Failure Diagnosis and Repair in Distributed Systems: Differential Provenance

Holly Zhou, Adrian Lin, Allen Wang, Trey Lachance, Kristina Shia, Peter Zhou, Dibya Bhattacharjee, Shivam Sarodia
Outline

- Differential provenance: problem statement
- Related works / state-of-the-art
- Algorithm
- Evaluation
What is differential provenance?

- **Provenance**: origin, source of an event
- Classical provenance tracks causal connections between network states and state changes
- Differential provenance also performs root cause analysis by reasoning about the differences between two provenance trees
- **Key insight**: Compare a “faulty” event with a reference event that is similar to the faulty event but produces the “correct” outcome
- Can be leveraged to debug complex problems
Example: Provenance

Stop B, 4:26 PM  
Stop Z, 5:01 PM  
Stop A, 4:13 PM  
Dispatched, 4:00 PM

It is 5:05 PM! Why is my bus 5 minutes late?
Example: Differential Provenance

Reference event
Dispatched, 4:00 PM
Stop A, 4:13 PM
Stop B, 4:21 PM
Stop Z, 4:56 PM

Faulty event
Dispatched, 4:00 PM
Stop A, 4:13 PM
Stop B, 4:26 PM
Stop Z, 5:01 PM

It is 5:00 PM! My bus is on time!

It is 5:05 PM! Why is my bus 5 minutes late?
Failure Diagnosis

- **Not** failure prevention or failure tolerance
  - Prevention: System does not allow a failure to happen
  - Tolerance: System continues to function correctly despite a failure
- Diagnosis - happens **after** a (potentially major) failure
- **Why** is this problem important?
  - Cloud outages have high impact and happen in the real world
  - Every second a service is down can mean lost money to the company; violates SLA with customer
  - We want to **quickly and accurately** find the root cause of the problem and fix it
Related Works

- Provenance is a concept actually borrowed from the database community
  - Databases maintain stable states and are able to roll back transactions in the event of failure
- Provenance now has extended applications to different areas like distributed systems, storage systems, operating systems, and mobile platforms
- Must examine and define Network Diagnostics in order to understand the differences between DiffProv and other systems
Related Works

- Provenance is a concept actually borrowed from the database community
  - Databases maintain stable states and are able to roll back transactions in the event of failure
- Provenance now has extended applications to different areas like distributed systems, storage systems, operating systems, and mobile platforms
- Must examine and define Network Diagnostics in order to understand the differences between DiffProv and other systems
Network Diagnostics

- Static analysis (Anteater, NetPlumber)
  - Examine software without execution (debugging source code)
Network Diagnostics

- Static analysis (Anteater, NetPlumber)
  - Examine software without execution (debugging)
Network Diagnostics

- Static analysis (Anteater, NetPlumber)
  - Examine software without execution (debugging source code)
- Dynamic analysis and probing (OFRewind, DEMi, ATPG)
  - Executing software with enough inputs and producing interesting outputs to debug
Network Diagnostics

- **Static analysis (Anteater, NetPlumber)**
  - Examine software without execution (debugging source code)

- **Dynamic analysis and probing (OFRewind, DEMi, ATPG)**
  - Executing software with enough inputs and producing interesting outputs to debug

- **Differences from DiffProv?**
  - No references events
    - Cannot focus on specific differences between “good” and “bad” events
  - Systems specific to the data plane, the part of the network responsible for carrying user traffic
    - Cannot be used to diagnose other distributed systems such as MapReduce
Network Diagnostics

- Static analysis (Anteater, NetPlumber)
  - Examine software without execution (debugging source code)
- Dynamic analysis and probing (OFRewind, DEMi, ATPG)
  - Executing software with enough inputs and producing interesting outputs to debug

Differences from DiffProv?
- Cannot focus on specific differences between “good” and “bad” events
- Systems specific to the data plane, the part of the network responsible for carrying user traffic
- Cannot be used to diagnose other distributed systems such as MapReduce

I JUST DID A MAPREDUCE
SO I GUESS YOU COULD SAY I HAVE A BIG DATA
Network Diagnostics

- Static analysis (Anteater, NetPlumber)
  - Examine software without execution (debugging source code)

- Dynamic analysis and probing (OFRewind, DEMi, ATPG)
  - Executing software with enough inputs and producing interesting outputs to debug

- Differences from DiffProv?
  - No references events
  - Cannot focus on specific differences between “good” and “bad” events
  - Systems specific to the data plane (the part of the network responsible for carrying user traffic)
  - Cannot be used to diagnose other distributed systems such as MapReduce

I JUST DID A MAPREDUCE

SO I GUESS YOU COULD SAY I HAVE A BIG DATA

NOT ONLY DO I ALSO HAVE A BIG DATA

BUT I ALSO DIAGNOSE FAILURES WITH DIFFPROV
Others Systems that Use Reference Events

