Ftrace & its RISC-V port

Driving Innovations™

高魁良 (Alan Kao)
Andes Technology
2018/07/18

About me

- Alan (Quey-Liang) Kao
- 2007~2017 NTHU
  - BS, MS
- 2017.10~ Andes Technology
  - software engineer
  - RISC-V Linux
- Interests in Linux
  - virtualization, container, tracing, perf

Outline

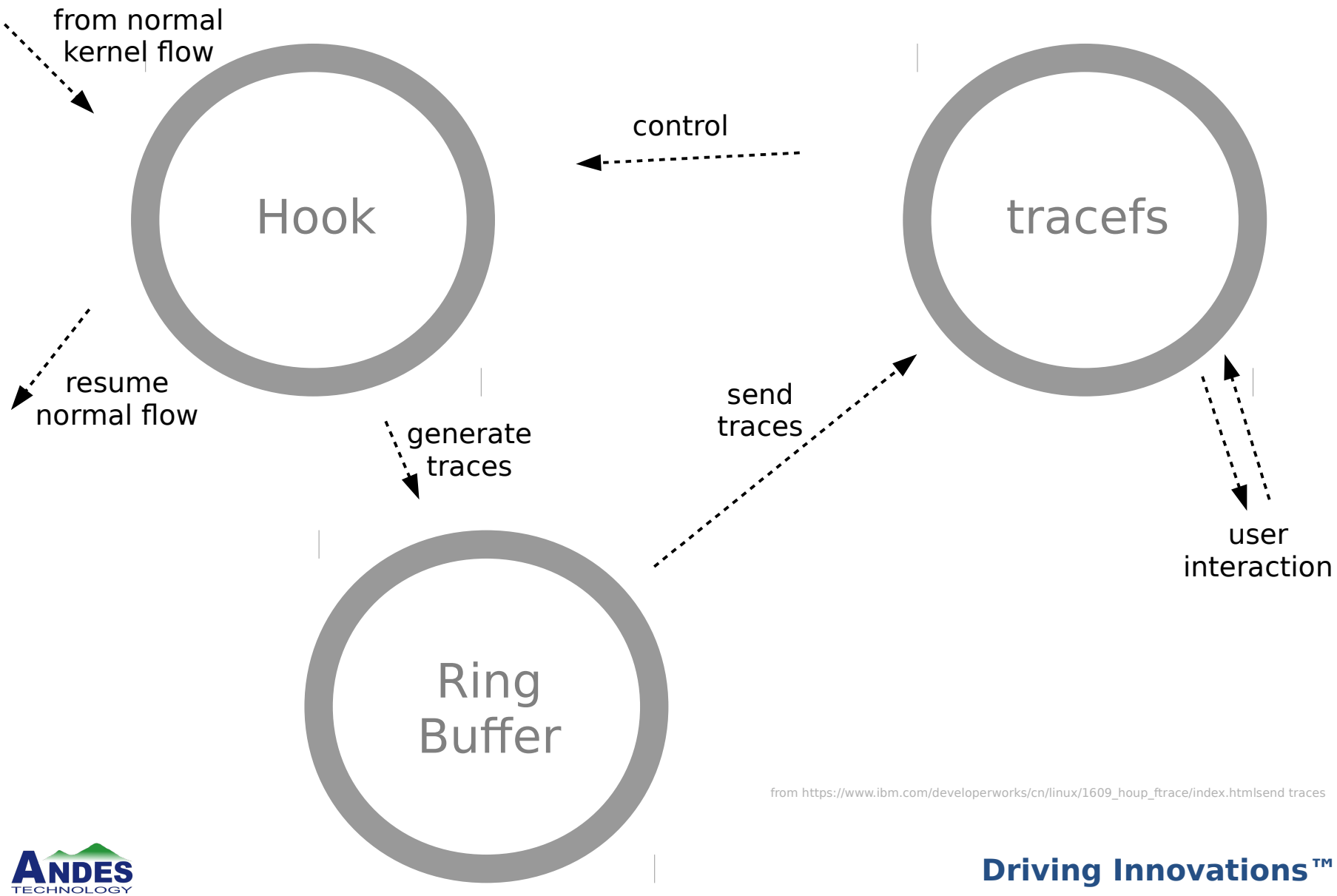
- Ftrace Overview
- In Linux Kernel
  - Generic Part
  - Arch-dependent Part
- RISC-V Port
- Future Work

