Measuring Function Duration with Ftrace

Tim Bird
Sony Corporation of America
tim.bird@am.sony.com

Abstract

FTrace is a relatively new kernel tool for tracing function execution in the Linux kernel. Recently, FTrace added the ability to trace function exit in addition to function entry. This allows for measurement of function duration, which adds an incredibly powerful tool for finding time-consuming areas of kernel execution.

In this paper, the current state of the art for measuring function duration with FTrace is described. This includes recent work to add a new capability to filter the trace data by function duration, and tools for analyzing kernel function call graphs and visualizing kernel boot time execution.

Introduction

Analyzing a running operating system kernel can be a difficult task. In the 2.6.27 version of the kernel, a powerful tracing mechanism called Ftrace was added to mainline Linux. Ftrace provides some very nice facilities for instrumenting the kernel, recording trace data, and outputting the data to user space.

The Ftrace system provides a generic tracing framework in the kernel, upon which several different kinds of tracers can be implemented. Different kinds of tracers utilize different methods of instrumenting the kernel code and different data collection algorithms.

Ftrace supports the ability do basic *function* tracing, which consists of recording information at the time of entry to every function executed in the kernel. Additionally, on some architectures, Ftrace supports the ability to perform *function graph* tracing, which involves tracking not just function entry but also function exit, and the ability to measure function duration. This is is useful to find performance problems and latency problems in the kernel.

This paper presents work by the author to add function graph tracing to the ARM architecture. This includes a description of the mechanisms used and some of the issues involved on the ARM architecture.

Also, this paper describes the author's efforts to add duration filtering to the function graph tracer. Even on a relatively slow processor, the kernel executes many thousands of functions per second. Without filtering, the length of time that data can be captured in the trace log without loss is very limited. By adding duration filtering, it is possible to greatly extend the duration of a trace, to capture more events of interest and to help isolate problem areas.

1 Overview of Ftrace Operation

1.1 Instrumentation

Ftrace operates by adding tracepoints to the Linux kernel. The insertion into the Linux kernel of locations where tracing information is recorded is referred to as *instrumentation*. Instrumentation comes in two main forms—explicitly declared tracepoints, and implicit tracepoints. Explicit tracepoints consist of developer-defined declarations which specify the location of the tracepoint, and additional information about what data should be collected at a particular trace site. Implicit tracepoints are placed into the code automatically by the compiler, either due to compiler flags or by developer redefinition of commonly used macros.

Function tracing and function graph tracing utilize implicit instrumentation. The kernel consists of many thousands of C functions, and it would be extremely impractical to maintain explicit tracepoint definitions for all of them. To instrument functions implicitly, when the kernel is configured to support function tracing the kernel build system adds `-pg` to the flags used with the compiler. This causes the compiler to add code to

the prologue of each function, which calls a special assembly routine called `mcount`. This compiler option is specifically intended to be used for profiling and tracing purposes.

Figures 1 and 2 show the ARM assembly code generated when compiling the short routine `sys_sync()` both with and without the `-pg` compiler flag. The assembly code was produced from the compiled object file with the command: `arm-eabi-objdump -S fs/sync.o >fs/sync.S` Comparing the two shows that the mcount call only takes a few extra instructions.

```
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:  ebfffa0    bl      408 <do_sync>
 584:  e3a00000   mov     r0, #0  ; 0x0
 588:  e89da800   ldmia   sp, {fp, sp, pc}
```

