SOFTWARE DEBUGGING AND MONITORING FOR MULTI-CORE SYSTEMS

Naser Ezzati
DORSAL Lab
Polytechnique Montreal, Canada

Sharif-DNSL
Jan 2016
AGENDA

- Problem Faced
- Static and Dynamic Analysis
- Debugging vs Tracing
- Trace Analysis Tools
- Research Tracks
- Conclusion
PROBLEM FACED

- Today's systems are composed of computers or virtual machines that interact between themselves.
- Understand the running behavior of those systems or find suspicious activities is more difficult.
- How to verify if the system is working as intended?
- Why is the system slow? Where is the bottleneck?
- Why do we get this incorrect answer once in a billion times?
- Are there intrusion attempts? Did they succeed?
- Are we leaking information?
Static vs. Dynamic Analysis

- **Static Analysis**
  - Source codes and other artifacts
    - Not available
    - Outdated
    - Difficult to analysis

- **Dynamic Analysis**
  - Runtime behaviour analysis
  - Tracing
    - Performance bottlenecks
TRACING

What is Tracing?

- Process of collecting information about the program's execution
  - Trace-points
    - Inserted before compile-time, enabled/disabled at run-time
    - People can use them to extract useful information without having to know the code
  - The later analysis of this information may help us understand why a part of the software is not behaving as it is expected to.
  - 'Heisenbugs' detection, hard to detect errors
    - Race conditions, Deadlocks, Non-deterministic behavior
    - Multiple layers
      - Middleware, VM, OS, hypervisor
    - Performance Problems
      - problems are not reproducible in the developer’s environment!

Debuggers:

- Debuggers are indispensable, but they only show a snapshot.
TRACING USECASES

- Finding cause of
  - Performance issues
  - Concurrency issues
  - Failures, crashes

- System-wide troubleshooting
  - Multiple layers, multi-core, multi-processor, multiple nodes, etc.

- Live monitoring of system in production
  - Resource usage (e.g. CPU load)
  - Raising alarms, warnings
  - Overload protection
TRACING TOOLS

Classification:
- Userspace Tracing
  - Chrome://tracing
- Kernel Tracing
- Hardware Tracing

ETW: Event Tracing for Windows

Linux Tracing Tools:
- SystemTap
- Perf
- DTrace
- LTTng
USERSPACE TRACING

- UST Trace Library
- Example:
  - LTTng UST
  - Chrome:tracing
void function(void)
{
    int i = 0;
    long vals[3] = { 0x42, 0xCC, 0xC001CAFE }; 
    float flt = M_PI;

    [...] 
    tracepoint(ust_tests_hello, 
    tptest, 
    i, 
    &vals, 
    flt);

    [...] 
}
TRACEPOINT_EVENT(
    /* Provider name */
    ust_tests_hello,

    /* Tracepoint name */
    tptest,

    /* Type, variable name */
    TP_ARGS(int, anint,
        long *, values,
        float, floatarg),

    /* Type, field name, expression */
    TP_FIELDS(const ctf_integer(int, intfield, anint),
        ctf_array(long, arrfield1, values, 3),
        ctf_float(float, floatfield, floatarg))
)
$ LD_PRELOAD=./wrapper.so ./prog

prog.c

ptr = malloc(64);

void* malloc(size_t size) // wrapper for malloc
{
    void* ret;
    static void* (*realmalloc)(size_t size) = NULL;
    if (realmalloc == NULL)
        realmalloc=dlsym(RTLD_NEXT, "malloc");
    ret = realmalloc(size);
    tracepoint(percepio, malloc, size, ret);
    return ret;
}
## Linux Kernel Tracing

<table>
<thead>
<tr>
<th>Tools</th>
<th>Trace-cmd</th>
<th>Perf tools</th>
<th>Ivm</th>
<th>SystemTap</th>
<th>LTTng</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ftrace debugfs</td>
<td></td>
<td>Perf syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function callgraph</td>
<td></td>
<td>Event Trace</td>
<td>Syscall trace</td>
<td>eBPF tracer</td>
<td>Misc tracers</td>
</tr>
<tr>
<td>Trace events</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ftrace callgraph</td>
<td></td>
<td>Traceevent</td>
<td>Dynamic Event</td>
<td>Static Event</td>
<td>Perfevent</td>
</tr>
<tr>
<td>instrumentations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ftrace (Function trace)</td>
<td></td>
<td>kprobes</td>
<td>uprobes</td>
<td>Tracepoint</td>
<td>Performance counter</td>
</tr>
</tbody>
</table>
**TRACEPOINT**

- Statically placed at different logical places in the kernel
- More than 250 tracepoints
- TRACE_EVENT() macro
TRAVE_EVENT MACRO

TRACE_EVENT(sched_switch,

TPPROTO(struct task_struct *prev,
        struct task_struct *next),

TPARGS(prev, next),

TPSTRUCT_entry(
    __array(
        __field(
            char, prev_comm, TASK_COMM_LEN
        ),
        __field(
            pid_t, prev_pid
        ),
        __field(
            int, prev_prio
        ),
        __field(
            long, prev_state
        )
    )

...)