- Employ combination of statistical analysis and data mining to learn configurations, system invariants (PeerPressure, EnCore, ClearView, Shen)
  - Don’t actually capture causality, or leverage the reference event to reduce possible set of diagnoses
Others Systems that Use Reference Events

- Employ combination of statistical analysis and data mining to learn configurations, system invariants (PeerPressure, EnCore, ClearView, Shen)
  - Don’t actually capture causality, or leverage the reference event to reduce possible set of diagnoses
- Systems that capture causality, but only between systems (Attariyan, Flinn)
  - Can only compare equivalent systems, not events
Others Systems that Use Reference Events

- Employ combination of statistical analysis and data mining to learn configurations, system invariants (PeerPressure, EnCore, ClearView, Shen)
  - Don’t actually capture causality, or leverage the reference event to reduce possible set of diagnoses
- Systems that capture causality, but only between systems (Attariyan, Flinn)
  - Can only compare equivalent systems, not events
- Systems that capture causality of events, but without using provenance (NetMedic)
  - Relies on statistical inference and learning to infer the faulty component
Compared to DiffProv

- Accuracy and Speed
  - Systems relying on statistical analysis (PeerPressure, NetMedic)
    - Do not have the overhead of capturing causality
    - **BUT** may introduce false positives or negatives in their diagnosis
Compared to DiffProv

- Accuracy and Speed
  - Systems relying on statistical analysis (PeerPressure, NetMedic)
    - Do not have the overhead of capturing causality
    - BUT may introduce false positives or negatives in their diagnosis
Compared to DiffProv

- Accuracy and Speed
  - Systems relying on statistical analysis (PeerPressure, NetMedic)
    - Do not have the overhead of capturing causality
    - BUT may introduce false positives or negatives in their diagnosis

- Root Cause
  - Network provenance on single events produce large amounts of data unrelated to the actual root cause
  - State-of-the-art DiffProv combines the use of a reference event as well as provenance to get the root cause of an event failure
How to Model the System? (Setup to DiffProv)

- Instead of differencing the two provenances, find the root cause of the differences
- Use tuples and derivation rules to describe items and interactions inside the network
Intuition (Why this works)

- Each provenance describes one particular path down the provenance graph
- Given an initial state, the sequence of events (path down the graph) is deterministic
Tuples

- Take the form TupleType(x, y, z…)
- Use to describe events and items
- Ex:
  - FlowEntry(5, 1.2.3.4, 8)
    - Event - Packets received on port 5 on IP 1.2.3.4 get sent out through port 8
  - pkt(5, 1.2.3.4)
    - Item - a packet received on port 5 on IP 1.2.3.4
  - GPA(2.3)
    - Item - my GPA
Derivation Rules

- Take the form of $A(...) \ :- \ B(...) \ C(...) \ $ meaning tuple $A$ is derived from the existence of tuple $B$ and $C$.
- Ex:
  - $\text{pkt}(9, 5.6.7.8) \ :- \ \text{pkt}(5, 1.2.3.4) \ \text{FlowEntry}(5, 1.2.3.4, 9)$
    - A packet appears at port 9 of IP 5.6.7.8 if there exists a packet at port 5 of IP 1.2.3.4 and a rule sending packets at port 5 to port 9.
  - $\text{GPA}(2.3) \ :- \ \text{UCBMFET}(\text{true}) \ \text{Nap}(20)$
    - A GPA of 2.3 is derived from browsing memes and napping for 20 hours.
Types of Nodes in our Network System

\text{INSERT}(n, \tau, t), \text{DELETE}(n, \tau, t): \text{ Base tuple } \tau \text{ was inserted (deleted) on node } n \text{ at time } t;

\text{EXIST}(n, \tau, [t_1, t_2]): \text{ Tuple } \tau \text{ existed on node } n \text{ from time } t_1 \text{ to } t_2;

\text{DERIVE}(n, \tau, R, t), \text{UNDERIVE}(n, \tau, R, t): \text{ Tuple } \tau \text{ was derived (underived) via rule } R \text{ on } n \text{ at time } t;

\text{APPEAR}(n, \tau, t), \text{DISAPPEAR}(n, \tau, t): \text{ Tuple } \tau \text{ appeared (disappeared) on node } n \text{ at time } t;
Key terms

- **Seed** - the “latest” node in our graph to cause the root event
  - Assume the seed is good - make variables in the seed immutable, seeds are equivalent
- **Equivalence** - establishing a mapping between variables for seeds of the same type
  - $A(x, y)_A \equiv A(x', y')_B$ means wherever $x$ appears in $A$’s tree, $x'$ should appear in the same spot on $B$’s tree

\[
A(x) \equiv A(x')
\]
The DiffProv Algorithm

Outline:

- Inputs: Two provenance trees, a good (reference) one $T_G$, and a bad one $T_B$
- Output: A set of changes $\Delta_{B\rightarrow G}$ to mutable tuples that 1) transforms $T_B$ into a tree that is equivalent to $T_G$, and 2) preserves $s_B$
- Identify the seed tuples in $T_G$ and $T_B$ (remember - seed tuples are immutable)
- Establish equivalence between the seeds
- Recursively:
  - Try to align the current subtrees of $T_G$ and $T_B$
  - If they can be aligned, recurse on parents
  - Else, make the missing tuple appear in $T_B$
  - Build the new provenance graph for $T_B$ after applying the change, and repeat
The DiffProv Algorithm: Identifying the Seed

- Start at the root of each tree
- Repeat:
  - Find child tuple (node) that appeared at the highest timestamp
  - If this is a leaf (of type INSERT), we have found our seed
  - If not, continue the procedure
The DiffProv Algorithm: Identifying the Seed
The DiffProv Algorithm: Identifying the Seed
The DiffProv Algorithm: Identifying the Seed
The DiffProv Algorithm: Establishing Equivalence

- Check if the seeds tuples are of the same type
  - Eg: A(1, 2, 4) and B(1, 2, 4) are not equivalent, because the seeds are of different types
The DiffProv Algorithm: Establishing Equivalence

Example:

\[ \text{pkstat}(pt, 8^*sz, c+1) :\text{- pkt}(pt, sz), \text{pktcnt}(c) \]

\[ \begin{align*}
\text{pkt}(51, 101) & \quad \text{pkt}(80, 100) \\
\text{pktcnt}(2) & \quad \text{pktcnt}(1) \\
\text{T}_B & \quad \text{T}_G
\end{align*} \]
The DiffProv Algorithm: Establishing Equivalence

Example:

pkstat(pt, 8*sz, c+1) :- pkt(pt, sz), pktcnt(c)

pt = 80 (T₀), 51(T₁)

sz = 100 (T₀), 101(T₁)
The DiffProv Algorithm: Establishing Equivalence

Example:

\[ \text{pkstat}(pt, 8 \cdot sz, c+1) :\text{-} \text{pkt}(pt, sz), \text{pktcnt}(c) \]

\[ pt = 80 (T_G), 51 (T_B) \]

\[ sz = 100 (T_G), 101 (T_B) \]
The DiffProv Algorithm: Aligning Larger Subtrees

Propagate taints

Example:

\[ \text{pkstat}(pt, 8*sz, c+1) :- \text{pkt}(pt, sz), \text{pktcnt}(c) \]

\[ pt = 80 \ (T_G), \ 51(T_B) \]

\[ sz = 100 \ (T_G), \ 101(T_B) \]

\[ p = pt, \ q = 8 \* \text{sz} \]
The DiffProv Algorithm: Aligning Larger Subtrees

Check if expected tuple exists

Example:

\[ \text{pktstat}(pt, 8*sz, c+1) :- \text{pkt}(pt, sz), \text{pktcnt}(c) \]

\[ pt = 80 \ (T_B), \ 51(T_G) \]

\[ sz = 100 \ (T_G), \ 101(T_B) \]

\[ p = pt, \ q = 8 * sz \]
The DiffProv Algorithm: Making Missing Tuples Appear

Propagate taints and inverted formulae from parent to child

Example:

pkstat(pt, 8*sz, c+1) :- pkt(pt, sz), pktcnt(c)

pt = 80 ($T_G$), 51($T_B$)

sz = 100 ($T_G$), 101($T_B$)

p = pt, q = 8 * sz
The DiffProv Algorithm: Making Missing Tuples Appear

For every child in $T_G$, compute equivalent child in $T_B$, checking if it exists. Recurse until base tuple hit.

Example:

```
pkstat(pt, 8*sz, c+1) :- pkt(pt, sz), pktcnt(c)
pt = 80 (T_G), 51(T_B)
sz = 100 (T_G), 101(T_B)
p = pt, q = 8 * sz
```
The DiffProv Algorithm: Updating $T_B$

If missing tuple is a base tuple, insert the equivalent tuple into $T_B$, and then repeat procedure.