Figure 1: ARM code *without* call to 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:  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}
```

Figure 2: ARM code *with* call to mcount

The `mcount` routine is in platform-specific assembly, located in the file `arch/arm/kernel/entry-common.S`, for the ARM platform. It is called every time a function is entered. Because of this, it is important that the routine have very low overhead, especially when tracing is disabled. Note that on some platforms, Ftrace includes the capability to use "dynamic tracepoints", whereby the tracepoints are replaced with 'nop' instructions at runtime, to reduce overhead when not tracing. This is a very neat capability, which dramatically reduces overhead and makes it feasible to leave tracing configured on even for some production systems. However, detailed discussion of this capability is outside the scope of this paper.

Another issue with use of `mcount` is that it is incompatible with certain kinds of compiler optimizations. `mcount` must be called with a consistent stack frame and frame pointer, in order for it to operate correctly. Some compiler optimizations produce stack frames, frame pointers, or call sequences that would cause `mcount` to be inaccurate, or worse, to function incorrectly. For example, on the ARM platform, the kernel must be compiled to use frame-pointers in order for function tracing to work correctly. That is, you cannot use the `-fomit-frame-pointers` compiler option.

Luckily, when the `-pg` compiler option is used, the gcc compiler automatically disables several optimizations which it might normally perform. Also, the kernel configuration system automatically adjusts compiler flags at build time to avoid conflicts between tracing options and optimization options.

1.2 Tracing at Runtime

At runtime, tracing is disabled until enabled by the user. In this situation, the `mcount` routine returns as quickly as possible to the instrumented function, and kernel processing continues. When tracing is enabled, `mcount` calls the function corresponding to the user-selected tracer, which then records information and makes an entry in the trace log.

Tracing can be enabled by the user by the manipulation of pseudo-files in the debug file system. The user can select what tracer to activate, and also set various tracing parameters. Files in the `Documentation/trace` directory describe the pseudo-files that are presented by Ftrace, the different tracers, and what parameters can be used by each one. In general, there are files for initiating and suspending a trace, adjusting the trace log size, for setting parameters for trace-time filtering, and for customizing the format of the trace log output.

1.3 Trace Data Capture

The trace log is kept in a new kernel data structure called the ring buffer. This data structure is specifically designed for holding trace data, for quick and lockless data entry, and for simultaneous reader and writer access to the buffer.

The ring buffer provides automatic management of timestamps used with the trace data. Also, it provides

page-aligned, per-cpu buffers for holding trace data. A more detailed description of the ring buffers is outside the scope of this document, but see `Documentation/trace/ring-buffer-design.txt` for more information.

Note that to avoid locking operations, data entry into the ring buffer is done in steps. First, the data position is reserved in the buffer, using the function `ring_buffer_lock_reserve()`. The data position is reserved in an atomic fashion, to avoid a costly lock operation. (Note that the word "lock" in the function name is misleading.)

Next the data for the trace event is filled in. If the trace data is to be saved (the normal case), then `ring_buffer_unlock_commit()` is called to commit the data to the buffer. If for some reason the event data should not be saved, then `ring_buffer_discard_commit()` can be called to eliminate the event from the buffer. If no other data has been written to the buffer, the discard_commit operation can remove the data from the buffer. However, if other data has been written, `ring_buffer_discard_commit()` just marks the data so that it is ignored by the tracer output system. In the case of filtering, it is highly desirable to not merely mark the data, but to actually remove it from the buffer, to free up space for other event data. This will be discussed in more depth in Section 2.1.

