Measuring Function Duration with Ftrace

By Tim Bird
Sony Corporation of America
<tim.bird (at) am.sony.com>
Outline

• Introduction to Ftrace
• Adding function graph tracing to ARM
• Duration Filtering
  – Trace coverage duration analysis
• Measuring kernel boot
• Post-trace analysis tools
• Performance impact
• Resources
Introduction to Ftrace

• What is Ftrace?
• Overview of operation
  − Instrumentation
  − Runtime operation
  − Data capture
  − Trace log output
• Function duration tracing
What is Ftrace?

• Ftrace is the first generic tracing system to get mainlined (Hurray!!)
  – Mainlined in 2.6.27
  – Derived from RT-preempt latency tracer

• Provides a generic framework for tracing
  – Infrastructure for defining tracepoints
  – Ability to register different kinds of tracers
  – Specialized data structure (ring buffer) for trace data storage
Overview of FTrace Operation

• Instrumentation
  – Explicit
    • Tracepoints defined by declaration
    • Calls to trace handler written in source code
  – Implicit
    • Automatically inserted by compiler
      – Uses gcc ‘-pg’ option
    • Inserts call to ‘mcount’ in each function prologue
    • Easy to maintain – no source code modifications
    • Only practical way to maintain 20,000+ tracepoints
mcount Routine

• ‘mcount’ is called by every kernel function
  – Except inlines and a few special functions
• Must be a low-overhead routine
• Incompatible with some compiler optimizations
  – E.g. cannot omit frame-pointers on ARM
  – Compiler disables some optimizations automatically
  – Works with ARM EABI
  – Analysis of assembly indicates that mcount callers have well-defined frames

• Misc note:
  – New mcount routine (_gnu_mcount) is coming
Code to Call mcount

```
00000570 <sys_sync>:  
570: e1a0c00d mov ip, sp  
574: e92dd800 stmdb sp!, {fp, ip, lr, pc}  
578: e24cb004 sub fp, ip, #4 ; 0x4  
57c: e3a00001 mov r0, #1 ; 0x1  
580: ebfffffa0 bl 408 <do_sync>  
584: e3a00000 mov r0, #0 ; 0x0  
588: e89da800 ldmia sp, {fp, sp, pc}  
```

```
00000570 <sys_sync>:  
570: e1a0c00d mov ip, sp  
574: e92dd800 stmdb sp!, {fp, ip, lr, pc}  
578: e24cb004 sub fp, ip, #4 ; 0x4  
57c: e1a0c00e mov ip, lr  
580: ebfffffff bl 0 <mcount>  
584: 0000028 andeq r0, r0, r8, lsr #32  
588: e3a00001 mov r0, #1 ; 0x1  
58c: ebffff9d bl 408 <do_sync>  
590: e3a00000 mov r0, #0 ; 0x0  
594: e89da800 ldmia sp, {fp, sp, pc}  
```
Trace setup at run-time

• Pseudo-files in debugfs
  - e.g. mount debugfs -t debugfs /debug

• Select a tracer
  - e.g. echo function_duration >current_tracer

• Set tracing parameters
  - e.g. echo 100 >tracing_threshold
  - echo duration-proc >trace_options
Trace Data Capture

• Ring Buffer
  - Specialized structure for collecting trace data
    • Manages buffer as list of pages
  - Latest version is lockless for writing
    • Ability to atomically reserve space for an event
  - Automatic timestamp management
  - Per-cpu buffers
    • Avoids requiring cross-CPU synchronization
    • Also avoids cache collisions
      - Very important for performance
Trace Output

• Output is human readable text
  – No special tools required to collect trace data

• Examples:
  – cat trace
    • Returns EOF at end of trace data
  – cat trace_pipe | grep foo >log.txt
    • Blocks at end of trace data

• Quick enable/disable
  – echo 0 >tracing_enabled
Ring Buffer Operations

- **ring_buffer_lock_reserve**
  - Atomically reserve space in buffer
- **ring_buffer_event_data**
  - Get pointer to place to fill with data
- **ring_buffer_unlock_commit**
  - Commit event data
- **ring_buffer_discard_commit**
  - Discard reserved data space
Function duration tracing

