Consistency Analysis in Bloom: A CALM and Collected Approach

Peter Alvaro, Neil Conway, William R. Marczak, Joseph M. Hellerstein
Status Quo

Distributed programming: increasingly common
  – Cloud computing, mobile
  – No longer just for the experts!

Distributed programming: still very difficult
  – Parallelism, asynchrony, partial failure, ...
Toward Disorderly Programming

Imperative languages are “ordered by default”

– Data: ordered array of cells
– Computation: ordered sequence of instructions
– This is a poor fit for distributed computing!

Instead: disorderly programming

– Data: unordered collections (sets)
– Computation: unordered bundle of declarative rules
– Ordering constructs provided when needed
– Success stories: MapReduce, parallel SQL
Outline

1. A new language: **Bloom**
   – Disorderly programming for distributed systems

2. A set of analysis tools: **CALM**
   – When is ordering needed in a distributed program?
Bloom

New language for distributed programming

– Prototype implementation as a Ruby DSL
  • Bud: “Bloom Under Development”

– Fully declarative semantics
  • Datalog + state update and asynchronous messages

– Rule-based language + some ideas from OOP
  • Abstract interfaces, modularity, encapsulation
Operational Model

Basic primitives:
- Local computation (Datalog fixpoint)
- State update
- Asynchronous messaging
## Bloom Statements

### Table

<table>
<thead>
<tr>
<th>Collection</th>
<th>Temporal Op</th>
<th>Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>persistent</td>
<td>&lt;= now</td>
</tr>
<tr>
<td>scratch</td>
<td>transient</td>
<td>&lt;+ next</td>
</tr>
<tr>
<td>channel</td>
<td>network transient</td>
<td>&lt;= delete (at next)</td>
</tr>
<tr>
<td>periodic</td>
<td>scheduled transient</td>
<td>~ async</td>
</tr>
<tr>
<td>interface</td>
<td>interface transient</td>
<td></td>
</tr>
</tbody>
</table>

### Temporal Operators

- `<=` now
- `<+` next
- `<-` delete (at next)
- `~` async

### Expressions

- map, flat_map
- reduce, group
- join, outerjoin
- empty?, include?
Abstract Delivery Protocol

module DeliveryProtocol
  include BudModule

  state do
    interface input, :pipe_in,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_sent,
      [:dst, :src, :ident] => [:payload]
  end
end
module BestEffortDelivery
  include DeliveryProtocol

state do
  channel :pipe_chan, [:@dst, :src, :ident] => [:payload]
end

declare
def snd
  pipe_chan ~> pipe_in
end

declare
def done
  pipe_sent <= pipe_in
end
end
module ReliableDelivery
    include DeliveryProtocol
    import BestEffortDelivery => :bed

state do
    table :buf,
        [:dst, :src, :id] => [:payload]
    channel :ack, [:@src, :dst, :id]
    periodic :clock, 2
end

declare
def rcv
    def ack
        ack <= bed.pipe_chan.map
            {|p| [p.src, p.dst, p.id]}
    end

def done
    got_ack = join [ack, buf],
    [ack.id, buf.id]
    msg_done = got_ack.map { |a, b| b}
    pipe_sent <= msg_done
    buf <= msg_done
end
end
end
Outline

1. A new language: Bloom
   – Disorderly programming for distributed systems

2. A set of analysis tools: CALM
   – When is ordering needed in a distributed program?
# Review: Monotonicity

<table>
<thead>
<tr>
<th>Monotonic Logic</th>
<th>Non-Monotonic Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The more you know, The more you know</td>
<td>• New inputs might require retracting previous conclusions</td>
</tr>
<tr>
<td>• e.g., map, filter, join</td>
<td>• To have a “certain” conclusion, must seal input set</td>
</tr>
<tr>
<td></td>
<td>• e.g., aggregation, negation</td>
</tr>
</tbody>
</table>
Monotonicity and Order

Monotonic:

– Output is insensitive to message delivery order

Non-Monotonic:

– Ordering may be needed for consistent results
  • Everyone must agree on the contents of the input set
– Simple analysis: identify **points of order**
  • Non-monotonic operators fed by asynchronous messages
Distributed Consistency

Strong Consistency:
– Enforce total order over messages
  • E.g., using Paxos, Two-Phase Commit, GCS, ...

Loose Consistency:
– Write application to tolerate *any* sequence of message orderings
  • E.g., idempotent, commutative, associative operations
  • Application-specific compensation logic
Practical Implications of CALM

Strong Consistency:

– Identify points of order without coordination logic
– Rewrite program to adjust points of order
  • Push coordination to “cheap” parts of the dataflow
  • Coordination as an optimization problem?

Loose Consistency:

– Track inconsistency “taint” through the program
  • Ensure that inconsistency is resolved by applying compensation logic
Recap

1. Order is a scarce resource!
   – Help the programmer use it wisely
2. What is coordination for?
   – Consistent results from non-monotonic logic
3. Draw user’s attention to points of order
   – Resolve via coordination or compensation
4. Bloom: pragmatic rule-based language for distributed programming
More Info

http://bloom.cs.berkeley.edu

Bud: alpha release shortly

Initial writeups:
  – CIDR’11 (overview, CALM)
  – Datalog 2.0 (declarative semantics)
  – PODS’11 (in submission)
  – PODS’10 keynote (conjectures about CALM)

Thanks to:
MSR, IBM Research, Yahoo! Research, NSF, AFOSR
Applying CALM: Coordination

• Given point of order, can we inject coordination logic automatically?
• Can we recognize equivalent choices for coordination?
  – Coordination strategy as an optimization problem
Applying CALM: Compensation

• Taint tracking: ensure that before output of a point of order is used, it is resolved via compensation logic

• Memories, guesses and apologies (Helland)
  – Common pattern for loose consistency
  – How can we help the programmer?