Example:

```
.pkstat(pt, 8*sz, c+1) :- pkt(pt, sz), pktcnt(c)
```

$pt = 80$ ($T_G$), $51$ ($T_B$)

$sz = 100$ ($T_G$), $101$ ($T_B$)

$p = pt$, $q = 8 * sz$
Good Properties

- Complexity: linear (in size of $T_G$)
  - Naive approaches: exponential
- DiffProv fast because:
  - Uses provenance - ignores unrelated tuples
  - Uses taints and formulas - never guesses
- No false positives
  - I.e. DiffProv never recommends changes that are not helpful
DiffProv Failures

- Seeds of $T_G$ and $T_B$ have different types
  - No valid solution to DiffProv, so a better reference must be chosen by operator
- Solution involves changing an immutable tuple
  - E.g. involved changing a static flow entry, or the seed event
  - No solution, but DiffProv can still show what needs to be changed
- Non-invertible rules
  - DiffProv can still output attempted change
Extensions

- Distributed Operation
  - DiffProv implementation can be decentralized
  - In DiffProv algorithm, all steps can be performed on a single vertex event, its parent, and its children
  - So, DiffProv can be implemented so each node only stores its local tuples

- Temporal provenance
  - When adding missing tuples, DiffProv must consider system state as of the time at which missing tuple would have been added.
  - DiffProv w/ temporal provenance keeps log of tuple updates to allow efficiently adding them shortly before first needed
Limitations and Open Problems

- **Minimality** - DiffProv is not necessarily returning the smallest difference
  - It only tries to use the same rules as in $T_G$, but other derivations may be better
- **Reference events**
  - Operator must provide a reference event for DiffProv
  - Future paper could present a version of DiffProv that automatically chooses a reference event
- **Performance anomalies**
  - Provenance focuses on individual events, but issues with performance are inherently based on aggregations of events
- **Non-determinism**
  - DiffProv assumes events are mostly deterministic, but in e.g. load balancers this is not necessarily the case
Implementation

- Five major components
  - Provenance recorder
  - Front-end
  - Logging engine
  - Replay engine
  - DiffProv reasoning engine
Provenance recorder

- Three ways to extract provenance info:
  - Directly infer from source code (e.g. infer from NDLog rules in RapidNet)
  - Hooks to report dependencies (e.g. used for MapReduce recorder)
  - External specifications
Logging and Replay Engines

- Supports temporal providence
- Two approaches:
  - Runtime based - logging engine records all base events and intermediate derivations
  - Query-time based - logging engine records ONLY base events, and replay engine computes the intermediate derivations
  - Query-time based requires deterministic, but is better runtime performance to not save all intermediate derivations
Front-End and DiffProv Reasoning Engine

- Front-End
  - Accepts programs in NDLog or NetCore
  - Converts NetCore -> NDLog
- DiffProv Reasoning Engine
  - Applies DiffProv on the providence trees using the algorithm discussed
  - Issues replay requests to update the trees to the replay engine
  - “Brain” of the whole implementation
Evaluation

a) How well can DiffProv identify the actual root cause of a problem?

b) Does DiffProv have a reasonable cost at runtime?

c) Are DiffProv queries expensive to process?

d) Does DiffProv work well in a complex network with realistic routing policies and heavy background traffic?

Figure 1: Example scenario (SDN debugging).
Evaluation: Finding Root Cause

<table>
<thead>
<tr>
<th>Query</th>
<th>SDN1</th>
<th>SDN2</th>
<th>SDN3</th>
<th>SDN4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good example ($T_G$)</td>
<td>156</td>
<td>156</td>
<td>156</td>
<td>201/201</td>
</tr>
<tr>
<td>Bad example ($T_B$)</td>
<td>201</td>
<td>156</td>
<td>201</td>
<td>156/145</td>
</tr>
<tr>
<td>Plain tree diff</td>
<td>278</td>
<td>238</td>
<td>74</td>
<td>278/218</td>
</tr>
<tr>
<td>DiffProv</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1/1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>MR1-D</th>
<th>MR2-D</th>
<th>MR1-I</th>
<th>MR2-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good example ($T_G$)</td>
<td>1051</td>
<td>1001</td>
<td>588</td>
<td>588</td>
</tr>
<tr>
<td>Bad example ($T_B$)</td>
<td>1051</td>
<td>848</td>
<td>588</td>
<td>438</td>
</tr>
<tr>
<td>Plain tree diff</td>
<td>164</td>
<td>306</td>
<td>240</td>
<td>216</td>
</tr>
<tr>
<td>DiffProv</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Evaluation: Runtime Cost

Unsuitable reference events?

Latency increased by 6.7% for SDN

Latency increased by 2.3% for MR - optimized to 0.2%

Logging rate for different traffic and packet size
Evaluation: Query Processing

3 jobs - correct, faulty, update

Parallelized first 2 jobs since separate trees

3.8 milliseconds max for DiffProv reasoning

SDN4 takes twice as long because there are 2 faults
Evaluation: Query Processing

Breakdown for 3.8 milliseconds max for DiffProv reasoning

Detecting first divergence and making missing tuples appear takes time (track taints, evaluate formulas)

Takes more time to derive broken flow entries in SDN

MR1-D takes longer for divergence detection since trees are deeper
Evaluation: Complex Network

H1-S1-S2-H2 setup where S2 drops packets to H2

Simulate Stanford University network setup

Added 10 additional faults on the path and 10 faults not on the path

Additional background traffic: other clients

It works!