Ineffective and effective way to find out latency bottlenecks by Ftrace

Yoshitake Kobayashi
Advanced Software Technology Group
Corporate Software Engineering Center
TOSHIBA CORPORATION

16 Feb, 2012
Agenda

- About Ftrace
- Problem definition
- Some actual examples to fix latency issue
- Conclusion
About Ftrace

- Ftrace is a lightweight internal tracer
  - Event tracer
  - Function tracer
  - Latency tracer
  - Stack tracer

- The trace log stay in the kernel ring buffer

- Documentation available in kernel source tree
  - Documentation/trace/ftrace.txt
  - Documentation/trace/ftrace-design.txt
About Ftrace

- Ftrace is a lightweight internal tracer
  - Event tracer
  - **Function tracer**
  - Latency tracer
  - Stack tracer

- The trace log stay in the kernel ring buffer

- Documentation available in kernel source tree
  - Documentation/trace/ftrace.txt
  - Documentation/trace/ftrace-design.txt
Definition of latency

- Latency is a measure of time delay experienced in a system, the precise definition of which depends on the system and the time being measured. Latencies may have different meaning in different contexts.
  

- In this presentation

---

expected time to wake up

latency

actual time to wake up
Problem definition

- Evaluate the scheduling latency by Cyclictest*... but

Problem definition

- Evaluate the scheduling latency by Cyclictest* .... but

- Need to identify the cause of above problems

Problem definition

- Evaluate the scheduling latency by Cyclictest*.... but

  **Case 1**

  ![Graph showing latency distribution for Case 1]

  Why?

  23ms

  **Case 2**

  ![Graph showing latency distribution for Case 2]

  Why?

- Need to identify the cause of above issues

---

*Cyclictest (RTwiki): https://rt.wiki.kernel.org/articles/c/y/c/Cyclictest.html*
First mistake

- Incorrect use of trace-cmd

```shell
# trace-cmd start -p function_graph ; target_prog ; trace-cmd stop
```

- The target_prog start with higher priority and stop if it missed deadline
First mistake

- Incorrect use of trace-cmd

```bash
# iostress '50%' &
# trace-cmd start -p function_graph ; target_prog ; trace-cmd stop
```

- The target_prog start with higher priority and stop if it missed deadline

- Ftrace record the logs in the kernel ring buffer
  - The older data just overwritten by new data
  - Evaluate with too much workload is not a good idea

- Need to care about scheduling priority
  - Run a shell with higher priority
  - Stop logging in the target_prog
An example for stop tracing (Cyclictest)

- rt-tests/src/cyclictest/cyclictest.c
- Cyclictest has following option
  
  "-b USEC" : send break trace command when latency > USEC

```c
if (!stopped && tracelimit && (diff > tracelimit)) {
    stopped++; tracing(0);
    shutdown++;
pthread_mutex_lock(&break_thread_id_lock);
    if (break_thread_id == 0)
        break_thread_id = stat->tid;
    break_thread_value = diff;
pthread_mutex_unlock(&break_thread_id_lock);
}
```
Case 1: Evaluation environment

- CPU: Intel Pentium 4 (2.66GHz)
- Memory: 512MB
- Kernel: Linux 2.6.31.12-rt21
  - `echo -1 > /proc/sys/kernel/sched_rt_runtime_us`
Case1: Summary of evaluation result

- Evaluated without other CPU work load

<table>
<thead>
<tr>
<th>Cycle [us]</th>
<th>Min.</th>
<th>Ave.</th>
<th>Max</th>
<th>Number of DL misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>18.118</td>
<td>19.162</td>
<td>25.629</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>18.171</td>
<td>19.269</td>
<td>32.615</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>17.935</td>
<td>19.361</td>
<td>27207.563</td>
<td>1</td>
</tr>
</tbody>
</table>
Case 1: Check the trace log (First attempt)

- Stop the Cyclic test if the evaluated latency is higher than the given maximum
- Backward search for `sys_clock_gettime()` function
- Check the log for `sys_rt_sigtimedwait()` function

Here! (highlighted line)
Case1: Check the kernel source (First attempt)

- The `activate_task()` defined in `kernel/sched.c`

```c
static void inc_nr_running(struct rq *rq)
{
    rq->nr_running++;
}
```

