Software Debugging and Monitoring for Heterogeneous Many-Core Telecom Systems

Overview of Research Program

Michel Dagenais, Professor
Dept. of Computer and Software Eng.
Objectives

- Smart clients everywhere, always online cloud-based services accessed transparently.
- Complex heterogeneous many-core systems.
- Crucial need for better tools!

The objective of this project is to provide better tools that can be used in production, under real load, for the debugging, performance analysis and monitoring of distributed heterogeneous many-core systems. We need to extract and collect information with low overhead, minimize the perturbation on the systems being studied, and efficiently analyse online the gigabytes of data produced.
Projects Evolution

- **TDMC**: Ericsson, DRDC, NSERC; static tracepoints for kernel and user-space tracing, multi-session tracing, a posteriori traces synchronization, state system.
- **AHLS**: Ericsson, DRDC, NSERC; live tracing and synchronization, user-definable state system and views (e.g. Windows), store multiple state subtrees in state system.
- **CTPD**: Ericsson, EfficiOS, NSERC, Prompt; tracing on Tilera, ARM, Xeon Phi, and GPUs, dynamic tracepoints, VM tracing and view, RPC Debugging, critical path.
- **HSDM**: Ericsson, EfficiOS, NSERC, Prompt; models (e.g., UML, MARTE) tracing and analysis, Kalray, Adapteva and Intel PT tracing, Cloud and Software Defined Networks tracing, specialized analysis for OpenCL and system resources (I/O, memory, power), tolerating unknown states.
Structure of Project

Project funding = 4 x industrial contribution
Modus operandi

• Problems identification and prioritisation with the industrial partners.

• M.Sc., Ph.D. and PostDoc students focus on these difficult applied research problems with input from industry.

• Research associates help the graduate students to integrate their new proposed algorithms in the toolchain for validation and optimisation.

• The best algorithms are added to the LTTng and Trace Compass toolchain with support from industrial partner's R&D engineers.

• LTTng is redistributed by most Linux vendors: Wind River, MontaVista, Linaro, LTSI, Debian, Ubuntu, Suse, Fedora/Red Hat... Code contributions were received from more than 100 individuals and 25 companies including Ericsson, Google, IBM, Red Hat, Ubuntu, Mentor...
**Objective:** develop common efficient mechanisms for tracing and debugging heterogeneous many-core systems.

- **T1M1:** tracing new architectures (Adapteva Epiphany, Kalray MPPA).
- **T1M2:** tracing new architectures (ARM CoreSight, Intel PT, bare-metal processors).
- **T1D1:** efficient algorithms to control debugging, profiling and tracing on many-core processors and to retrieve data.
- **T1D2:** scalable tracing / debugging mechanisms for dynamic instrumentation, checking conditions, aggregating values...

**Key:** low disturbance, low overhead, no impact on system, dynamically adjust the overhead / level of details compromise
Userspace Tracing

Approx time by event – 1 thread (nanoseconds)

- LTTng UST: 280 nanoseconds
- UST (W/O OPT): 500 nanoseconds
- Dtrace: 2400 nanoseconds
- SystemTap: 6000 nanoseconds
Userspace Tracing

Approx time by event – 8 threads (nanoseconds)

UST: 280
Dtrace: 19400
SystemTap: 56000
Trace collection performance

• User-space tracing in user space, no system call.
• “Unlikely if” for tracepoint activation.
• Efficient binary format being optimised and standardized as the Multi-Core Association Common Trace Format. No formatting!
• Per CPU buffers with local lockless atomic operations.
• Read Copy Update (RCU) synchronisation for configuration information (tracepoint activation, multiple sessions...).
• Zero-copy trace recording on disk.
• Tracepoints may be inserted even in interrupt and NMI contexts.
• Efficient timestamping (rdtsc) even from virtual machines.
• Most efficient and flexible tracer available!
Latency-tracker

