Sledgehammer: Cluster-Fueled Debugging

Andrew Quinn, Jason Flinn, and Michael Cafarella
Debugging

• What was your most challenging bug?
Tracing Tools

- Inject logic into software to track program state \[1\][2]
- Re-create failing execution and analyze tracing output

[1] “Where is the Bug and How Is It Fixed? An Experiment with Practitioners” (FSE ‘17)
Tracing Tools

• Tradeoff between amount of tracing and overhead
Tracing Tools

- Tradeoff between amount of tracing and overhead
Cluster-Fueled Debugging

- Accelerate complex debugging queries
- Shared cluster => on-demand debugging tool
Tracing Tools

• But, the work of these tools are sequential!
Sledgehammer

- Replay-Based: queries run over past execution
Sledgehammer

• Replay-Based: queries run over past execution

Scales to 1024 cores (speedup 416x)
Outline

• Vision of Cluster-Fueled Debugging
• Debugging Scenario
• Parallelizing a Query
• Evaluation
Scenario - memcached

Developer

“corrupt cache”

What’s the root cause?
What line of code?
Scenario – Today’s Tools

Developer

“corrupt cache”

Instrumentation

// walk cache data-structure
Char *checkCache();

checkCache();

Tracing

// find invalidation points:
void invalidInsts(int fd, int out);

Analyzers
Scenario – Today’s Tools

“corrupt cache”

```
// walk cache data-structure
Char *checkCache();
```

```
// find invalidation points:
void invalidInsts(int fd, int out);
```

```
void invalidInsts(int fd, int out);  
```

---

```
checkCache();
```

Instrumentation

---

```
Continuous Function Evaluation
```

- Infeasible Overhead
- Acceptable Overhead
- Limited Tracing
- Full Tracing

---

Developer

Amount of Tracing

Tracing Overhead

```
Tracing Overhead
```

```
Amount of Tracing
```

```
Limited Tracing
```

```
Full Tracing
```

---

Acceptable Overhead

```
```

Infeasible Overhead

```
```

Limited Tracing

```
```

Full Tracing

```
```
Scenario - Sledgehammer

Developer

\[\text{“corrupt cache”}\]

SH_Cont(checkCache)

//walk cache data-structure
Char *checkCache();

// find invalidation points:
void invalidInsts(int fd, int out);

Tracing

Sledgehammer

(New Debugging Tool)
Continuous Function Evaluation:
execute tracing after each instruction

CFE Log:
inst1, valid
inst2, invalid
...

Analyzers

Annotations
Scenario - Sledgehammer

Developer

```
// find invalidation points:
void invalidInsts(int fd, int out);
```

Analyzers

```
// walk cache data structure
Char *checkCache();
```

Annotations

```
SH_Cnt(checkCache)
```

Tracing

```
Continuous Function Evaluation:
execute tracing after each instruction
```

Sledgehammer

```
(CNew Debugging Tool)
inst1, valid
inst2, invalid
...```

```
```

Scenario - Sledgehammer

Character

“corrupt cache"

// walk cache data

Char *

checkCache();

// find invalidation points:
void invalidInsts(int fd, int out);

Does the lock protect the cache?

Developer

Sledgehammer

Analyzers

SH_Cont(checkCache)

Annotations

Tracing

(New Debugging Tool)

Continuous Function Evaluation:
execute tracing after each instruction

CFE Log:
inst1, valid
inst2, invalid
...

Does the lock protect the cache?
Scenario - memcached

Developer

“corrupt cache”

SH_Cont(checkCache)
SH_Hook(lock, lockAcquire)
SH_Hook(unlock, lockRelease)

Annotations

//walk cache data-structure
Char *checkCache ();
void lockAcquire(mutex *m);
void lockRelease(mutex *m);

Tracing

// find invalid and unlocked:
void badUpdate(int fd, int out);

Analyzers

Sledgehammer

(New Debugging Tool)
Continuous Function Evaluation:
Track tracing output after every instruction

(Cutting-edge Tool)
Retro-Logging:
Inject new logging into past execution

Introvirt SOSP ‘05
Scenario - Sledgehammer

Developer

“corrupt cache”

Sledgehammer

Analyzers

SH_Cont(checkCache)
SH_Hook(lock, lockAcquire)
SH_Hook(unlock, lockRelease)

Annotations

Tracing
(New Debugging Tool)
Continuous Function Evaluation:
Track tracing output after every instruction

Retro-Logging:
Inject new logging into past execution

Inst1 (initialization)
Inst2 (BUG)

void lockAcquire(mutex *m);
void lockRelease(mutex *m);

// find invalid and unlocked:
void badUpdate(int fd, int out);

Introduction SOSP ’05
Outline

• Vision of Cluster-Fueled Debugging
• Debugging Scenario
• Parallelizing a Query
• Evaluation
Parallelizing a Query

- Epoch parallelism – leverage deterministic replay

Doubleplay (ASPLOS ‘12)
Parallelizing a Query

- Unconstrained tracing code causes divergences