Ftrace (Tracing) Overview

Introduction

- **F**unction **Tracer**
  - now serves as a framework, under the tracing subsystem
- Originally a tracer in the PREEMPT_RT patchset
  - to measure latency
  - maintained mainly by Steven Rostedt
  - into kernel (2.6.27) in 2008
- 2009: discipline
  - Documentations/trace/ftrace.txt, a tutorial on LWN
  - a paper in 11th Real-Time Linux Workshop
- **“Ftrace adds little overhead to the system when tracing.”** -- Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead, *ACM Computing Survey 2018 June*

Big Picture: Tracing



from https://www.ibm.com/developerworks/cn/linux/1609_houp_ftrace/index.htmlsend traces

Components

- Probing/Hooking Mechanism
  - relies on gcc's profiling option: `-pg`
  - inserts a hook function (`_mcount`) after each function prologue
- Ring buffer
  - A data structure supporting producer/consumer operation
- Through debugfs/tracefs
  - legacy: debugfs
  - after 2015: tracefs
  - often mounted in /sys/kernel/debug/tracing, privilege required
- Kernel config options
  - `CONFIG_FTRACE, CONFIG_DYNAMIC_FTRACE`
  - `CONFIG_FUNCTION_TRACER, CONFIG_FUNCTION_GRAPH_TRACER`
  - `CONFIG_IRQSOFF/PREEMPT/SCHED_TRACER, ...`

Basic Usage

- Example 1: start function tracer

```
# cat available_tracers
function function_graph ...

# echo function > current_tracer && \
  echo 1 > tracing_on
```

- Example 2: set white/black list to trace

```
# cat available_filter_functions
...          // some compiler-modified names here

# echo foo bar > set_ftrace_filter  // white list

# echo foo bar > set_ftrace_notrace  // black list (superior)
```

- Example 3: check the trace
  - `# cat trace` // the blocking alternative: `trace_pipe`
- Example 4: event tracing

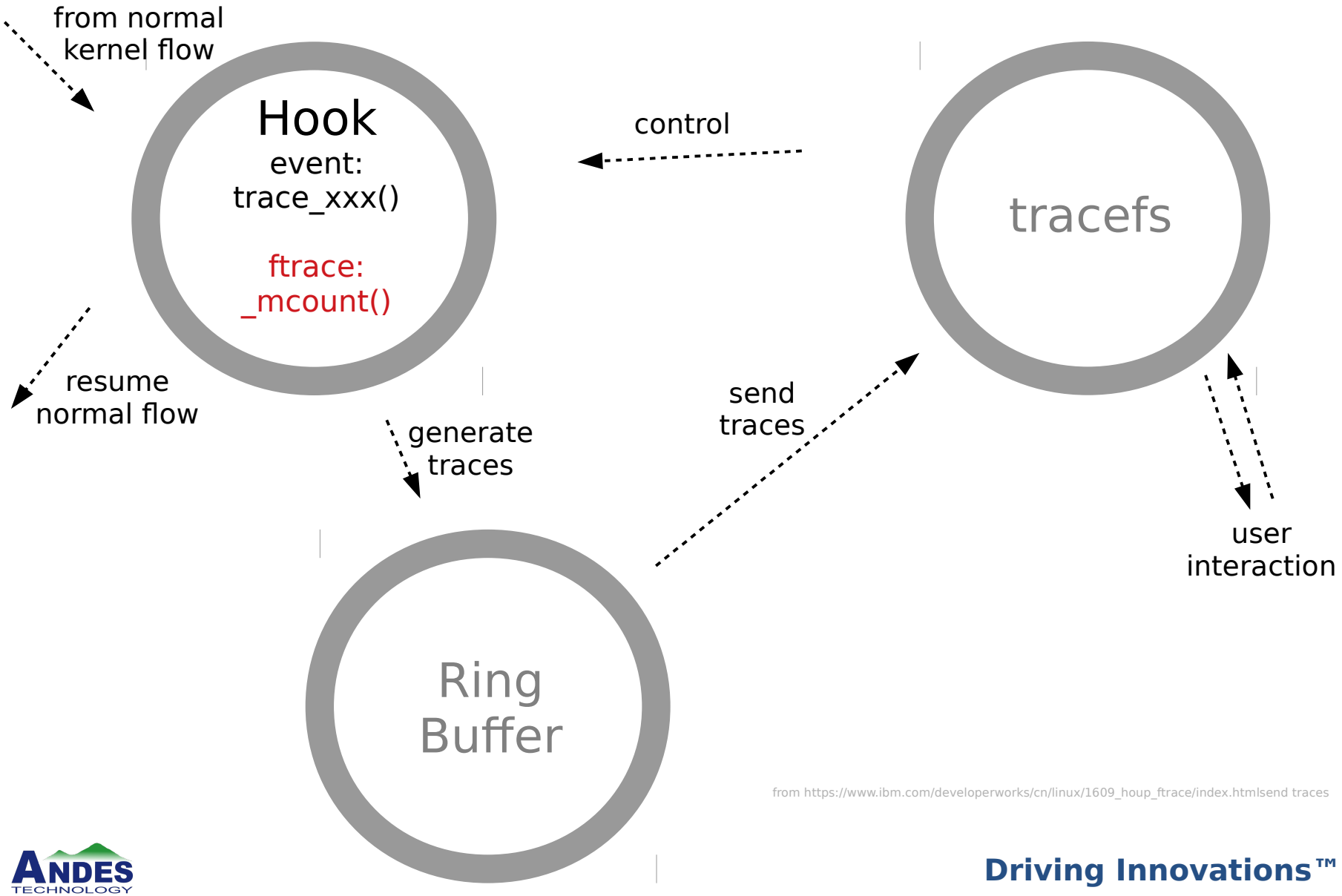
```
# cat available_events
...

# echo 1 > events/xxx/enable
```