Only an increment instruction

```
static void activate_task(struct rq *rq, struct task_struct *p, int wakeup, bool head)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;

    enqueue_task(rq, p, wakeup, head);
    inc_nr_running(rq);
}
```

Here!
Case1: Check the trace log (Second attempt)

- Result of the function graph tracer

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>184.976213</td>
<td>0) 0.404 us pre_schedule_rt();</td>
</tr>
<tr>
<td>184.976214</td>
<td>0) put_prev_task_rt() {</td>
</tr>
<tr>
<td>184.976214</td>
<td>0) update_curr_rt() {</td>
</tr>
<tr>
<td>185.004323</td>
<td>0) 0.537 us sched_avg_update();</td>
</tr>
<tr>
<td>185.004324</td>
<td>0) 28109.76 us }</td>
</tr>
<tr>
<td>185.004325</td>
<td>0) 0.496 us pick_next_task_rt();</td>
</tr>
<tr>
<td>185.004326</td>
<td>0) 0.442 us perf_counter_task_sched_out();</td>
</tr>
<tr>
<td>185.004327</td>
<td>0) 0.434 us native_load_sp0();</td>
</tr>
<tr>
<td>185.004328</td>
<td>0) 0.465 us native_load_tls();</td>
</tr>
</tbody>
</table>

Here!
Case1: Check the kernel source (Second attempt)

- Update curr_rt() defined in kernel/sched.c

```c
static void update_curr_rt(struct rq *rq)
{
    struct task_struct *curr = rq->curr;
    struct sched_rt_entity *rt_se = &curr->rt;
    struct rt_rq *rt_rq = rt_rq_of_se(rt_se);
    u64 delta_exec;

    if (!task_has_rt_policy(curr))
        return;

    delta_exec = rq->clock - curr->se.exec_start;
    if (unlikely((s64)delta_exec < 0))
        delta_exec = 0;

    schedstat_set(curr->se.exec_max, max(curr->se.exec_max, delta_exec));
    curr->se.sum_exec_runtime += delta_exec;
    account_group_exec_runtime(curr, delta_exec);
    curr->se.exec_start = rq->clock;
    cpuacct_charge(curr, delta_exec);
    sched_rt_avg_update(rq, delta_exec);
}
```

Maybe here!
But different result.
**Case1: Check the trace log (Third attempt)**

```c
schedule() {
    __schedule() {
        rou_qsctr_inc();
        ... 
        pre_schedule_rt() {
            pull_rt_task();
        }
        put_prev_task_rt() {
            update_curr_rt() {
                sched_avg_update();
            }
        }
        pick_next_task_fair();
        pick_next_task_idle();
        perf_counter_task_sched_out();
    }
}
```

*Here!*
Case1: Check the kernel source (Third attempt)

- Pick_next_task() also defined in kernel/sched.c

```c
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
  ...
  if (likely(rq->nr_running == cfs.nr_running)) {
    p = fair_sched_class.pick_next_task(rq);
    if (likely(p))
      return p;
  }
  class = sched_class_highest;
  for (;;) {
    p = class->pick_next_task(rq);
    if (p)
      return p;
    ...
    class = class->next;
  }
}

asmlinkage void __sched __schedule(void)
{
  ...
  put_prev_task(rq, prev);
  next = pick_next_task(rq);
}
```

Different result again!
Case1: Summary of the evaluation results

- Latency occurs in different points each time and difficult to identify the cause
- Only occurs on specific hardware
- Maybe SMI (System Management Interrupt)
Learn a lesson from Case1

- One shot trace log is not enough
Case2: Evaluation environment

- CPU: ARM Coretex-A8
- Memory: 512MB
- Kernel: Linux 2.6.31.12-rt21
  - A function graph tracer patch applied
    http://elinux.org/Ftrace_Function_Graph_ARM
Case2: Summary of evaluation result (First attempt)

- Evaluated without other CPU workload

<table>
<thead>
<tr>
<th>Cycle [μs]</th>
<th>Min.</th>
<th>Ave.</th>
<th>Max</th>
<th># of DL misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>11</td>
<td>19.025</td>
<td>921</td>
<td>3018</td>
</tr>
<tr>
<td>500</td>
<td>11</td>
<td>19.458</td>
<td>684</td>
<td>5026</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
<td>23.011</td>
<td>717</td>
<td>0</td>
</tr>
</tbody>
</table>
Case 2: Summary of evaluation result (First attempt)

