TraceCompare: Automatic Identification of Differences between Executions

François Doray
Tracing Summit – August 2015
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 don’t understand 100% of the systems they develop.

Tracing: Record events that occur during the execution of a system.
View a trace in TraceCompass
Can we facilitate the diagnosis of performance variations with an algorithm that automatically identifies differences between two groups of execution traces?
1. Related Work

2. Solution

3. Case Studies

4. Performance Evaluation
Approximation of Critical Path Giraldeau & Dagenais

- Heuristic that uses kernel events to build:
  - Graph of dependencies between threads.
  - List of segments that belong to the critical path of an execution.

Alternate solution: **Dapper** Sigelman & al. (2010)
1. Related Work / Comparing Task Executions

"Frames" mode of Chrome
Chromium Authors

Differential Flame Graphs Gregg (2014)

Spectroscope Sambasivan & al. (2007)

TraceDiff Trumper & al. (2013)
1. Related Work

2. Solution

3. Case Studies

4. Performance Evaluation
2. Solution / Required Events

**cpu_stack**
- Generated periodically when a thread is on the CPU.
- Uses 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 / Required Events

**cpu_stack**
- Generated periodically when a thread is on the CPU.
- Uses 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 of executions.
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>

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

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

\[ X \]
\[ t = 8 - 7 = 1 \]
### 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></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**: \( t = 9 - 1 = 8 \)
- **B**: \( t = 6 - 2 = 4 \)
- **X**: \( t = 8 - 7 = 1 \)
- **T**: \( t = 6 - 3 = 3 \) (Wait thread 2)
- **X**: \( t = 5 - 3 = 2 \) (Wait disk)
- **D**: \( t = 5 - 4 = 1 \)
2. Solution / Enhanced Calling Context Tree

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

◉ Context of each latency.

\[
\begin{align*}
A & \quad t = 9 - 1 = 8 \\
B & \quad t = 6 - 2 = 4 \\
X & \quad t = 8 - 7 = 1 \\
T & \quad t = 6 - 3 = 3 \\
X & \quad t = 5 - 3 = 2 \\
D & \quad t = 5 - 4 = 1
\end{align*}
\]

(Wait thread 2)

(Wait disk)
2. Solution / Enhanced Calling Context Tree

[18:10:27.684] sys_write_entry: { cpu_id = 0 },
{ fd = 4, count = 1024 }

[18:10:27.783] sys_write_exit: { cpu_id = 0 },
{ ret = 0 }

[18:10:28.093] sched_switch: { cpu_id = 0 },
{ prev_tid = 4, next_tid = 10 }

[18:10:28.689] app:hello: { cpu_id = 0 },
{ str = "Hello World!" }

◉ State History Tree
2. Solution / Comparison View

Filters to build groups of executions.

Total Time

Running Time

Bytes Read from Disk

Group A

Group B
Red = time difference between compared groups.
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.
Let's review some concepts:

**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.
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.

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.
1. Related Work

2. Solution

3. Case Studies

4. Performance Evaluation
4. Performance Evaluation / Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>LTTng overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prime</strong></td>
<td>0.2%</td>
</tr>
<tr>
<td>CPU only.</td>
<td></td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>5%</td>
</tr>
<tr>
<td>Long disk requests.</td>
<td></td>
</tr>
<tr>
<td><strong>Mongod</strong></td>
<td>9%</td>
</tr>
<tr>
<td>Interactions between threads.</td>
<td></td>
</tr>
</tbody>
</table>

* Quad-core Intel® Core™i7-3770 CPU @ 3.4 GHz, 16 GB RAM, 7200 RPM hard drive.
## 4. Performance Evaluation / Overhead Comparison

<table>
<thead>
<tr>
<th>Application</th>
<th>LTTng Overhead (Linux)</th>
<th>DTrace Overhead (Mac)</th>
<th>ETW Overhead (Windows)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prime</td>
<td>-0.1% ±0.3%</td>
<td>1.0% ±0.1%</td>
<td>0.0% ±0.1%</td>
</tr>
<tr>
<td>CPU only.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mongod</td>
<td>8% ±1%</td>
<td>24% ±0%</td>
<td>24% ±1%</td>
</tr>
<tr>
<td>Interactions between threads.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* 95% confidence intervals.

* MacBook Pro Quad-core Intel® Core i7™-3720QM @ 2.6 GHz, 8 GB RAM, SSD.
Summary

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

Future Work

◉ Support more interactions:
  ○ VMs
  ○ GPUs

◉ Dynamic languages / JIT

◉ Support code refactoring
Thanks!

QUESTIONS?

Try the demo:
fdoray.github.io/tracecompare

F. Giraldeau and M. R. Dagenais, "Approximation of critical path using low-level system events", to be published.


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).