Event tracing

- Recall the good old days: printk
  - `printk("%s\n", current->filename);`
  - quick and dirty
- Event tracing?
  - much quicker, much cleaner, and much more organized printk
  - Define an interface first, says, **`struct task_struct`**
  - Get a tracing function ready
    - e.g. trace_task_struct_here(struct task_struct* ts)
    - check macro TRACE_EVENT for details
  - Use it all around in the kernel
    - e.g. you may check `trace_rcu_utilization()` in the kernel
- The relation between **event tracing** and **ftrace**
  - both belong to tracing subsystem
  - ftrace has higher flexibility: no recompilation is needed!

Recall



from https://www.ibm.com/developerworks/cn/linux/1609_houp_ftrace/index.htmlsend traces

Ftrace Hook

- gcc generates a `_mcount()` for almost every function
  - after the prologue
- Static ftrace (seems being deprecating)
  - every function calls into `_mcount()`
  - does some checks
    - is tracing on?
    - what tracer am I using now?
  - finally, returns like nothing happened (important!)
  - but ..... **why** on earth should we suffer such overhead?
- **Dynamic ftrace**
  - If tracing is not needed for a function
    - patch the `_mcount()` to be **nop**(s)
  - assigning a tracer to a function = patching to **a call to the tracer**
  - less overhead, tracing on demand

The Generic Part of Ftrace

Case Study

- The `function_graph` tracer
  - the happy path of a normal execution

```
------------------------------------------
0)  kworker-1479  =>    sh-1475
------------------------------------------

0) + 13.500 us   |                finish_task_switch();
0) * 42171.10 us |              } /* schedule */
0) * 42230.30 us |            } /* schedule_hrtimeout_range_clock */
0) * 42304.60 us |          } /* schedule_hrtimeout_range */
0) * 42372.30 us |        } /* poll_schedule_timeout */
0)               |        __fdget() {
0) + 12.600 us   |          __fget_light();
0) ! 104.200 us  |        }
0)               |        tty_poll() {
0) + 11.900 us   |          tty_paranoia_check();
```

```
0)               |  sys_read() {
0)               |    ksys_read() {
0)               |      __fdget_pos() {
0) + 11.700 us   |        __fget_light();
0) + 70.100 us   |      }
0)               |      vfs_read() {
0) + 12.100 us   |        rw_verify_area();
0)               |        __vfs_read() {
0)               |          tty_read() {
0) + 11.400 us   |            tty_paranoia_check();
```

User input (1)

```
# cd /sys/kernel/debug/tracing
# echo function_graph > current_tracer
# echo 1 > tracing_on
```

- kernel/trace/trace.c
  - the write method to **current_tracer** is defined as

```
static const struct file_operations set_tracer_fops = {
    .write = tracing_set_trace_write,
};
```

  - tracing_set_trace_write
    - tracing_set_tracer(tr, “function_graph”);

