Diagnosing Performance Variations by Comparing Execution Traces

François Doray
Progress Report Meeting - May 2015
Introduction

Performance is a critical requirement

Sources of performance variations
- Update to a program, library or OS
- Interaction between tasks
- Programming error
- Different system load

Developers are not aware of this

Tracing
- Lots of details
Can we facilitate the diagnosis of performance variations with an algorithm that automatically identifies differences between groups of execution traces?
1. Literature Review

2. Solution

3. Case Studies

4. Performance Evaluation
**Dapper** Sigelman & al. (2010)
- Associate an identifier to incoming requests.
- Propagate the identifier.

**Critical Path in TraceCompass** Giraldeau & Dagenais
- Heuristic based on kernel events.

### Diagrams

1. **Dapper**
   - pid=1
   - req=1
   - pid=2

2. **Critical Path in TraceCompass**
   - pid=1
   - wake-up
     - target pid = 2
   - pid=2
   - wake-up
     - target pid = 1
1. Literature Review / Comparing Task Executions

**“Frames” mode of Chrome**
Chromium Authors

**Spectroscope**  Sambasivan & al. (2007)

<table>
<thead>
<tr>
<th></th>
<th>App</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 ms</td>
<td>23%</td>
<td></td>
</tr>
<tr>
<td>1500 ms</td>
<td>77%</td>
<td></td>
</tr>
</tbody>
</table>

**Differential Flame Graphs**  Gregg (2014)

Image credit: Brendan Gregg / With permission.

**TraceDiff**  Trumper & al. (2013)

Image credit: Jonas Trümper / With permission.
1. Literature Review / Call Stack

With Frame Pointer
- Traverse a linked list.

<table>
<thead>
<tr>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return address</strong></td>
</tr>
<tr>
<td>Previous ebp</td>
</tr>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Args</td>
</tr>
<tr>
<td><strong>Return address</strong></td>
</tr>
<tr>
<td>Previous ebp</td>
</tr>
<tr>
<td>Local variables</td>
</tr>
</tbody>
</table>

Without Frame Pointer
- Extract rules from the .eh_frame section of ELF.
- Implemented by libunwind.

<table>
<thead>
<tr>
<th>IP</th>
<th>CFA</th>
<th>ebp</th>
<th>eip</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0002</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
1. Literature Review

2. Solution

3. Case Studies

4. Performance Evaluation
2. Solution / Tracing call stacks

**cpu_stack**
- Generated periodically when a thread is running.
- Using ITIMER_PROF.

**syscall_stack**
- Generated on long system calls.
- Duration of system calls tracked in a kernel module.
- Stack captured from a signal handler.
2. Solution / Tracing call stacks

**cpu_stack**
- Generated periodically when a thread is running.
- Using ITIMER_PROF.

**syscall_stack**
- Generated on long system calls.
- Duration of system calls tracked in a kernel module.
- Stack captured from a signal handler.

+ Kernel events to compute the critical path
2. Solution / Enhanced Calling Context Tree

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Call A</td>
</tr>
<tr>
<td>2</td>
<td>Call B</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Return B</td>
</tr>
<tr>
<td>7</td>
<td>Call X</td>
</tr>
<tr>
<td>8</td>
<td>Return X</td>
</tr>
<tr>
<td>9</td>
<td>Return A</td>
</tr>
</tbody>
</table>

\[
A \\
\quad t = 9 - 1 = 8
\]

\[
B \\
\quad t = 6 - 2 = 4
\]

\[
X \\
\quad t = 8 - 7 = 1
\]
## 2. Solution / Enhanced Calling Context Tree

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Call A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Call B</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Wait thread 2</td>
<td>Call X</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Wait disk</td>
</tr>
<tr>
<td>5</td>
<td>Return B</td>
<td>Return X</td>
</tr>
<tr>
<td>6</td>
<td>Return B</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Call X</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Return X</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Return A</td>
<td></td>
</tr>
</tbody>
</table>

\[
A \\
\quad t = 9 - 1 = 8
\]

\[
B \\
\quad t = 6 - 2 = 4
\]

\[
X \\
\quad t = 8 - 7 = 1
\]

\[
T \\
\quad t = 6 - 3 = 3
\]

(Wait thread 2)

\[
X \\
\quad t = 5 - 3 = 2
\]

(Wait disk)

\[
D \\
\quad t = 5 - 4 = 1
\]
2. Solution / Enhanced Calling Context Tree

- Any type of latency.
  - CPU usage
  - Disk / network
  - Dependencies between threads

- Context of each latency.