• Traces function entry and exit

• What is it good for?
  - See relationship between functions
    • Is a GREAT way to learn about kernel
    • Find unexpected/abnormal code paths
  - Measure function duration
    • Find long latencies and performance problems

• But, the -pg option only instruments function entry
Hooking function exit

• Normal ‘function’ tracer just traces function entry capture

• To capture function exit, a trampoline is used
  - mcount:
    • Saves real return address
    • Replaces return address with address of trampoline
  - In exit tracer, return to the real return address
Diagram of Trampoline

- **Caller**
- **Function**
- **Func entry Tracer**
- **mcount**
- **Func exit Tracer**
- **Stack**
  - ret addr
- **Thread info**
  - struct ret_stack
    - caller 1
    - caller 2
Why Filter by Duration?

• To extend the capture duration time
  – By reducing, at runtime, the amount of trace data
  – Without a duration filter, you can only capture about 0.4 seconds worth of data

• To see only long-duration functions
  – When looking for long-lasting functions, you don’t need to see the short ones (in most cases)
Filtering by Duration - first try

• Added duration filter to 'function_graph' tracer

• Method:
  − Compare duration to threshold
  − Discard function entry and exit events

• Its easy to discard exit event
  − Just don’t commit data

• Trickier to discard entry event
  − `ring_buffer_event_discard()` converts event to padding if subsequent events have been committed to buffer
    • Wastes a lot of space
    • Severely constrains the time coverage for a trace
Filtering by Duration - second try

- Created new 'function_duration' tracer

- Method:
  - Don't save function entries to trace log at all
    - Only save call time on function return stack
  - At function exit, compare duration to threshold
  - Omit exit entry events for short duration functions

- Results in simpler, *and faster* code

- Only issue is that log is displayed in order of function exit (not function entry)
  - Can be solved with a simple sort on trace output
Trace time coverage: graph vs duration tracer

<table>
<thead>
<tr>
<th>Tracer</th>
<th>Duration Filter Value</th>
<th>Total Function Count</th>
<th>Time Covered by Trace</th>
<th>Trace Event Count</th>
<th>Projected Trace Time Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>0</td>
<td>3.295M</td>
<td>0.39 s</td>
<td>27316</td>
<td>0.39 s</td>
</tr>
<tr>
<td>Graph</td>
<td>1000</td>
<td>3.310M</td>
<td>1.29 s</td>
<td>26630</td>
<td>1.39 s</td>
</tr>
<tr>
<td>Graph</td>
<td>100000</td>
<td>3.309M</td>
<td>1.34 s</td>
<td>26438</td>
<td>1.34 s</td>
</tr>
<tr>
<td>Duration</td>
<td>0</td>
<td>2.906M</td>
<td>0.38 s</td>
<td>27597</td>
<td>0.38 s</td>
</tr>
<tr>
<td>Duration</td>
<td>1000</td>
<td>2.788M</td>
<td>21.70 s †</td>
<td>3943</td>
<td>154.00 s</td>
</tr>
<tr>
<td>Duration</td>
<td>100000</td>
<td>2.795M</td>
<td>21.31 s †</td>
<td>208</td>
<td>2868.00 s</td>
</tr>
</tbody>
</table>

† The test finished without filling the buffer.  