User input (2)

```
# cd /sys/kernel/debug/tracing
# echo function_graph > current_tracer
# echo 1 > tracing_on
```

- kernel/trace/trace.c
  - the write method to **tracing_on** is defined as

```
static const struct file_operations rb_simple_fops = {
    .write = rb_simple_write,
};
```

  - rb_simple_write
    - kstrtoul_from_user to read the input number
    - tracer_tracing_on(tr);
      - ring_buffer_record_on(tr->trace_buffer.buffer);
        tr->buffer_disabled = 0;
        smp_wmb();
    - tr->current_trace->start(tr);
      - function_graph tracer doesn't have this

tracing_set_tracer (1)

```
static int
tracing_set_tracer(struct trace_array *tr, const char *buf)
{
        struct tracer *t;
        int ret = 0;

        mutex_lock(&trace_types_lock);
...
        for (t = trace_types; t; t = t->next) {
                if (strcmp(t->name, buf) == 0)
                        break;
        }
        if (!t) {
                ret = -EINVAL;
                goto out;
        }
```

struct trace_array is the top-level data structure for tracing

trace_types is a list containing all available tracers

tracing_set_tracer (2)

```
        if (t == tr->current_trace)
                goto out;
```

do nothing if the same tracer is requested

```
...

        /* If trace pipe files are being read,
           we can't change the tracer */
        if (tr->current_trace->ref) {
                ret = -EBUSY;
                goto out;
        }

        tr->current_trace->enabled--;
        if (tr->current_trace->reset)
                tr->current_trace->reset(tr);
```

a variable of struct tracer have many methods, reset() is one of them

```
        /* Current trace needs to be nop_trace
           before synchronize_sched */
        tr->current_trace = &nop_trace;
```

nop_trace is an instance of struct tracer

tracing_set_tracer (3)

```
...
        if (t->init) {
                ret = tracer_init(t, tr);
                if (ret)
                        goto out;
        }

        tr->current_trace = t;
        tr->current_trace->enabled++;
 out:
        mutex_unlock(&trace_types_lock);
        return ret;
}
```

this calls t->init() later

t = &graph_trace

tracer_init

```
void tracing_reset_online_cpus(struct trace_buffer *buf)
{
        struct ring_buffer *buffer = buf->buffer;
        int cpu;
        if (!buffer)
                return;
        ring_buffer_record_disable(buffer);

        /* Make sure all commits have finished */
        synchronize_sched();
        buf->time_start = buffer_ftrace_now(buf, buf->cpu);

        for_each_online_cpu (cpu)
                ring_buffer_reset_cpu(buffer, cpu);
        ring_buffer_record_enable(buffer);
}

int tracer_init(struct tracer *t, struct trace_array *tr)
{
        tracing_reset_online_cpus(&tr->trace_buffer);
        return t->init(tr);
}
```

record_enable/disable atomically change a flag in this buffer

reset per cpu buffer

Tracer methods

- kernel/trace/trace_functions_graph.c
  - the methods are defined as

```
static struct tracer graph_trace __tracer_data = {
        .name           = "function_graph",
        .update_thresh  = graph_trace_update_thresh,
        .open           = graph_trace_open,
        .pipe_open      = graph_trace_open,
        .close          = graph_trace_close,
        .pipe_close     = graph_trace_close,
        .init           = graph_trace_init,
        .reset          = graph_trace_reset,
        .print_line     = print_graph_function,
        .print_header   = print_graph_headers,
        .flags          = &tracer_flags,
        .set_flag       = func_graph_set_flag,
...
};
```

  - these are registered

graph_trace_init

```
void set_graph_array(struct trace_array *tr)
{
        graph_array = tr;
        smp_mb();
}
static int graph_trace_init(struct trace_array *tr)
{
    int ret;

    set_graph_array(tr);
    ret = register_ftrace_graph(&trace_graph_return,
                                &trace_graph_entry);
    if (ret)
        return ret;
    tracing_start_cmdline_record();
    return 0;
}
```

following two callbacks cannot get this tr easily, so keep it in a accessible global variable.

we have seen a context switch record, remember?

register_ftrace_graph (1)

```
int register_ftrace_graph(trace_func_graph_ret_t retfunc,
                          trace_func_graph_ent_t entryfunc)
{
        int ret = 0;
        mutex_lock(&ftrace_lock);

        /* we currently allow only one tracer
           registered at a time */
        if (ftrace_graph_active) {
                ret = -EBUSY;
                goto out;
        }

        ftrace_graph_active++;
        ret = start_graph_tracing();
        if (ret) {
                ftrace_graph_active--;
                goto out;
        }
```

fork->copy_process also checks this global variable

preparing the memory graph tracer needed, not "start tracing" at all .....