- Kernel module to track down latency problems at run-time.
- Simple API that can be called from anywhere in the kernel (tracepoints, kprobes, netfilter hooks, hardcoded in other module or the kernel tree source code).
- Keep track of entry/exit events and calls a callback (e.g., to dump stack or take a trace snapshot) if the delay between the two events is higher than a threshold, or no exit before timeout.

```c
latency_tracker_event_in(tracker, key, threshold, timeout, callback);
...
latency_tracker_event_out(tracker, key);
```
Sample usage

- Block layer latency
  - Delay between block request issue and complete
- Wake-up latency
  - Delay between sched_wakeup and sched_switch
- Network latency
- IRQ latency
- System call latency
  - Delay between the entry and exit of a system call
- Offcpu latency
  - How long a process has been scheduled out

```c
on sched_switch:
    event = latency_tracker_get_event(next_pid);
    if event && ((now - event->start) > threshold):
        dump_stack(next_pid);
```
Tracker performance

- Controlled memory allocation.
- Lockless free-list.
- Out-of-context reallocation of memory if needed/enabled.
- Using userspace-rcu hashtable (kernel version) for lockless insert and lookup.
- Custom call_rcu thread to avoid the variable side-effects of the builtin one.
Dynamic instrumentation

- When we cannot recompile or restart program.
- When dynamic decisions can help optimise tracing.
- Comparison of different kernel and user-space solutions, using bytecode and JIT compilation.
Using eBPF bytecode or JIT for tracing

**Chart 1:**

*Pure eBPF Filter Performance with 50 Predicates*

- **Hardcoded**
- **eBPF(JIT)**
- **eBPF(Interpreted)**

**Chart 2:**

*Pure eBPF Filter Performance with Increasing Number of Predicates*

- **Hardcoded**
- **eBPF(JIT)**
- **eBPF(Interpreted)**

1018 ns/event
305 ns/event
**T2: Cloud Debugging and Monitoring**

**Objective:** provide efficient tools to monitor and understand the performance of clouds and cloud-based applications.

- **T2M1:** data collection, analysis and monitoring for clouds (OpenStack).
- **T2M2:** data collection, analysis and monitoring for Software Defined Networks (OpenDaylight).
- **T2D1:** monitoring of migrating virtual machines in the cloud, with ability to navigate through information at multiple levels.

**Key:** combine information from multiple levels, maintain scalability through sampling and varying levels of details.
Real Performance of Virtual Machines
**Objective:** propose a number of new advanced analysis and views, in particular for specialized parallel processors.

- **T3M1:** specialized programming models such as OpenCL, OpenMP and MPI.
- **T3M2:** specialized analysis and views for tasks such as virtual / physical memory usage, and power consumption.
- **T3D1:** sophisticated pattern matching on the state history and tracing events.
- **T3D2:** trace correlation for regression testing, highlighting the differences between similar runs of different versions, or in different contexts.

**Key:** tackle huge traces efficiently, help the user model the system and automate problem identification, provide contextualized analysis.
Trace Analysis

• Present events from different traces on a common reference time.
• Build a model of the traced system (process table, opened file descriptors...).
• Allow easy and efficient navigation and zooming in the trace.
• Present statistics and histograms for different intervals in the trace.
• Help identify the dependencies among events and trace the critical path.
• Traces may contain hundreds of GBs of data!
Eclipse Linux Tools
Distributed Traces Synchronization

- Each node, VM or heterogeneous processor may have its own independent clock, yet correlating events from different traces on a common time reference is needed.

- First order effect: different initial time and clock frequency.

- Second order effects: clock jitter, latency to read clock value, clock frequency drift with time.

