Measuring Function Duration with Ftrace

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

On ARM
Outline

• Introduction to Ftrace
• Adding function graph tracing to ARM
• Duration Filtering
  − Optimizing the discard operation
• 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 graph 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
  - Assembly analysis indicates that mcount callers have well-defined frames
• Misc note:
  - New mcount routine (GNUC_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: ebfffffe bl 0 <mcount>
584: 00000028 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_graph >current_tracer
- Set tracing parameters
  - e.g. echo 100 >tracing_threshold
  - echo funcgraph-abstime >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 graph 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

mcount

Func entry Tracer

Stack

ret addr

Thead_info

struct ret_stack

caller 1

caller 2
Filtering by Duration

• Compare duration to threshold
• Discard function entry and exit events
• 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
Optimizing Event Discard

• Normally, can’t discard events after other events are committed to buffer

• However, with duration filtering, if an event is filtered for duration, then all children functions are filtered also

• “Last event” in buffer is always function entry for current exit
  - Only have to “rewind” one event, which is relatively easy (and likely safe)
Results from optimized discard

<table>
<thead>
<tr>
<th>Discard operation</th>
<th>Duration Filter Value</th>
<th>Total Function Count</th>
<th>Time covered by Trace</th>
<th>Trace event count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discard_event</td>
<td>0</td>
<td>3.292M</td>
<td>0.39 s</td>
<td>27392</td>
</tr>
<tr>
<td>Discard_event</td>
<td>1000</td>
<td>3.310M</td>
<td>1.29 s</td>
<td>26630</td>
</tr>
<tr>
<td>Discard_event</td>
<td>100000</td>
<td>3.309M</td>
<td>1.34 s</td>
<td>26438</td>
</tr>
<tr>
<td>Rewind_tail</td>
<td>0</td>
<td>3.295M</td>
<td>0.39 s</td>
<td>27316</td>
</tr>
<tr>
<td>Rewind_tail</td>
<td>1000</td>
<td>3.327M</td>
<td>31.26 s</td>
<td>35565</td>
</tr>
<tr>
<td>Rewind_tail</td>
<td>100000</td>
<td>3.328M</td>
<td>79.44s †</td>
<td>1669</td>
</tr>
</tbody>
</table>

† The test only lasted 79 seconds—extrapolating the results yields a trace coverage time of 27 minutes
Example of Use

$ mount debugfs -t debugfs /debug
$ cd /debug/tracing
$ cat available_tracers
function_graph function sched_switch nop
$ echo 0 >tracing_enabled
$ echo 1000 >tracing_thresh
$ echo function_graph >current_tracer
$ echo 1 >tracing_enabled
$ for i in 'seq 1 10'; do ls /bin | sed s/a/z/g; done
$ echo 0 >tracing_enabled
$ echo funcgraph-abstime >trace_options
$ echo funcgraph-proc >trace_options
$ cat trace
### Function Graph Results

```bash
# tracer: function_graph
#
# TIME     CPU  TASK/PID  DURATION  FUNCTION CALLS
# 193.719625 | 0 | ls-556 | |  sys_lstat64() {
# 193.719641 | 0 | ls-556 | |  vfs_lstat() {
# 193.719650 | 0 | ls-556 | |  vfs_fstatat() {
# 193.719660 | 0 | ls-556 | |  user_path_at() {
# 193.719722 | 0 | ls-556 | |  do_path_lookup() {
# 193.719755 | 0 | ls-556 | |  path_walk() {
# 193.719777 | 0 | ls-556 | |  __link_path_walk() {
# 193.719826 | 0 | ls-556 | |  do_lookup() {
# 193.719855 | 0 | ls-556 | |  nfs_lookup_revalidate() {
# 193.719883 | 0 | ls-556 | |  )
# 193.719946 | 0 | ls-556 | |  )
# 193.719965 | 0 | ls-556 | |  )
# 193.719986 | 0 | ls-556 | |  )
# 193.720016 | 0 | ls-556 | |  )
# 193.720045 | 0 | ls-556 | |  )
# 193.720069 | 0 | ls-556 | |  )
# 193.720099 | 0 | ls-556 | |  )
# 193.720108 | 0 | ls-556 | |  )
# 193.720139 | 0 | ls-556 | |  )
# 193.720315 | 0 | ls-556 | |  sys_lstat64() {
# 193.720337 | 0 | ls-556 | |  vfs_lstat() {
# 193.720346 | 0 | ls-556 | |  vfs_fstatat() {
# 193.720357 | 0 | ls-556 | |  user_path_at();
# 193.720410 | 0 | ls-556 | |  )
# 193.720419 | 0 | ls-556 | |  )
# 193.720452 | 0 | ls-556 | |  )
```
Post-trace analysis

- Using ftd to analyze data
  - Measuring function counts
  - Measuring “local time”
    - wall time minus sub-routine wall time
    - May be wrong if we block
      - Need an option to subtract time that function was scheduled out
  - Filter, sort, select output columns, etc.
# Ftd Output

<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_proc_getattr</td>
<td>38</td>
<td>690552053</td>
<td>18172422</td>
<td>1234634</td>
</tr>
</tbody>
</table>
Performance issues

• Overhead of tracing
  - Can be substantial
    • Average function duration = 1.72 μs
    • Overhead = 18.89 microseconds per function
  - Test used was CPU-bound
    • find /sys >/dev/null
    • With I/O bound test, ratio of overhead to average function length should be much lower
## 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>9.25 s</td>
<td>2.91M</td>
<td>1.72 us</td>
<td>-</td>
</tr>
<tr>
<td>Nop</td>
<td>10.30 s</td>
<td>2.92M</td>
<td>2.05 us</td>
<td>0.33 us</td>
</tr>
<tr>
<td>Graph disabled</td>
<td>19.85 s</td>
<td>2.98M</td>
<td>5.22 us</td>
<td>3.50 us</td>
</tr>
<tr>
<td>Graph active</td>
<td>72.15 s</td>
<td>3.29M</td>
<td>20.61 us</td>
<td>18.89 us</td>
</tr>
</tbody>
</table>
Roadmap and future work

• Mainline stuff
  – ARM function graph tracing
  – Duration filtering
    • Recently rejected – back to the drawing board??

• Need to use functionality to improve bootup time
Measuring kernel boot

- Requirements for using ftrace in early boot
  - Availability of clock source
  - Static(?) definition of trace parameters
    - Start location for tracing (optimally start_kernel)
  - Initialization of ring buffer and tracer registration
    - Would be nice to do at compilation time, but that’s hard!
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