- State History Tree.

\[ A \]
\[ t = 9 - 1 = 8 \]

\[ B \]
\[ t = 6 - 2 = 4 \]

\[ X \]
\[ t = 8 - 7 = 1 \]

\[ T \]
\[ t = 6 - 3 = 3 \]

\[ X \]
\[ t = 5 - 3 = 2 \]

\[ D \]
\[ t = 5 - 4 = 1 \]

(Wait thread 2)

(Wait disk)
Filters to build groups of executions.

- **Total Time**
- **Running Time**
- **Bytes read from disk**

Group A

Group B
2. Solution / Comparison View

«Enhanced» Differential Flame Graph

Transition between threads

- FunctionA()
- FunctionB()
- FunctionC()
- OtherThread()
- [wait thread]
- [wait disk]
- sys:read
- FunctionD()
- HandleRequest(int)
- main
1. Literature Review

2. Solution

3. Case Studies

4. Performance Evaluation
3. Case Studies

MUTEX
Mutex held during a long operation for no reason.
In MongoDB.

SLEEP
Using sleeps to synchronize threads.
In MongoDB.

PREEMPTION
Critical operation preempted by a low priority thread.

DISK
Web request slowed down by the OS committing data to the disk.
3. Case Studies

**MUTEX**
Mutex held during a long operation for no reason.
In MongoDB.

**SLEEP**
Using sleeps to synchronize threads.
In MongoDB.

**PREEMPTION**
Critical operation preempted by a low priority thread.

**DISK**
Web request slowed down by the OS committing data to the disk.
Demo

Try it yourself in a browser:
fdoray.github.io/tracecompare
1. Literature Review

2. Solution

3. Case Studies

4. Performance Evaluation
Let's review some concepts.

**PRIME**
CPU-Only.
Stacks: 0.1%
Stacks + critical: 0.2%

**BABELTRACE**
Short system calls.
Stacks: 1%
Stacks + critical: 1%

**FIND**
Long disk requests.
Stacks: 2%
Stacks + critical: 5%

**MONGOD**
Multi-thread.
Stacks: 2%
Stacks + critical: 9%
### 4. Performance Evaluation

**PRIME**
- CPU-Only.
- Stacks: 0.1%
- Stacks + critical: **0.2%**

**BABELTRACE**
- Short system calls.
- Stacks: 1%
- Stacks + critical: 1%

**FIND**
- Long disk requests.
- Stacks: 2%
- Stacks + critical: 5%

**MONGOD**
- Multi-threaded.
- Stacks: 2%
- Stacks + critical: **9%**

---

ETW on Windows: 0.0%
DTrace on Mac: 1.0%

ETW on Windows: 19%
DTrace on Mac: 22%

* MacBook Pro with Quad-core Intel® Core i7™-3720QM at 2.6 GHz, 8 GB RAM, SSD for Windows and Mac benchmarks.
Summary

◉ Trace **call stacks**.
◉ **Enhanced calling context trees**.
◉ Compare groups of executions using **histograms** and **flame graphs**.
◉ **Works** with open-source and enterprise apps.

Future Work

◉ Support more interactions:
  ○ VMs
  ○ GPU
  ○ Application-specific
◉ **Dynamic languages / JIT**
◉ Support code refactoring
Thanks!

ANY QUESTIONS?

Try the demo:
fdoray.github.io/tracecompare


Presentation by François Doray, master’s student at the Distributed open reliable systems analysis lab (DORSAL) of Polytechnique Montreal.

Special thanks to SlidesCarnival for releasing this presentation template for free (CC BY 4.0).