= Estimate
Example of Use

```
$ mount debugfs -t debugfs /debug
$ cd /debug/tracing
$ cat available_tracers
function_graph function_duration function sched_switch nop
$ echo 0 >tracing_enabled
$ echo 100 >tracing_thresh
$ echo function_duration >current_tracer
$ echo 1 >tracing_enabled ; do \
   ls /bin | sed s/a/z/g ; done ; echo 0 >tracing_enabled
$ echo duration-proc >trace_options
$ cat trace >/tmp/trace.txt
$ cat /tmp/trace.txt | sort -k3 > /tmp/trace.txt.sorted
```
## Function Duration Results (sorted)

```plaintext
# tracer: function_duration
#
# CPU TASK/PID CALLTIME DURATION FUNCTION CALLS
# | | | | | |
0) sed-562 | 502.854252393 | ! 436.833 us | bprm_mm_init
0) sed-562 | 502.854254893 | ! 321.500 us | mm_alloc
0) sed-562 | 502.854270893 | ! 296.500 us | mm_init
0) sed-562 | 502.854279393 | ! 266.166 us | get_pgd_slow
0) sed-562 | 502.854744059 | ! 229.500 us | prepare_binprm
0) sed-562 | 502.854765393 | ! 198.666 us | kernel_read
0) sed-562 | 502.854769226 | ! 183.333 us | vfs_read
0) sed-562 | 502.854780393 | ! 142.000 us | do_sync_read
0) sed-562 | 502.854785559 | ! 120.667 us | nfs_file_read
0) sed-562 | 502.854982393 | ! 538.000 us | copy_strings_kernel
0) sed-562 | 502.854985726 | ! 521.667 us | copy_strings
0) sed-562 | 502.854993893 | ! 470.000 us | get_arg_page
0) sed-562 | 502.854997226 | ! 455.500 us | get_user_pages
0) sed-562 | 502.855000059 | ! 421.667 us | __get_user_pages
0) sed-562 | 502.855031393 | ! 285.666 us | handle_mm_fault
0) sed-562 | 502.855037726 | ! 101.833 us | __pte_alloc
```
Measuring kernel boot

- Can start tracer early in boot sequence
- Use “ftrace=function_duration” on kernel command line
  - Can specify “tracing_thresh=<value>”
- Tracer is initialized after kernel core (timers, memory, interrupts), but before all initcalls
  - On my hardware, tracer starts about 50 milliseconds after start_kernel()
- Had to restore instrumentation to functions in _init segment
- Need to stop trace after point of interest
Introducing a stop trigger

- Use "trace_stop_fn=<func_name>" on kernel command line
- Trace stops on ENTRY to named function
- To use, figure out a fairly unique function, which runs immediately after the area of interest
- An initcall works very well
  - Initcall functions have unique names in kernel
Example of early boot trace

• To trace most of kernel boot:
  – Add this to the kernel command line:
    • “ftrace=function_duration tracing_thresh=200
    trace_stop_fn=run_init_process”
  – If the trace doesn't cover the whole boot, increase tracing_thresh and try again

• To trace an individual initcall:
  – Find initcall following the one you are interested in
    • Can use initcall_debug on kernel command line
      • ex: pty_init follows tty_init
  – Kernel command line:
    • “ftrace=function_duration trace_stop_fn=pty_init”
Post-trace analysis

- fdd tool is provided to analyze data
- What fdd shows:
  - function counts, total time, average duration
  - sub-routine with the longest duration, how many times it was called
  - Local time = total time minus sub-routine total time
    * Is approximately the cost of the local execution of a function
- Notes:
  - Total time may be wrong if process is scheduled out or if a filter was active
    * May need an option to subtract time that function was scheduled out
  - You can filter, sort, select output columns, etc.
```markdown
fdd Output

```markdown

```bash
$ fdd /tmp/trace.txt -n 15

<table>
<thead>
<tr>
<th>Function</th>
<th>Count</th>
<th>Time</th>
<th>Average</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedule</td>
<td>59</td>
<td>1497735270</td>
<td>25385343</td>
<td>1476642939</td>
</tr>
<tr>
<td>sys_write</td>
<td>56</td>
<td>1373722663</td>
<td>24530761</td>
<td>2892665</td>
</tr>
<tr>
<td>vfs_write</td>
<td>56</td>
<td>1367969833</td>
<td>24428032</td>
<td>3473173</td>
</tr>
<tr>
<td>tty_write</td>
<td>54</td>
<td>1342476332</td>
<td>24860672</td>
<td>1212301170</td>
</tr>
<tr>
<td>do_path_lookup</td>
<td>95</td>
<td>1076524931</td>
<td>11331841</td>
<td>34682198</td>
</tr>
<tr>
<td>__link_path_walk</td>
<td>99</td>
<td>1051351737</td>
<td>10619714</td>
<td>6702507</td>
</tr>
<tr>
<td>rpc_call_sync</td>
<td>87</td>
<td>1033211085</td>
<td>11875989</td>
<td>1700178</td>
</tr>
<tr>
<td>path_walk</td>
<td>94</td>
<td>1019263902</td>
<td>10843233</td>
<td>3425163</td>
</tr>
<tr>
<td>rpc_run_task</td>
<td>87</td>
<td>960080412</td>
<td>11035407</td>
<td>2292360</td>
</tr>
<tr>
<td>rpc_execute</td>
<td>87</td>
<td>936049887</td>
<td>10759194</td>
<td>2316635</td>
</tr>
<tr>
<td>__rpc_execute</td>
<td>87</td>
<td>932779083</td>
<td>10721598</td>
<td>11383353</td>
</tr>
<tr>
<td>do_lookup</td>
<td>191</td>
<td>875826405</td>
<td>4585478</td>
<td>9510659</td>
</tr>
<tr>
<td>call_transmit</td>
<td>100</td>
<td>785408085</td>
<td>7854080</td>
<td>5871339</td>
</tr>
<tr>
<td>__nfs_revalidate_inode</td>
<td>38</td>
<td>696216223</td>
<td>18321479</td>
<td>1652173</td>
</tr>
<tr>
<td>nfs_procgetattr</td>
<td>38</td>
<td>690552053</td>
<td>18172422</td>
<td>1234634</td>
</tr>
</tbody>
</table>
```
Performance issues