1.4 Trace output

Finally, a user can access the trace data via more debugfs pseudo-files. Trace data is formatted in plain text, and intended to be easily readable by humans, as well as easily processable by post-trace analysis tools.

Trace data can be accessed either after a trace has completed, or during a trace run.

1.5 Function Graph Tracing

Function graph tracing is a form of function tracing where both the function entry and exit are tracked by the tracer. With "regular" function tracing, only function entry is traced. When both the entry and exit of functions are available, it is possible to see the relationship between functions. It is possible to reconstruct the complete graph of function calls for a particular operation in the kernel. This is very helpful to understand the operation of the kernel, and also to detect anomalies in kernel operation. Also, by measuring both entry and exit, it is possible to measure the duration of each function.

Function graph tracing utilizes the same compiler instrumentation as function tracing. However, using the `mcount` mechanism to capture the exit assembly routine of a function requires some tricky manipulation of the stack and call sequence. Since the `-pg` compiler option only adds instrumentation for function entry, the Ftrace system needs to adjust the register and stack conditions before returning to execute the instrumented function so that Ftrace can regain control when the function exits.

It does this with a return "trampoline". This is shown in Figure 3. When Ftrace is called on function entry, it records the real return address (the address that the instrumented function was called from) and saves it in the process' task structure. Because multiple functions will nest before the returns are processed, these are kept in a stack of return addresses. After Ftrace calls the function graph tracer, it replaces the return address (either on the stack or in a register, depending on the architecture and ABI being used) with the address of an Ftrace routine to handle the return trace. Then Ftrace returns to the instrumented routine so that it can execute. When the instrumented routine finishes and returns, instead of returning to it's original caller, it returns to Ftrace. Ftrace then calls the function graph tracer again, with the function exit tracepoint data. Then Ftrace retrieves the the real return address from the task structure, and returns to the real caller.

2 Adding Function Graph Tracing to ARM

Function graph tracing was originally developed on the x86 architecture. This section describes some of the isses encountered while adding support for this feature to the 2.6.30 Linux kernel, for the ARM platform.

Here is the list of problems encountered, and the solutions implemented to fix them.

1. Basic function tracing was supported for the ARM architecture, but testing revealed that the system hung when it was activated on my particular platform.

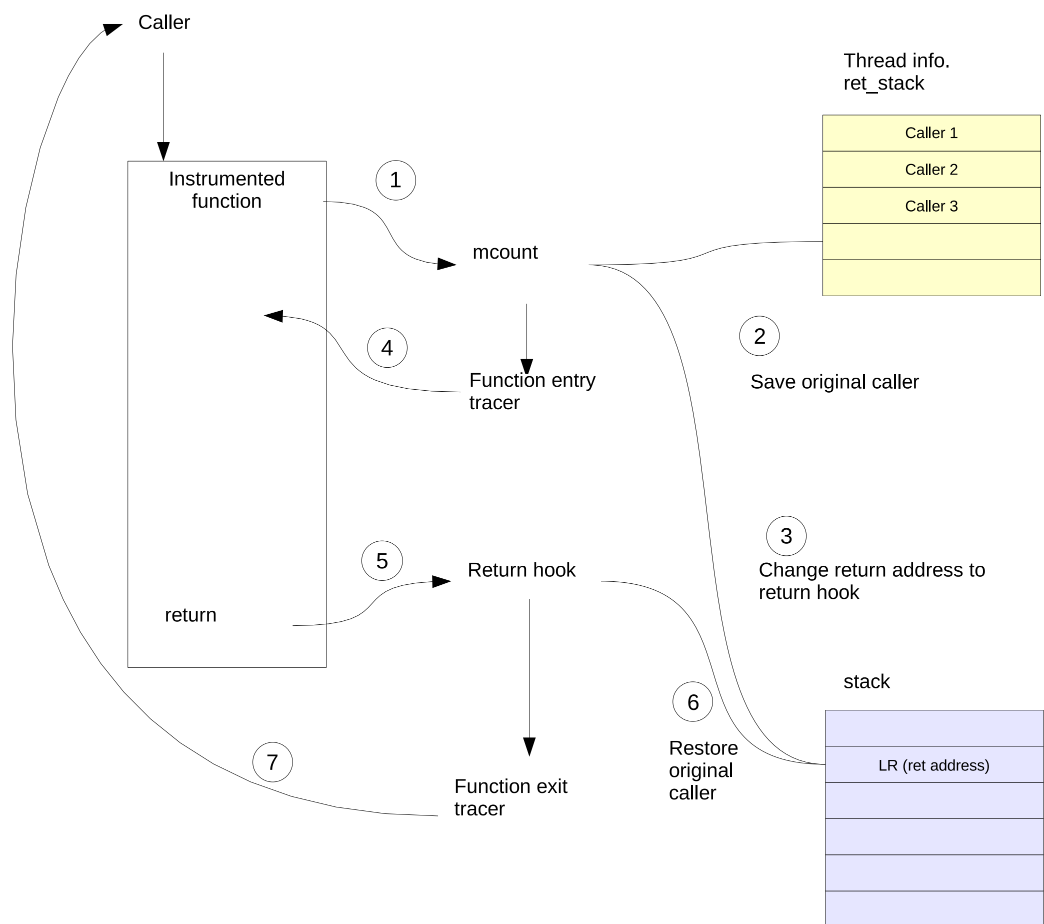

Figure 3: Mcount Handling and Return Trampoline

This was eventually determined to be an issue with recursion in the tracing code, due to some routines in the timestamping code path being instrumented. For this, I added the 'notrace' attribute to my platform-specific `sched_clock()` and all possible nested functions called by that routine.

2. Function graph tracing was implemented by doing the following:

   - Extending the ARM mcount routine in `arch/arm/kernel/entry-common.S` to check for and call a registered graph tracer function.
   - Adding a return trampoline for FTrace for ARM
   - Adding the return stack data structure to the task structure for processes
   - Adding an interrupt segment to the ARM linker directive file

   This was required because portions of the function graph display code examine routines to see whether they are interrupt functions. They do this by checking whether the function resides in the "interrupt segment". Note that I did not actually declare any routines to be interrupt routines, which is done with a qualifier on the function declaration.

3. I also modified the code to use a higher-resolution clock source for timestamps. The default clock source on my OMAP platform produced timestamps with a resolution of only 31 microseconds. This did not allow trace event times to be distinguished accurately. Luckily, there was another clock source (the `MPU_TIMER`, in my case) available that had higher resolution.