TP_fast_assign(
...)

);
/* context_switch - switch to the new MM and the new thread's register state. */
static inline void
collection_switch(struct rq *rq, struct task_struct *prev, 
                     struct task_struct *next)
{
    struct mm_struct *mm, *oldmm;

    prepare_task_switch(rq, prev, next);
    trace_sched_switch(prev, next);
    mm = next->mm;
    oldmm = prev->active_mm;
    ...

1649.424856148  kernel.sched_schedule  { prev_pid = 24, next_pid = 1765 }

Time stamp  Event type  Payload
**Probes**

- **Kprobes**: dynamic kernel tracing
  - Function calls, returns, line numbers
- **Uprobes**: dynamic user-level tracing
LTTNG + TRACE COMPASS
LTTng: Linux Trace Toolkit Next Generation

- Scalable tracer
- Fast tracer
- Minimal impact and overhead on the target
- Output data in unified format (CTF)
- Flight-recorder
- Support kernel and user-space tracing
- Easy installation:
  - Support kernel from 2.6.38 +
- Linux Distribution:
  - Ubuntu
  - Debian
  - Fedora
  - Arch
  - Suse
  - Red Hat
- Other OS:
  - Android, FreeBSD, Cygwin
**Host-Side User Interfaces**
- Babeltrace (MIT/BSD)
  - Trace converter
  - Trace pretty printer
  - Allow open source and proprietary plugins
  - libbabeltrace (MIT/BSD)

**LTtng Command Line Interface (GPLv2)**
- liblttngctl (GPLv2)
  - llttng-ust (GPLv2.1)

**LTtng Session Daemon (GPLv2)**
- liblttngctl (GPLv2.1)
  - llttng-ust (GPLv2.1)

**Eclipse Tracing and Monitoring Framework (EPL)**
- Trace display and analysis
- Trace control
- Allow open source and proprietary plugins

**Custom Control Software**
- Interface with proprietary cluster management infrastructures
  - liblttngctl (GPLv2.1)

**Host**
- SSH connexion

**Target**

- **C/C++ Application**
  - Tracepoint*
  - Tracepoint Probes*
  - llttng (GPLv2)
  - llttng-ust (GPLv2.1)

- **Java/Erlang Application**
  - Tracepoint*
  - llttng VM adaptor
  - Tracepoint Probes*
  - llttng (GPLv2)
  - llttng-ust (GPLv2.1)

- **Linux kernel**
  - Tracepoint*
  - Dynamic probes (kprobes)
  - llttng modules (GPLv2/GPLLv2.1)
  - llttng-ust (GPLv2.1)
  - llttng-consumer (GPLv2)

**LTtng Consumer Daemon (GPLv2)**
- Zero-copy data transport or aggregator
- Export raw trace data, statistics and summary data
- Snapshots from in-memory flight recorder mode
- Store all trace data, discard on overrun

**Local storage CTF†**

**CTF† over TCP/UDP/SSH**

**† Common Trace Format (CTF)**
- Compact binary format
- Self-described
- Handles HW&SW tracing
- TCP and UDP network streaming
- Flexible data layouts for expressiveness and highest throughput
- Layout allows fast seek and processing of very large traces (> 10GB).

**Tracepoint and Probes Characteristics**
- Low overhead, no trap, no system call,
- Re-entrant: Signal, thread and NMI-safe,
- Wait-free read-copy update,
- Can be used in real-time systems,
- Use GCC asm goto and Linux kernel static jumps,
- Cycle-level time-stamp,
- Runtime activation of statically and dynamically inserted instrumentation,
- Non-blocking atomic operations,
- Allow tracing of proprietary applications and proprietary control software (GPLv2.1 license).
LTtng is FAST! (Kernel)

Strace VS LTtng Tracing

Timing of a find of 100000 files (seconds)

- find: 0.54 seconds
- find + lttnng: 1.4 seconds
- find + strace: 38.8 seconds
LT Tng IS FAST! (UST)

**Userspace Tracing**

Approx time by event – 1 thread
(nanoseconds)

- LT Tng UST: 280
- Dtrace: 2400
- SystemTap: 6000
LTNTng IS FAST!
EXAMPLE KERNEL TRACE SESSION: SIMPLE AND UNIFIED COMMAND LINE