register_ftrace_graph (2)

```
…
                                          so-called register part
        ftrace_graph_return = retfunc;
        ftrace_graph_entry = entryfunc;
        ret = ftrace_startup(&graph_ops,
                             FTRACE_START_FUNC_RET);
out:
        mutex_unlock(&ftrace_lock);
        return ret;
}
```

- ftrace_startup
  - modify _mcount() hook positions
  - so that (almost) every kernel function calls **entryfunc** at entry, and calls **retfunc** at exit

  …… **but how???**

The Arch-dependent Part of Ftrace

Let's start with gcc -pg

- without -pg

```
<foo>:
00: addi sp, sp, -16
04: sd s0, 0(sp)
08: sd ra, 8(sp)
0c: addi s0, sp, 16
10: mv a0, ra
...
```

- with -pg

```
<foo>:
00: addi sp, sp, -16
04: sd s0, 0(sp)
08: sd ra, 8(sp)
0c: addi s0, sp, 16
10: call _mcount
14: mv a0, ra
...
```

Porting

- Seems that the **only** thing in arch-dep part is
  - patching codes
    - during boot: patch **`_mcount()`** **call-sites** to **nop**
    - on request: patch **nops** to **calls to some tracer**
  - In short, **not enough**
- Follow official tutorial
  - old (2009), but comprehensive
  - **`function`** and **`function_graph`** tracer
    - all tracers are based on these two prototype
  - static ftrace
  - dynamic ftrace
    - patching functions, and wrappers for them

function tracer

hook

```
<func1>:...
    call func2
    ...
    call func3
    ...
```

```
<func2>:...
    call _mcount
    call func4
    ...
    call func5
    ...
```