   To use this clock source for trace timestamps, I modified the OMAP `sched_clock()` to use the

different clock read routine for this timer.

4. I added duration filtering, using the existing `tracing_thresh` debugfs pseudo-file. The duration, calculated on function exit, was checked against this threshold and events discarded if the threshold was not met.

5. I optimized the duration filtering by adding routines to the ring buffer code to allow for discarding already-committed events. This change is discussed in the next section.

2.1 Optimizing the Discard of Trace Events

The function graph tracer places two events in the trace log for each function call. One event is logged for trace entry, and one for trace exit. The duration of the function is recorded in the trace exit event. In the first version of the duration filtering code, function exit events were discarded using `ring_buffer_discard_commit()`, and the function entry events were discarded using `ring_buffer_discard_event()`. `ring_buffer_discard_commit()` can usually back up the write pointer for the log, resulting in complete removal of the event from the trace buffer. However, `ring_buffer_discard_event()` just marks an entry as padding, and leaves it in the buffer.

This means that even though a trace log entry is not used in the trace output, it still occupies space in the trace log buffer, reducing the total number of events that can be held in the buffer at once.

Normally, previously committed entries in the trace log cannot be removed from the log, since subsequent entries cannot be moved to reclaim the space in the log without adding unacceptable overhead to the trace operation. So for post-commit filters, the only option is to mark the entry to be discarded as pad and leave it in the log.

However, the case of a duration filter is special, in that if a function is less than the duration threshold, all nested functions will also be less than the threshold. This means that, when using per-cpu trace buffers, and processing an exit event, if all nested function entry and exit events have been eliminated from the trace buffer, the last event in the trace buffer will always be the function entry event for the function that is currently exiting.

This observation allows for optimization of the ring buffer discard operation. If no other events follow the event to be discarded in the ring buffer, then it is possible to back up the commit and write pointers for the event (avoiding the unacceptably costly move operation), and eliminate the function entry event completely from the buffer.

I implemented a new ring buffer routine, called `ring_buffer_rewind_tail()` to do this more intelligent discard. In order to validate that `rewind_tail()` improved the length of the trace, compared to a regular `discard_event()`, I measured the amount of time I could capture in a trace, using different duration filter values with the different routines. All tests were performed with a buffer size of 1408KB.

The results of this testing are found in Table 1.

Discard operation	Duration filter value	Total function count	Time covered by trace	Trace event count
discard_event	0	3.292M	0.39 s	27392
discard_event	1000	3.310M	1.29 s	26630
discard_event	100000	3.309M	1.34 s	26438
rewind_tail	0	3.295M	0.39 s	27316
rewind_tail	1000	3.327M	31.26 s	35565
rewind_tail	100000	3.328M	†79.44 s	1669

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

Table 1: Comparison of Discard Operations

The results clearly show the efficacy of the optimized discard operation. When function entry events were left in the trace log, the log filled up after approximately 1.3 seconds, no matter what the duration filter value was. The low value (1669) for the event count in the last row of the table indicates that the test completed before the log became full. When almost all filtered function entry events are removed from the log using the `rewind_tail()` operation, the buffer can hold almost as many events of interest as the size of the buffer allows.

3 Example of Use

In this section, I describe use of the function graph tracer with duration filtering. For this example, I piped data between two Linux commands operating on file system data. The sample program is `busybox`, running the 'ls' and 'sed' commands, with sed executing a trivial

character replacement script. This was run in a loop 10 times.

Steps:

```
 $ 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
```

Figure 3 shows the first 25 lines of function graph trace output. Note that for this example I turned on the `funcgraph-abstime` and `funcgraph-proc` trace output options. Duration times for the functions are shown in units of microseconds, on the line containing the closing brace indicating the function exit.

Note that all functions in the log output took longer than 1000 microseconds to complete. Other functions which took less time than the `tracing_thresh` were filtered at runtime from the log.

3.1 Using 'ftd' to Analyze Data

To analyze system data, a post-processing tool called *ftd* was written. `ftd` stands for "function trace dump", and it is a script with the capability to show call counts and cumulative time for functions in a trace log. `ftd` is written in Python. If you are analyzing a trace log from an embedded target, it is recommended to move your trace log to a development host and run `ftd` there, rather than on the target.

`ftd` currently requires the absolute time and process information per trace line in the trace log, in order to work properly. Make sure these display options are set before retrieving the trace log data and using `ftd` on the data.

To retrieve the trace log data, use:

```
 $ cat trace >/tmp/trace-data.txt
```

To see a list of functions, sorted by total time spent in them, use:

```
 $ ftd /tmp/trace-data.txt
```

The first 10 lines of results for this command on some sample data are shown in Figure 4.

Other useful tasks that `ftd` can be used for include

- Sorting the function list by function count—the number of times that the function was called during the trace.
- Examining the *local time* of a function. The local time of a function is the elapsed time between the start and end of the function, minus the time spent in all functions called between the start and end of the function. Note that this includes not just children function called by this function, but also interrupts. Local time also includes the time spent in user space, and in other processes' kernel functions (ie, when the function's process is scheduled out.) So local time should be interpreted cautiously, with this understanding.
- Finding the subroutines called by functions the most times.

See `ftd -h` for usage help, for the command line options to use for these tasks.

4 Performance Measurements

The performance of various Ftrace configurations was measured, to get a sense of how much overhead tracing caused during kernel execution.

All results are for an OMAP 5912 processor running at 192 MHZ. The program I traced was a simple shell script consisting of:

```
for i in `seq 1 10` ; do
   echo $i ; find /sys >/dev/null ;
done
```

I found that this sequence was CPU-bound and spent most of it's time in the kernel. Raw data is not provided here, but the results of my testing showed that the overhead for function graph tracing is quite large. My tests generated approximately 3 million kernel function calls.