```
Crash or corrupt output!
```

Analyzer

Output
Tracing Isolation

- Prevent tracing code from updating replay state
- Requires scalability and lightweight technique

<table>
<thead>
<tr>
<th>Poor Scalability</th>
<th>Heavyweight</th>
<th>Lightweight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Pin / Valgrind</td>
</tr>
<tr>
<td>Good Scalability</td>
<td>Checkpointing (Introvirt SOSP ‘05)</td>
<td>Compiler-based Isolation</td>
</tr>
</tbody>
</table>
Compiler-based Isolation

- Instrument tracing code to log updates
- Revert updates using log

<table>
<thead>
<tr>
<th>Undo-log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
</tr>
<tr>
<td>Write</td>
</tr>
</tbody>
</table>
Compiler-based Isolation

- Instrument tracing code to log updates
- Revert updates using log

Scalable and Lightweight!
Parallelizing a query

- Multi-tiered approach to parallelize analysis
Parallelizing a query

- Multi-tiered approach to parallelize analysis
Analysis

• Developers construct…
  • Local runs on each core in parallel
  • Stream passes information over chain
  • Tree combines input from multiple analyzers
Analysis

- Developers construct…
  - Local runs on each core in parallel
  - Stream passes information over chain
  - Tree combines input from multiple analyzers

Parallel faster by up to 4x (mean 2x)
Parallelizing a query
Sledgehammer Tools

- Continuous Function Evaluation: Run after every instruction
- Retro-Timing: Query timing information
- Retro-Logging: Inject new logging code
Continuous Function Evaluation

- Logically, execute function after each instruction
- Instead, force determinism and trigger on input changes
Continuous Function Evaluation

- Static instrumentation produces input set
- Memory page protections track updates to input set
Outline

• Vision of Cluster-Fueled Debugging
• Debugging Scenario
• Parallelizing a Query
• Evaluation
## Scenarios

<table>
<thead>
<tr>
<th>Name</th>
<th>Benchmark</th>
<th>Replay Time</th>
<th>Tracing (millions)</th>
<th>1-Core Latency</th>
<th>1024-Core Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Corruption</td>
<td>Nginx</td>
<td>2s</td>
<td>8</td>
<td>5m25s</td>
<td>1s</td>
</tr>
<tr>
<td>Wild Store</td>
<td>MongoDB</td>
<td>30s</td>
<td>3</td>
<td>2h8m8s</td>
<td>17s</td>
</tr>
<tr>
<td>Atomicity Violation</td>
<td>Memcached</td>
<td>1m38s</td>
<td>43</td>
<td>2h10m52s</td>
<td>14s</td>
</tr>
<tr>
<td>Memory Leak</td>
<td>Nginx</td>
<td>1m16s</td>
<td>4</td>
<td>26m15s</td>
<td>3s</td>
</tr>
<tr>
<td>Lock Contention</td>
<td>Memcached</td>
<td>1m33s</td>
<td>76</td>
<td>54m42s</td>
<td>11s</td>
</tr>
<tr>
<td>Apache 45605</td>
<td>Apache</td>
<td>51s</td>
<td>2</td>
<td>4m10s</td>
<td>1s</td>
</tr>
<tr>
<td>Apache 25520</td>
<td>Apache</td>
<td>60s</td>
<td>4</td>
<td>11m57s</td>
<td>1s</td>
</tr>
</tbody>
</table>
Scalability

• Why does Sledgehammer not scale perfectly?
  - Initialization (tracer injection, restoring epoch start)
  - Outliers (latency of slowest vs. average epoch)

• More analysis in the paper
Cluster-Fueled Debugging

• Accelerate complex queries to interactive latencies
Sledgehammer

Scales to 1024 cores (speedup 416x)
Faster than without debugging!

Analysis