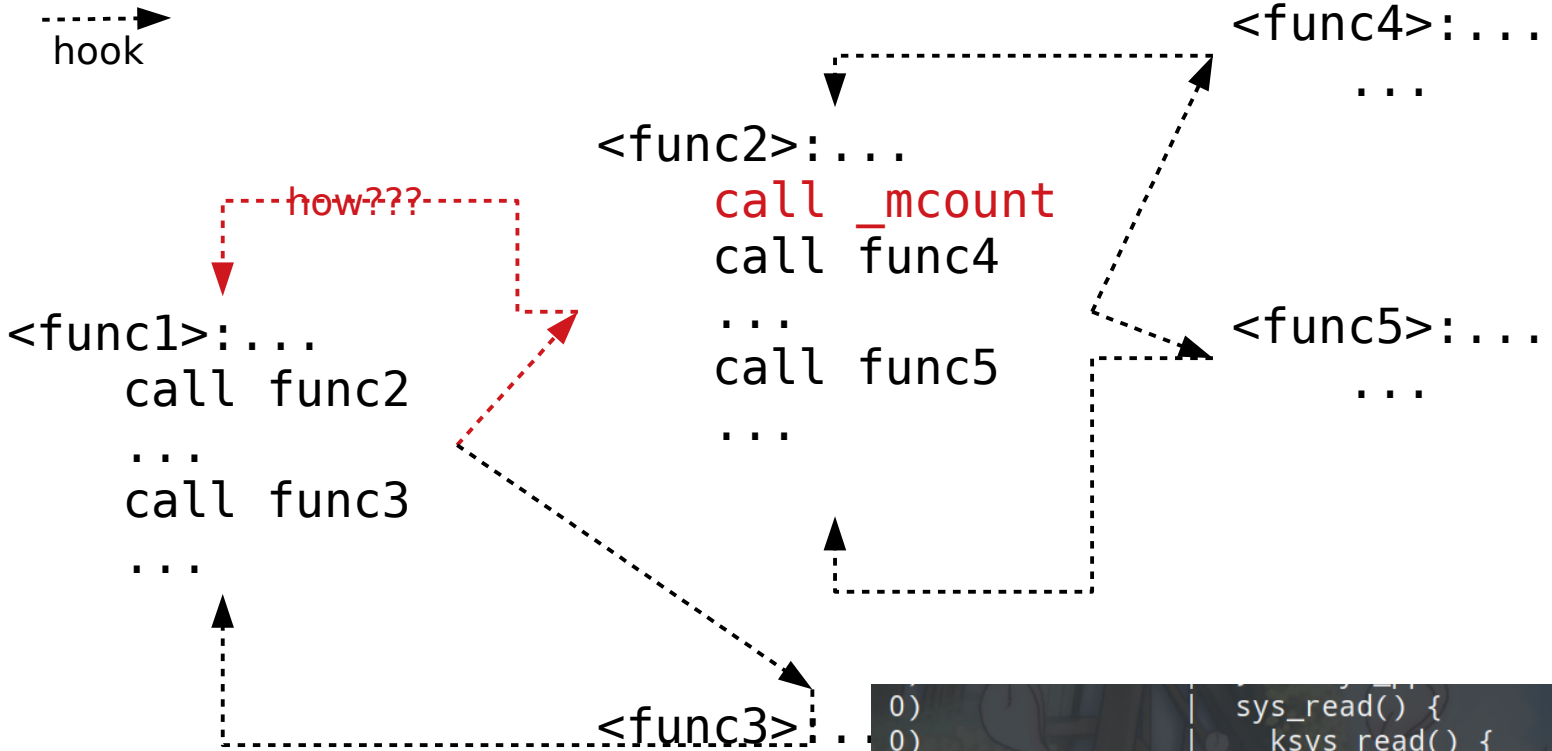
```
<func3>:...
    call func6
    ...
    call func7
```

```
<func4>:...
    call _mcount
    ...
```

```
<func5>:...
    ...
```

```
        ash-1888  [000] dn.2  6126.900000: clear_buddies <-pick_next_task_fair
        ash-1888  [000] dn.2  6126.900000: set_next_entity <-pick_next_task_fair
        ash-1888  [000] dn.2  6126.900000: __update_load_avg_se.isra.2 <-set_next_entity
        ash-1888  [000] dn.2  6126.900000: __update_load_avg_cfs_rq.isra.3 <-set_next_entity
rcu_preempt-8     [000] d..2  6126.900000: finish_task_switch <-__schedule
rcu_preempt-8     [000] d..2  6126.900000: _raw_spin_unlock_irq <-finish_task_switch
rcu_preempt-8     [000] ...2  6126.900000: preempt_count_sub <-_raw_spin_unlock_irq
rcu_preempt-8     [000] ...1  6126.900000: preempt_count_sub <-schedule
rcu_preempt-8     [000] ....  6126.900000: prepare_to_swait_event <-rcu_gp_kthread
rcu_preempt-8     [000] ....  6126.900000: prepare_to_swait <-prepare_to_swait_event
rcu_preempt-8     [000] ....  6126.900000: _raw_spin_lock_irqsave <-prepare_to_swait
rcu_preempt-8     [000] d...  6126.900000: preempt_count_add <-_raw_spin_lock_irqsave
rcu_preempt-8     [000] d..1  6126.900000: _raw_spin_unlock_irqrestore <-prepare_to_swait
```

function_graph tracer




```
0)               |  sys_read() {
0)               |    ksys_read() {
0)               |      __fdget_pos() {
0) + 11.700 us   |        __fget_light();
0) + 70.100 us   |      }
0)               |      vfs_read() {
0) + 12.100 us   |        rw_verify_area();
0)               |        __vfs_read() {
0)               |          tty_read() {
0) + 11.400 us   |            tty_paranoia_check();
```

foo calls bar, bar returns to foo

- in C-like pseudo code
- `function` tracer
  - **`ftrace_trace_function(ra_foo, ra_bar)`**
  - the information is sufficient
- `function_graph` tracer
  - **`ftrace_graph_trace_function(&ra_foo, ra_bar)`**
  - call **`ftrace_graph_entry`**
  - record `ra_foo` in a stack data structure
  - put **`ftrace_graph_return`** in `&ra_foo`
    - so that when `bar()` returns, it goes there
  - resume the execution of `bar()`
  - actual flow: `foo->bar->fg_entry->bar->fg_return->foo`

Ftrace: RISC-V port

The Design of Static Tracers

- Every function calls into `_mcount()` after its prologue
- Inside `_mcount()` (shoule be implemented in asm)
  - `function` tracer
    - condition check: is the global function pointer **`ftrace_trace_function`** equals to **`ftrace_stub`**?
    - if so, `function` tracer is not enabled
    - otherwise, provided with `ra_foo` and `ra_bar`, call it
  - `function_graph` tracer
    - condition check: is the global function pointer **`ftrace_graph_return`** equals to **`ftrace_stub`**?
    - if so, `function_graph` tracer is not enabled
    - otherwise, provided with `&ra_foo` and `ra_bar`, call a wraper **`prepare_ftrace_return`**, which can be implemented in C
      - it does everything we described previously, and more

Trouble! Cannot boot?