```
# 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     | ! 1028.500 us |                    _text();
  193.719946 |   0)     ls-556     | ! 1189.500 us |                  }
  193.719965 |   0)     ls-556     | ! 1258.500 us |                }
  193.719986 |   0)     ls-556     | ! 1775.167 us |              }
  193.720016 |   0)     ls-556     | ! 1874.333 us |            }
  193.720045 |   0)     ls-556     | ! 2018.167 us |          }
  193.720069 |   0)     ls-556     | ! 2143.000 us |        }
  193.720099 |   0)     ls-556     | ! 2397.000 us |      }
  193.720108 |   0)     ls-556     | ! 2415.167 us |    }
  193.720139 |   0)     ls-556     | ! 2478.334 us |  }
  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     | ! 1094.500 us |        user_path_at();
  193.720410 |   0)     ls-556     | ! 1738.167 us |      }
  193.720419 |   0)     ls-556     | ! 1758.500 us |    }
  193.720452 |   0)     ls-556     | ! 1825.500 us |  }
```

Figure 4: A function graph trace, with a duration filter of 1000 microseconds

The overhead per call, when tracing was active, was approximately 18.9 microseconds per call. The average time to execute a kernel function call during the test was 1.7 microseconds, so this represents a significant overhead. It should be noted that function graph tracing requires 2 calls through the tracer code per function called (one each for entry and exit).

Tracer Status	Elapsed Time	Function count†	Time per function	Overhead per function
TRACE=n	9.25 s	2.91M	1.72 us	-
nop	10.30 s	2.92M	2.05 us	0.33 us
graph disabled	19.85 s	2.98M	5.22 us	3.50 us
graph active	72.15 s	3.29M	20.61 us	18.89 us

†*Function counts were estimated, using data from other testing*

Table 2: Overhead of Function Graph Tracing

I found that the overhead per function with tracing disabled was about .3 microseconds per function. This added, on average, 19% overhead to kernel execution. The overhead for when function graph tracing was active was approximately 1100%. (That's right, over *one thousand* percent).

It should be noted that these are microbenchmarks, operating on a test designed to be kernel-function intensive, using non-blocking operations. The CPU utilization of these tests was always close to 100%. The overhead of using Ftrace on a system with a real user-space workload and real I/O would not be this high.

5 Future Work

The primary motivation for adding these features to Ftrace on ARM is to use them to help find problem areas in early boot. The next step in developing these features is to make it possible to use them during early kernel startup, to see which functions are taking a long time to execute, or which functions are called excessively during kernel startup.

Unfortunately, it may prove difficult to utilize Ftrace during early boot. Some of the requirements for doing this are listed below.

Function	Count	Time	Average	Local
schedule	70	1353560.333	19336.576	1337519.333
pipe_wait	1	526363.500	526363.500	56.535
preempt_schedule	320	414278.260	1294.620	3870.986
preempt_schedule_irq	17	294134.456	17302.027	-82.004
_text	465	278833.987	599.643	-58897.146
handle_IRQ_event	436	239268.153	548.780	88927.501
handle_mm_fault	396	228733.980	577.611	5986.491
local_bh_enable	1342	220684.604	164.445	16004.635
do_DataAbort	304	197972.822	651.226	61489.333
sys_wait4	4	144681.433	36170.358	144681.433

Figure 5: Output of `ftd` command

Requirements for using Ftrace in early boot:

- Early clock – The tracing systems depends on the availability of a clock source for timestamps very early in the boot sequence. On many platforms (X86, MIPS, and PPC), cpu registers are available from power-on which can be used for this purpose. On ARM, clocks are not initialized until after the kernel has already started running. This would limit how early tracing could start on ARM.

- Static trace parameters – Trace parameters, such as the start location for the trace, and the duration threshold, would have to be specified at compile time to be available from the earliest kernel execution points (ie. `start_kernel()`.)

- Static ring buffer – Possibly the most difficult problem is pre-initializing the ring buffer data structures to prepare them for receiving trace data. Other early-accessible data structures in the kernel, such as the kernel's printk log buffer, are much simpler and their initialization state can be prepared by the compiler.

Another area that should be worked on is performance. The overhead of Ftrace should be reduced. The generality of the Ftrace system, and utilization of generic clock routines and ring buffer code, add substatial overhead to a system that should be lightweight. Currently, Ftrace adds approximately 6 times more overhead, on the same hardware, than a function graph tracing system that the author used previously[1].

Finally, this work should be submitted (again) to the kernel mailing list for review and consideration for mainlining. The patches for the 2.6.31-rc1 kernel and the `ftd` may currently be found at `http://elinux.org/Ftrace_Function_Graph_ARM`.

6 Conclusion

The Ftrace system continues to be enhanced with new features and capabilities. This new duration filtering feature should help kernel developers continue to enhance the operation of the kernel. This effort is particularly focused on finding and reducing latencies in early boot, so that the Linux kernel can continue to be improved in the area of fast booting.

7 Biography

Tim Bird is a senior staff engineer for Sony Corporation of America, where he works on enhancing the Linux kernel for use in Sony Consumer Electronics products. He is also Chair of the Architecture Group of the Consumer Electronics Linux Forum (CELF), where he directs initiatives designed to enhance Linux for use in any CE products. He can be reached at `<tim dot bird (at) am dot sony dot com>`

[1] Kernel Function Trace - see `http://elinux.org/Kernel_Function_Trace`