- Overhead of tracing can be big
  - Average function duration = 3.22 μs
  - Overhead = 11.4 microseconds per function

- Use a CPU-bound test to measure overhead
  - “find /sys >/dev/null”
  - With an I/O-bound test (or a real-workload), the ratio of overhead to average function duration should be lower

- With ftrace compiled into kernel, but the 'NOP' tracer selected, the overhead in my test was about 12%
## Overhead Measurements

<table>
<thead>
<tr>
<th>Tracer status</th>
<th>Elapsed time</th>
<th>Function count</th>
<th>Time per function</th>
<th>Overhead per function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE=n</td>
<td>8.85 s</td>
<td>2.751M *</td>
<td>3.22 us</td>
<td>-</td>
</tr>
<tr>
<td>Tracer=nop</td>
<td>9.94 s</td>
<td>2.757M *</td>
<td>3.61 us</td>
<td>0.39 us</td>
</tr>
<tr>
<td>Tracer=duration, enabled=0</td>
<td>21.57 s</td>
<td>2.816M</td>
<td>7.66 us</td>
<td>4.44 us</td>
</tr>
<tr>
<td>Tracer=duration, thresh=0</td>
<td>42.55 s</td>
<td>2.911M</td>
<td>14.62 us</td>
<td>11.40 us</td>
</tr>
<tr>
<td>thresh=1</td>
<td>42.80 s</td>
<td>2.923M</td>
<td>14.64 us</td>
<td>11.42 us</td>
</tr>
<tr>
<td>thresh=10</td>
<td>30.87 s</td>
<td>2.850M</td>
<td>10.83 us</td>
<td>7.61 us</td>
</tr>
<tr>
<td>thresh=-100</td>
<td>24.58 s</td>
<td>2.824M</td>
<td>8.70 us</td>
<td>5.48 us</td>
</tr>
<tr>
<td>thresh=1000</td>
<td>21.40 s</td>
<td>2.802M</td>
<td>7.64 us</td>
<td>4.42 us</td>
</tr>
<tr>
<td>thresh=10000000</td>
<td>21.43 s</td>
<td>2.803M</td>
<td>7.64 us</td>
<td>4.42 us</td>
</tr>
</tbody>
</table>

* = estimated
Roadmap and future work

• Mainline try 2
  – Patches:
    • ARM function graph assembly support
    • function_duration tracer
    • changes to ftrace for use at boot time

• Need to use functionality to improve bootup time
  – Have already identified a few problems
    • call_usermode_helper (*may already be done*)
    • ip_auto_config
References

• Ftrace tutorial at OLS 2008
  − http://people.redhat.com/srostedt/ftrace-tutorial.odp

• “The world of Ftrace” at Spring 2009 LF Collaboration Summit
  − http://people.redhat.com/srostedt/ftrace-world.odp

• Patches and tools for this talk
  − http://elinux.org/Ftrace_Function_Graph_ARM
Questions & Answers