- in arch/riscv/kernel/head.S
  - `call setup_vm`
  - `call relocate`
- RISC-V turns MMU translation on in `relocate()`
  - recall: condition check: is the global function pointer **`ftrace_trace_function`** equals to **`ftrace_stub`**?
    - the former is a initialized function pointer to `ftrace_stub`, which is determined in compile time
    - the later is loaded in runtime
    - code snippet

```
la  t4, ftrace_stub
la  t3, ftrace_trace_function
ld  t5, 0(t3)
bne t5, t4, do_trace
```

  - the trouble comes here! in `setup_vm()`, the only C function before translation is on

Trouble! Cannot Pass FP test?

- Frame Pointer Test for `function_graph` tracer
  - ensure the stack at enter and exit are consistent
  - RISC-V frame pointer is `s0`
    - example function prologue (_mcount is called right after prologue)

```
addi sp, sp, -16
sd   s0, 0(sp)
sd   ra, 8(sp)
addi s0, sp, 16  # s0 is the stack pointer at entry
```

    - example function epilogue (ftrace_graph_return is hooked after ret)

```
ld   s0, 0(sp)
ld   ra, 8(sp)
addi sp, sp, 16
ret              # ideally, compare this sp to that s0
```

- However ..... `printk()` fails this!
  - the variable length argument ABI manipulates FP differently
  - solution: compare **the old FP**: **`-16(s0)`** at entry and **`s0`** at exit

First Patch: Static Ftrace

commit 10626c32e3827bca560217966f5bd586c4e91584
Author: Alan Kao <nonerkao@gmail.com>
Date: Mon Dec 18 17:52:48 2017 +0800

riscv/ftrace: Add basic support

This patch contains basic ftrace support for RV64I platform.
Specifically, function tracer (**HAVE_FUNCTION_TRACER**), function graph tracer (**HAVE_FUNCTION_GRAPH_TRACER**), and a frame pointer test (**HAVE_FUNCTION_GRAPH_FP_TEST**) are implemented following the instructions in Documentation/trace/ftrace-design.txt.

Note that the functions in **both ftrace.c and setup.c should not be hooked with the compiler's -pg option**: to prevent infinite self-referencing for the former, and to ignore early setup stuff for the latter.

Signed-off-by: Alan Kao <alankao@andestech.com>
Signed-off-by: Palmer Dabbelt <palmer@sifive.com>

The Design of Dynamic Ftrace

- We need the ability to patch the `_mcount()` call-sites
  - during boot: patch **`_mcount()` call-sites** to **nop**
  - on request: patch **nops** to **calls to some tracer**
- scripts/recordmcount.pl
  - as a **watever.o** is generated at compile time
    - this script is activated
    - parse the offset of all the call-sites
      - `R_RISCV_CALL _mcount`
    - put these offsets as the **`_mcount_loc`** ELF section
  - as the time of **vmlinux** final linking
    - combine all the **`_mcount_loc`** ELF section as a whole
    - in `ftrace_init()`, patch all those sites to nop
- Some APIs for these tasks

Trouble! Wrong Offsets?

- code snippet from `objdump -dr`

```
0:  1141      addi  sp,sp,-16
2:  e022      sd    s0,0(sp)
4:  e406      sd    ra,8(sp)
6:  0800      addi  s0,sp,16
8:  8506      mv    a0,ra
a:  00000097 auipc ra,0x0 # add upper imm to pc, HI20
    a: R_RISCV_CALL _mcount
    a: R_RISCV_RELAX        *ABS*
e:  000080e7 jalr  ra      # jump and link reg, LO12
```

- a general call in RISC-V is 8 bytes
  - a relaxing one is 4 bytes, which breaks the following offset
- Solution: in arch/riscv/Makefile

```
+ifeq ($(CONFIG_DYNAMIC_FTRACE),y)
+       LDFLAGS_vmlinux := --no-relax
+endif
```