$lttng create session
$lttng enable-event -k-a (or -k sched_switch)
$lttng enable-event -u -a
$lttng start
$lttng stop
$lttng view
LTtng Common Trace Format Viewers

- **Babeltrace** (MIT/BSD)
  - Trace converter
  - Trace pretty printer
  - Allow open source and proprietary plugins
  - libbabeltrace (MIT/BSD)

- **LTtngTop** (GPLv2)
  - Top-alike resource usage view
  - Read live from buffers
  - libbabeltrace (MIT/BSD)

- **Eclipse Tracing and Monitoring Framework** (EPL)
  - Trace display and analysis
  - Trace control
  - Allow open source and proprietary plugins

- Local storage
  - CTF†

- Memory-mapped buffers or splice, poll, ioctl

- Trace Data
- Libraries
OUTPUT TRACE: BABELTRACE

```
[13:36:30.119503690] (0.000004293) kernel exit syscall: { cpu_id = 0 }, { ret = 139998849839104 }
[13:36:30.119513141] (0.0000000277) kernel sys rt sigprocmask: { cpu_id = 0 }, { how = 2, nset = 0x7FF504E8640, oset = 0x0, sigsetsize = 8 }
[13:36:30.119513896] (0.0000000755) kernel exit syscall: { cpu_id = 0 }, { ret = 0 }
[13:36:30.119523258] (0.0000001754) uist uist tests_demo:starting: { cpu_id = 0 }, { value = 123 }
```
Output Trace: System Call Trace

lttng enable-event --syscall -a
<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4129</td>
<td>4129</td>
<td>firefox-bin</td>
</tr>
<tr>
<td>7709</td>
<td>7709</td>
<td>ifconfig</td>
</tr>
<tr>
<td>4196</td>
<td>4196</td>
<td>wicd</td>
</tr>
<tr>
<td>4971</td>
<td>4971</td>
<td>kworker/1:2</td>
</tr>
<tr>
<td>7447</td>
<td>7447</td>
<td>kworker/0:1</td>
</tr>
<tr>
<td>7373</td>
<td>7373</td>
<td>/usr/bin/x-term</td>
</tr>
<tr>
<td>2580</td>
<td>2580</td>
<td>Xorg</td>
</tr>
<tr>
<td>2441</td>
<td>2441</td>
<td>dbus-daemon</td>
</tr>
<tr>
<td>4227</td>
<td>4227</td>
<td>wicd-monitor</td>
</tr>
<tr>
<td>4075</td>
<td>4075</td>
<td>tor</td>
</tr>
<tr>
<td>6021</td>
<td>6021</td>
<td>xscreensaver</td>
</tr>
<tr>
<td>7298</td>
<td>7298</td>
<td>kworker/u:0</td>
</tr>
<tr>
<td>6808</td>
<td>6808</td>
<td>kworker/u:2</td>
</tr>
<tr>
<td>2498</td>
<td>2498</td>
<td>acpid</td>
</tr>
<tr>
<td>5957</td>
<td>5957</td>
<td>awesome</td>
</tr>
<tr>
<td>4114</td>
<td>4114</td>
<td>uml_switch</td>
</tr>
<tr>
<td>2585</td>
<td>2585</td>
<td>wpa_supplicant</td>
</tr>
<tr>
<td>7682</td>
<td>7682</td>
<td>lttnngtop</td>
</tr>
<tr>
<td>18675</td>
<td>18675</td>
<td>migration/1</td>
</tr>
<tr>
<td>28967</td>
<td>28967</td>
<td>watchdog/0</td>
</tr>
<tr>
<td>28969</td>
<td>28969</td>
<td>watchdog/1</td>
</tr>
<tr>
<td>7682</td>
<td>7682</td>
<td>lttnngtop</td>
</tr>
<tr>
<td>7682</td>
<td>7684</td>
<td>lttnngtop</td>
</tr>
<tr>
<td>7588</td>
<td>7648</td>
<td>lttnng-sessionid</td>
</tr>
<tr>
<td>7588</td>
<td>7588</td>
<td>lttnng-sessionid</td>
</tr>
<tr>
<td>7274</td>
<td>7274</td>
<td>swapper/1</td>
</tr>
<tr>
<td>7274</td>
<td>7274</td>
<td>kworker/0:2</td>
</tr>
</tbody>
</table>
TRACE COMPASS

- Eclipse IDE integration or RCP app
- Framework to build trace visualization and analysis tools
  - Mix kernel and userspace trace analysis
- Scalable: handle traces exceeding memory
- Extensible for any trace or log format
  - CTF, Binary, text, XML etc.
- Reusable views and widgets
- Control flow view, histogram, event list, trace statistics, CPU/resource allocation
**Stateful Analysis: State System**

- State system abstracts events, analyses traces and creates models to be displayed

![Diagram showing the components of a state system](image-url)

- **Events**:
  - `sched_switch(process)`
  - `irq_entry`
  - `irq_exit`
  - `sched_switch(swapper)`

- **States**:
  - WAIT
  - USERMODE
  - INTERRUPTED
  - USERMODE
  - WAIT
Control Flow View

- Display processes state changes (color-coded) over time
  - USERMODE, SYSCALL, INTERRUPEP, WAIT_FOR_CPU, etc
Display system resource states (color-coded) over time
DATA DRIVEN ANALYSIS
HOW TO USE TRACING?