- Identify corresponding send and receive events for packets to estimate clock differences.
Synchronisation

- New linear incremental algorithm to compute the clock offset and drift between two traces.
- Send and received events are matched by unique id (e.g., TCP sequence number for a given connection) and are incrementally added to build the upper and lower convex hull bounds.
Modeled State

- Current State Tree expressed as a generic attribute tree.
- For the Linux kernel, the Current State Tree contains: process table, file descriptors, sockets...
- Some events cause a state change (based on event type, event fields, current state).
- State builder expressed as custom java code or declarative XML expression.
- Efficient State History Tree database for quickly navigating through traces.
- The same declarative XML expressions can be used to define views, filters, abstract events and metrics.
Declarative XML expression example

XML structure

```xml
<filterHandler filterName="sched_switch">
  <initialFsm id="sched_switch" />
  <transitionInput id="sched_switch">
    <event eventName="sched_switch"/>
  </transitionInput>
  <action id="update Current_thread">
    <stateChange>
      <stateAttribute type="location" value="CurrentCPU" />
      <stateAttribute type="constant" value="Current_thread" />
      <stateValue type="eventField" value="next_tid" />
    </stateChange>
  </action>
  <fsm id="sched_switch" multiple="false">
    <precondition input="sched_switch"/>
    <stateTable>
      <stateDefinition name="sched_switch">
        <transition input="sched_switch" next="sched_switch" action="update Current_thread" />
        <transition input="#other" next="sched_switch" />
      </stateDefinition>
    </stateTable>
    <initialState id="sched_switch"/>
  </fsm>
</filterHandler>
```
State History Tree
Critical Path Analysis

• For each process, the elapsed time is broken down by state: running, waiting for resource (file, pipe, socket, CPU) or sleeping on a timer.

• The critical path for reaching a point in a process is the chain of unblocking events (wakeup) and their origin (device ready, lock freed or packet received from another process).

• Find and display the critical path with a breakdown of elapsed time per process and by state (e.g. web request took 3s of Firefox, 2s of X11, 5s of Apache, 3s of MySQL).
Critical Path Analysis view
Profile a path segment by call stack
Call stack computation

Unwind time according to call stack depth

CPython traceback time according to call stack depth

Time (ns)

Call stack depth

Offline Online

Built-in LTTng-UST
Real-time systems analysis

- Develop a model to define real-time task executions in a trace.
- Identify common problems in real-time systems and useful information for their analysis.
- Develop a method to analyze the trace segment corresponding to an execution to identify if the execution presents a problem.
Overview

1) Control Flow View

![Control Flow View](image1)

2) Define executions

![Define executions](image2)

3) Stackbars View

![Stackbars View](image3)

4) Critical Flow View with CP Complement view

![Critical Flow View](image4)

5) Other views

![Other views](image5)
Stackbar view: comparing task executions

![Figure 10: Stackbars view](image)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Start Time</th>
<th>Duration</th>
<th>Elapsed</th>
<th>Tid</th>
<th>Name</th>
<th>Running</th>
<th>Preempted</th>
<th>0</th>
<th>200,000</th>
<th>400,000</th>
<th>600,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17:44:23.626622785</td>
<td>814966</td>
<td>22015</td>
<td>timer</td>
<td>17092</td>
<td>530678</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17:44:23.346201997</td>
<td>791927</td>
<td>22015</td>
<td>timer</td>
<td>20258</td>
<td>593460</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17:44:23.074673317</td>
<td>790409</td>
<td>22015</td>
<td>timer</td>
<td>19904</td>
<td>597548</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17:44:23.518375271</td>
<td>772877</td>
<td>22015</td>
<td>timer</td>
<td>18452</td>
<td>533306</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>17:44:22.066646552</td>
<td>584651</td>
<td>22015</td>
<td>timer</td>
<td>18796</td>
<td>417085</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>17:44:22.297204106</td>
<td>583135</td>
<td>22015</td>
<td>timer</td>
<td>18651</td>
<td>423377</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>17:44:22.573953293</td>
<td>563930</td>
<td>22015</td>
<td>timer</td>
<td>27012</td>
<td>394978</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17:44:22.203374386</td>
<td>401778</td>
<td>22015</td>
<td>timer</td>
<td>28081</td>
<td>8022</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>17:44:22.012605091</td>
<td>398794</td>
<td>22015</td>
<td>timer</td>
<td>29505</td>
<td>5893</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>17:44:22.702171764</td>
<td>394549</td>
<td>22015</td>
<td>timer</td>
<td>90284</td>
<td>7888</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>17:44:22.966398830</td>
<td>393609</td>
<td>22015</td>
<td>timer</td>
<td>21984</td>
<td>5338</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>17:44:22.286280208</td>
<td>386999</td>
<td>22015</td>
<td>timer</td>
<td>32593</td>
<td>37716</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>17:44:22.212617985</td>
<td>377255</td>
<td>22015</td>
<td>timer</td>
<td>28344</td>
<td>35306</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>17:44:23.781131115</td>
<td>374956</td>
<td>22015</td>
<td>timer</td>
<td>28409</td>
<td>8458</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>17:44:22.409627769</td>
<td>370939</td>
<td>22015</td>
<td>timer</td>
<td>28334</td>
<td>7955</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distribution: duration and along the time axis