- Evaluated with approx 60% CPU workload

### Cycle: 300us

<table>
<thead>
<tr>
<th>Cycle [μs]</th>
<th>Min.</th>
<th>Ave.</th>
<th>Max</th>
<th># of DL misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>13</td>
<td>34.165</td>
<td>980</td>
<td>3018</td>
</tr>
<tr>
<td>500</td>
<td>13</td>
<td>33.234</td>
<td>760</td>
<td>5025</td>
</tr>
<tr>
<td>1000</td>
<td>12</td>
<td>35.014</td>
<td>714</td>
<td>0</td>
</tr>
</tbody>
</table>

### Cycle: 500us

### Cycle: 1000us
Case2: Check the trace log (First attempt)

Here!

TOSHIBA
Leading Innovation

Yoshitake Kobayashi - Embedded Linux Conference 2012 -
Case2: Identify the latency  (First attempt)

This function is used to process and free completed qtds for a qh. (drivers/usb/host/ehci-q.c)
Case2: Cause and possible solution

- **Cause**
  - Too many qh_completions() function call
  - Default polling threshold is 100ms

- **Possible solution**
  - Change the polling threshold to 10ms

```diff
diff --git a/drivers/usb/host/ehci-hcd.c b/drivers/usb/host/ehci-hcd.c
index 0c9b7d2..db2efd2 100644
--- a/drivers/usb/host/ehci-hcd.c
+++ b/drivers/usb/host/ehci-hcd.c
@@ -83,7 +83,7 @@ static const char hcd_name [] = "ehci_hcd";
 #define EHCI_TUNE_FLS 2 /* (small) 256 frame schedule */
 #define EHCI_IAA_MSECS 10 /* arbitrary */
-#define EHCI_IO_JIFFIES (HZ/10) /* io watchdog > irq_thresh */
+#define EHCI_IO_JIFFIES (HZ/100) /* io watchdog > irq_thresh */
 #define EHCI_ASYNC_JIFFIES (HZ/20) /* async idle timeout */
 #define EHCI_SHRINK_FRAMES 5 /* async qh unlink delay */
```
Case2: Summary of evaluation result (Second attempt)

- Evaluated without other CPU workload

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Min</th>
<th>Ave.</th>
<th>Max</th>
<th># of DL miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>11</td>
<td>17.590</td>
<td>209</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>11</td>
<td>17.583</td>
<td>117</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
<td>17.610</td>
<td>154</td>
<td>0</td>
</tr>
</tbody>
</table>
Case2: Summary of evaluation result (Second attempt)

- Evaluated with approx 60% CPU workload

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Min</th>
<th>Ave</th>
<th>Max</th>
<th># of DL miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>13</td>
<td>33.365</td>
<td>259</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>13</td>
<td>29.695</td>
<td>228</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>12</td>
<td>33.222</td>
<td>199</td>
<td>0</td>
</tr>
</tbody>
</table>

Maximum latency reduced to 1/4
Learn a lesson from Case2

- Need to check what kind of functions are called
- Need to count the number of callees
How to stabilize latency less than 50µs

- Required spec is the following:
  - Single process
  - No network, No USB devices, No graphic device, No storage device
  - latency < 50µs

- Evaluation result (cyclictest: cycle=250µs)

![Graph showing latency and number of samples]

Why?
Second mistake

- Deadline miss ratio: 100%
- Function graph tracing requires additional overhead
Evaluate the latency with trace ON

This result seems to have same characteristics

Trace with the following setting

- Stop if the latency is more than 790µs
- Stop if the latency is between 575 and 585
Conclusion

- Evaluating with too much workload is not a good idea
  - Required log data is lost

- One shot trace log is not enough
  - Need to check performance characteristics between trace on and off
  - Need to check if latency occurs at similar points

- Require some creative thinking to identify the cause
  - Check if the same function takes different execution time
    - Statistics approach may be applied
  - Check all functions included in specific area to identify each function’s overhead
  - Eliminate callee’s overhead to calculate pure execution time for each function’s algorithm