ADVANCED TROUBLE-SHOOTING OF REAL-TIME SYSTEMS

BERND HUFMANN, ERICSSON
Troubleshooting tool

Framework to build trace visualization and analysis tools

Scalable: handle traces exceeding memory

Extensible for any trace or log format: Binary, text, XML etc.

Reusable views and widgets

Available as standalone product or set of plug-ins
TRACE COMPASS OVERVIEW
### Events Table

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Channel</th>
<th>Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-06-10 09:36:12.670 038 881</td>
<td>channel0</td>
<td>kmem_kmalloc</td>
<td>call_site=0xfffffffffa02d6b3e, ptr=0xffffffffa02d6b3e, bytes_req=708, bytes_alloc=1024, gfp_f</td>
</tr>
</tbody>
</table>
COMMON FEATURES

› Searching

› Filtering

› Highlighting
COMMON FEATURES

› Trace annotation (bookmarks) and markers
Pattern analysis
- Find a sequence of data within a trace

Customize Trace Compass without adding code
- Generate state systems
- Do timing analysis
- Define specialized views
CALL STACK VIEW

› Extensible view to display of call stacks over time
› LTTng-UST and finstrument-functions of GCC
TRACE CORRELATION

› Trace Compass can open multiple traces together to view it as one
  – This is called an Experiment

› Useful for
  – Traces coming from multiple nodes
  – Traces from applications written in different languages
  – Different layers (network, etc.)

› Traces can be synchronized by time
  – Manually
  – Automatic algorithm (extensible)
BUILT-IN TRACE TYPES

- Linux Tracing Toolkit - LTTng (UST, Kernel)
- Text & XML Logs (custom parsers)
- Common Trace Format – CTF
  - application, kernel, HW, bare metal, etc.
- Packet Capture
- Best Trace Format - BTF
- GDB Trace Points
TIMING ANALYSIS

› Real-time systems

› We have two metrics to analyse
  › what is the data and when did it come

› Timing is as important as data

› Measure time between a start and end state
  – Simple: Start and end event
  – Often: State Machine to determine start and end

› Represent execution times, latencies, latency chains etc.
TIMING ANALYSIS

› **Locate** timing problems
  › Missed **deadlines**
  › **Potential** missed deadline (find problem before it occurs)

› **Analyze** timing problems
  › Find root cause and solution
  › Solve **difficult** to debug sporadic problems
EXAMPLE

Soft IRQ Latency

- softirq_raise
- softirq_handler_entry
- softirq_handler_exit

Parameter: CPU ID, IRQ #

Latency 1

Latency 2

Total Latency
GENERALIZATION

- Time between start and end
- Time for each transition
- Percentage sub-duration vs total
YOUR TIMING ANALYSIS

› Define a state machine for timing analysis
  – Implementation in Java as Trace Compass extension
  – Data-driven pattern matching (in XML)
    › Defining timing analyses on-the-fly

› Store in a built-in segment store

› Visualize data in various supplied views
VISUALIZATION

› Table
  – Get raw data
  – Explore data
  – Sorting, highlighting, filtering

› Scatter Chart
  – Latency vs Time
  – Have a big picture of the current range
Visualization

- Distribution Chart
  - Find outliers and modes easily

- Statistics
  - Min, max, average etc.
  - Find worst offenders
  - Find worst possible offender combination
TIMING ANALYSIS

› **Locate** timing problems
  › Missed deadlines
  › **Potential** missed deadline (find problem before it occurs)

› **Analyze** timing problems
  › Find root cause and solution
  › Solve **difficult** to debug sporadic problems
EXAMPLE ROOT CAUSES

› System overload

› System misconfiguration (e.g. wrong priorities of tasks)

› Priority inversion
  – Lower priority task is blocking higher priority task (indirectly)

› Blocked threads, starvation, deadlock

› Slow code
RESOURCES VIEW

- Displays resources states (color-coded) over time
  - CPUs, IRQs, SoftIRQs

![Image of Resources View](image-url)
CRITICAL PATH

Displays of **system wait chains** for given process
PRIORITY VIEW

› Group processes per CPU and priority
› Quickly find priority inversion or misconfigured task priorities
› Note: View not mainlined yet – Prototype!
Find contention at the Kernel level using LTTng

Realized as XML pattern analysis

Count of simultaneous waits

Show all in timing analysis views

Uaddr vs Thread Gantt chart
OS TRACING OVERVIEW

- Overloaded resources
- CPU, Memory and IO Usage
- Counter-intuitive example, CPU usage too low:
  - Kernel memory usage is rising
    - Find the offending process
  - IO usage is high
    - Maybe it’s swaps
  - Too many seeks?
    - Low IO, low CPU, low memory usage and low bandwidth
FLAME GRAPH VIEW

› Aggregation of function durations per call stack
› Highlights most time consuming execution path
› Find functions for performance optimization
FUTURE DEVELOPMENT

› User-configurable periodic markers
› Custom charts
› Enhanced call graph analysis and views
› Call stack views using data-driven analysis
› Pin & clone of views
› Time based import of traces/experiments
› Scalable segment store
› Enhanced searching, filtering and highlighting in Gantt charts
› Data-driven analysis and view enhancements
› Cropping of traces
› Priority view
› …
REFERENCES

› Project pages
  – http://tracecompass.org

› Documentation
  – Trace Compass User Guide
  – Trace Compass Developer Guide
REFERENCES

› Linux Tracing Toolkit (LTTng)
  - http://lttng.org/

› Diagnostic and Monitoring Working Group
  - http://diamon.org/

› Common Trace Format (CTF)
  - http://diamon.org/ctf/

› Trace Research Project
  - http://hsdm.dorsal.polymtl.ca/
CONTACTS

› Bernd.Hufmann@ericsson.com

› Mailing list
  - tracecompass-dev@eclipse.org

› IRC
  - oftc.net #tracecompass

› Mattermost
  - https://mattermost-test.eclipse.org/eclipse/channels/trace-compass
Q&A
CUSTOM PARSERS

- Custom Text and XML Parsers
  - Line based parser with regex
  - XML based extracting data from XML elements and their attributes
EXAMPLE

› High Resolution Timer – cyclic test application of rt-tests
› Latency between timer expiry till task starts

Event: 1 2 3 4 5

\[ \Delta 1 \quad \Delta 2 \quad \Delta 3 \quad \Delta 4 \]

Latency = \( \Delta 1 + \Delta 2 + \Delta 3 + \Delta 4 \)

› Event 1: Timer expires
› Event 2: Interrupt begins executing
› Event 3: Interrupt handler marks the task to react
› Event 4: Linux scheduler switches to the task
› Event 5: Application task begins executing