Figure 18: Problematic executions

Figure 19: Normal executions

Figure 20: Time View
Critical path view for a task execution
TraceCompare: diagnose performance variations

- Good versus bad run: program, library or OS version, hardware, network load?
- Collect execution traces and compute different metrics for different tasks (e.g., time for complete process, search request, or frame redraw).
- Add call stack samples collected periodically, or upon a very long system call.
Call context tree with nested calls and waits

<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>

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

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

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

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

(Wait thread 2)

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

(Wait disk)

\[
D \\
t = 5 - 4 = 1
\]
Select groups of executions based on metrics

- Total Time
- Running Time
- Bytes read from disk

Group A vs Group B
Differential flame graph of the two groups

- View of call stack over time for the tasks (assuming relatively stable call tree).
- Average duration of each call in group A versus group B, highlight where the differences are significant.
Tracing the GPU as an asynchronous resource
Combining info from OpenCL, driver and HW

User Space
- OpenCL Application
- tracepoints

Kernel Space
- Linux Kernel
- Drivers (Graphic, Audio, ...)
- tracepoints

Hardware Layer
- CPU
- Sound card
- GPU
- ...
CPU trace

Trace Compass:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>11393</td>
<td>11256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lttng</td>
<td>11423</td>
<td>11393</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sinoscope</td>
<td>11456</td>
<td>11393</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sinoscope</td>
<td>11457</td>
<td>11456</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sinoscope</td>
<td>11458</td>
<td>11456</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lttng</td>
<td>11460</td>
<td>11393</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unity-scope-hom</td>
<td>3301</td>
<td>1982</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GPU trace

CodeXL
(AMD’s GPU tracing tool)

Limitations:
• OpenCL application has to be launched by CodeXL
• Recording trace performance for large traces
• AMD GPU only
Combined trace: GPU contention
Combined trace: CPU preemption
T4: Model-Driven Engineering support

**Objective:** fully support model-driven engineering with tracing, debugging and testing tools.

**Personnel:** Prof. Juergen Dingel, Francis Bordeleau Ph.D., Sébastien Gérard Ph.D., Bernd Hufmann B.Eng., RA1.

- T5M1, T5M2: extend the model-level to store as attribute values, current states, transitions taken and messages sent.
- T5D1: new algorithms and techniques for the automatic correction and refinement of environment assumptions based on runtime information.
- T5M3, T5M4: extend the model-level to specify test cases, execute the tests and annotate the results.
- T5D2: new automatic test generation algorithms for correcting, refining and completing the model-level information.
Model-Driven Engineering support
Conclusion

• Efficient tracing is an essential tool for high performance heterogeneous distributed systems.

• Free software as infrastructure: natural collaboration, rich user community, easy customization, no surprise (e.g. product or supplier end of life, high cost per seat).

• Viewpoints of multiple clients: benefit from the experience, use cases and best practices of several technology leaders.

• Technology transfer: research and prototyping in Polytechnique, collaborative development of mainline implementations.

• LTTng delivers low disturbance, scalable tracing.