e | Comparison |
|            | | Conjunction |
|            | | Disjunction |
|            | | Negation |
Group packets into sets by header fields or payload, time, and location

Aggregate sets into scalar values

Apply constraints on those values

Policy Language

(bytes

{  
  eth.type ~ 0x0800 and
  ip.proto ~ 0x06 and
  tcp.dstport ~ 80 and
  tumbling 60s and
  location (h1, h2)
}) < 5GB
Policy Language

Group packets into sets by header fields or payload, time, and location

Aggregate sets into scalar values

Apply constraints on those values

(bytes

{eth.type ~ 0x0800 and ip.proto ~ 0x06 and tcp.dstport ~ 80 and tumbling 60s and location (h1, h2)
}) < 5GB

HTTP traffic
Policy Language

Group packets into sets by header fields or payload, time, and location

Aggregate sets into scalar values

Apply constraints on those values

(bytes
  {
    eth.type ~ 0x0800 and
    ip.proto ~ 0x06 and
    tcp.dstport ~ 80 and
    tumbling 60s and
    location (h1, h2)
  }) < 5GB

HTTP traffic

in the last minute

Thursday, June 20, 13
Policy Language

Group packets into sets by header fields or payload, time, and location.

Aggregate sets into scalar values.

Apply constraints on those values.

```
(bytes
    {
        eth.type ~ 0x0800 and
        ip.proto ~ 0x06 and
        tcp.dstport ~ 80 and
        tumbling 60s and
        location (h1, h2)
    }) < 5GB
```
Policy Language

```
(bytes
{
  eth.type ~ 0x0800 and
  ip.proto ~ 0x06 and
  tcp.dstport ~ 80 and
  tumbling 60s and
  location (h1, h2)
}) < 5GB
```

Constraints limit access to shared resources (e.g., bandwidth)

Only exposed mutable state is partition-able
Assign variable to make state explicit:

$x < 5\text{GB}$

Add additional variables to transform global to local:

$h1 + h2 < 5\text{GB}$

Allocate resources:

$h1 + h2 < 5\text{GB}, h1 < 2.5\text{G}, h2 < 2.5\text{G}$

Verify allocations with Z3 constraint solver
Empower Tenants to Refine Policies

- Original allocations might not be optimal
- Tenants may want to modify policies
- Re-negotiation allows tenants to make modifications
  - New policy must imply the old
  - Re-negotiation can happen hierarchically or in peer-to-peer fashion
Scalable Enforcement

- Every end-host runs an interpreter
- Rate-limits or drops packets
- Implements a distributed reference monitor
- Restricted language interface means kernel extension is safe
- Overall correctness theorem proves that global policy is enforced
<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>Track in- and out- of network usage</td>
</tr>
<tr>
<td>Trading</td>
<td>Swap aggregate bandwidth between tenants</td>
</tr>
<tr>
<td>Tiered</td>
<td>Provide different bandwidth caps based on usage</td>
</tr>
<tr>
<td>Isolation</td>
<td>Ensure data from tenants travels different routes</td>
</tr>
</tbody>
</table>
Merlin Allocation is Fast

<table>
<thead>
<tr>
<th>Time</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>99μs</td>
<td>19μs</td>
</tr>
</tbody>
</table>

Average time to update allocations with peer-to-peer scheme
Merlin Has Low Overhead

Most enforcers can saturate 1Gb link
Scalability Experiment

- Implement DPI with Merlin and middlebox with Click modular router
- Measure packet latency as network load increases
## Merlin Reduces Complexity

<table>
<thead>
<tr>
<th>Middlebox</th>
<th>Merlin</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 lines of C++ code</td>
<td>1 line of Merlin policy</td>
</tr>
<tr>
<td>1 line Click script</td>
<td></td>
</tr>
<tr>
<td>5 additional routing rules</td>
<td></td>
</tr>
</tbody>
</table>
Merlin Scales with Network Traffic

Average latency per echo (ms)

Throughput (Mbits/s)

Middlebox
Merlin

Thursday, June 20, 13
Next Steps

- Algorithms for provisioning resources and placing operators
  - Variations on multi-commodity flow, etc.
- Specialization for particular application domains
  - Language support for large-scale machine learning
- Deploy on different architectures
  - Target GPUs for video/graphics processing
Future Work

- Distributed systems of all kinds at all layers of the stack
- Particular focus on:
  - Data processing and storage
  - Network management
  - Integration across layers
- Language support for simplifying system development
Contributions

Data Stream Processing
- SQL to Streaming [VLDB 10]
- Brooklet Calculus [ESOP 10]
- Catalog of Optimizations [ACM CSUR]
- River IL [DEBS 12]
- Distributed state in streaming [DEBS 12]
- Dynamic Optimizations [DEBS 13]
- SPL language [IBM JRD 13]

Content Distribution
- NaKika [NSDI 07]

Distributed Storage
- PADS [NSDI 09]

Software Defined Networks
- Merlin [TR 13]
Summary

- Systems and languages for processing big data are critical
- Scale and performance demands will continue to increase
- Now is an exciting time to work in networking
- In the middle of a revolution in networking
- Use formal models to build systems that are provably correct and have high-performance
http://www.cs.cornell.edu/~soule