- Learning
  - OS concepts teaching by LTTng traces
- Debugging
  - Program comprehension
  - Bug finding
  - Root-cause analysis
- Anomaly Detection
- Dependency Analysis
- Network Analysis
Research Tracks

- Multi level trace analysis:
  - Multi Level Trace Abstraction
    - Metric based, Data driven, Visual abstraction, Resource abstraction
      - Stateful trace aggregation
  - Multi Level Trace Visualization
    - Label Placement
  - Statistics Framework (offline and online data)
RESEARCH TRACKS (2)

- **Automated Fault Identification**
  - Pattern Library
    - Various Attack Patterns
  - Kernel Execution Path
    - example: apache, shell
RESEARCH TRACKS (3)

- Dependency Analysis
  - System-level critical path analysis
  - Provide trace analysis tools to understand the overall performance of a distributed application.
What is the server doing?

1) GET /wkdb/polls/1
2) POST vote
3) GET redirect
What is the server doing?
Device wake-up

Task wake-up
Trace Compare: diagnose performance variations by comparing traces
  - Compare different parts of a trace
  - Compare different trace files
    - Normal or faulty execution

Case 1: a server has to read data from the disk to fulfill requests. At regular intervals, a request is about 5 times slower than usual.

Case 2: a client application generates data and inserts it into a MongoDB database (version 2.5.4). Most of the time, the whole operation takes around 10 ms. However, a fraction of the time, the operation takes more than 100 ms.

Case 3: batch insert commands are sent to a MongoDB server. The commands are run in less than 700 μs most of the time. However, about 1 in 10 000 commands takes between 3 and 5 seconds to complete.
RESEARCH TRACKS (4)

- Data Driven Analysis
  - Language
    - XML
  - Aggregation
  - Filtering
    - Debugging the patterns
  - Program comprehension
  - Attack detection:
    - SYN Flood Attack
  - I/O latency analysis
    - Why some web requests take too long time?
```c
void workFor(int micros)
{
    clock_t start, end;
    start = clock();
    end = start + (micros / (CLOCKS_PER_SEC / 1000000));
    while (clock() < end) {};
}
int main(int argc, char **argv)
{
    int nb_threads = 5;
    int nb_loops = 10;
    int i;
    srand(time(NULL));
    fprintf(stderr, "Tracing...\n");
    #pragma omp parallel private(i) num_threads(nb_threads)
    for (i = 0; i < nb_loops; i++) {
        int id = omp_get_thread_num() + 1;
        /* Loop starts here */
        workFor(rand() % 50000);
        //Connection attempted
        tracepoint(ust_myprog, connection_wait, id);
        workFor(rand() % 50000);
        //Connection is established
        tracepoint(ust_myprog, connection_start, id);
        workFor(rand() % 50000);
        //Connection ends
        tracepoint(ust_myprog, connection_end, id);
    }
    fprintf(stderr, "Done.\n");
    return 0;
}
```
<eventHandler eventName="ust_myprog:connection_wait">
  <stateChange>
    <stateAttribute type="constant" value="Threads" />
    <stateAttribute type="eventField" value="id" />
    <stateValue type="int" value="$STATE_CONNECTING" />
  </stateChange>
</eventHandler>

<eventHandler eventName="ust_myprog:connection_start">
  <stateChange>
    <stateAttribute type="constant" value="Threads" />
    <stateAttribute type="eventField" value="id" />
    <stateValue type="int" value="$STATE_ESTABLISHED" />
  </stateChange>
</eventHandler>

<eventHandler eventName="ust_myprog:connection_end">
  <stateChange>
    <stateAttribute type="constant" value="Threads" />
    <stateAttribute type="eventField" value="id" />
    <stateValue type="null" />
  </stateChange>
</eventHandler>

...
CONCLUSION

- Many problems can only be studied live, in production.
- Tracing is a great solution.
- LTTng and TraceCompass provide an excellent platform to build advanced analysis.
THANK YOU

Downloads:
- LTTng: www.lttng.org
- TraceCompass: www.tracecompass.org
- Publications: www.dorsal.polymtl.ca

Me:
- email: n.ezzati@polymtl.ca, ezzati@gmail.com