Dynamic Ftrace: 3+1 APIs

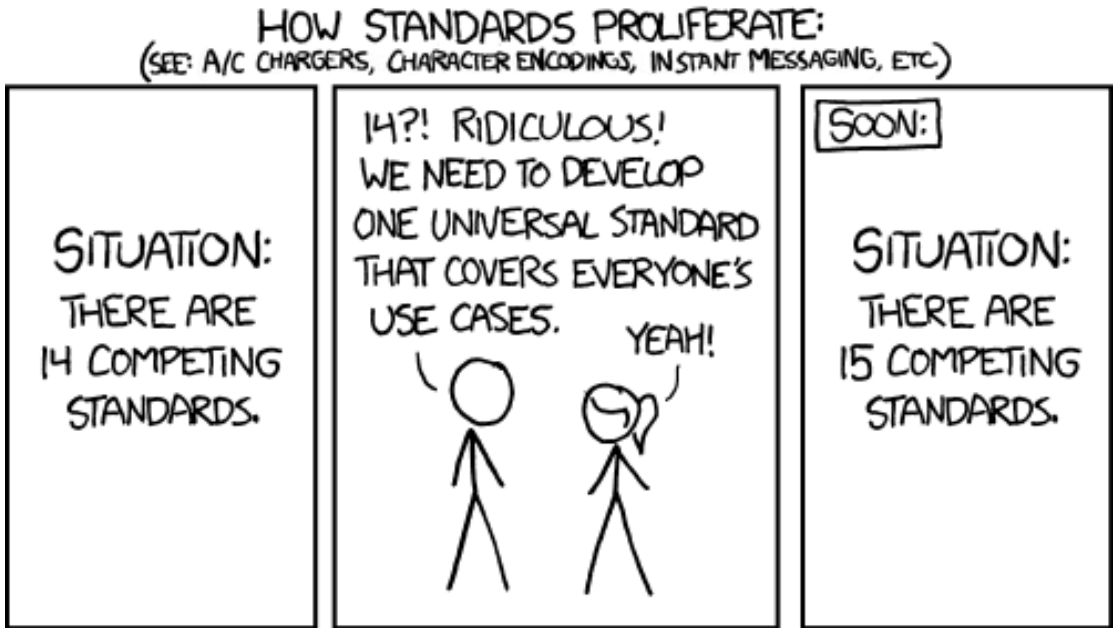
- **`ftrace_make_nop(mod, rec, addr)`**
  - **`mod`** can be kernel or some module
  - **`rec->ip`** is the `_mcount()` call site address
  - **`addr`** should be the place it originally calls to, check before patch
- **`ftrace_make_call(rec, addr)`**
  - **`addr`** is often the **`ftrace_caller`** function
- **`ftrace_caller`**, in assembly
  - a wraper for tracers
- **`ftrace_update_ftrace_function(func)`**
  - **`func`** is the new tracer (or wrapper)
  - change the tracer to **`func`** in **`ftrace_caller`**

Dynamic Graph: 2 APIs more

- **`ftrace_enable_ftrace_graph_caller(void)`**
  - the wraper **`prepare_ftrace_return`** called in **`ftrace_caller`**
- **`ftrace_disable_ftrace_graph_caller(void)`**
  - **`addr`** is often the **`ftrace_caller`** function

Not a Trouble but ...

- Consider these names:
  `ftrace_modify_code`
  `arch_ftrace_update_code`
  `ftrace_code_replace`
  `ftrace_modify_caller`
  `ftrace_modify_call`
  `ftrace_call_replace`
  `__ftrace_modify_caller`
  `__ftrace_…`
- So clear, Huh?
- so, I just made a new naming style
- my choice:
  `make_call`
  `ftrace_check_current_call`
  `__ftrace_modify_call`



ANDES TECHNOLOGY

Patch Set: Dynamic Ftrace

From: Alan Kao <alankao@andestech.com>
Date: Mon, 12 Feb 2018 19:16:41 +0800
Subject: [PATCH v4 0/6] Add dynamic ftrace support for RISC-V platforms
Cc: Greentime Hu <greentime@andestech.com>

This patch set includes the building blocks of dynamic ftrace features for RISC-V machines.

Alan Kao (6):
  riscv/ftrace: Add **RECORD_MCOUNT** support
  riscv/ftrace: Add **dynamic function tracer** support
  riscv/ftrace: Add **dynamic function graph tracer** support
  riscv/ftrace: Add ARCH_SUPPORTS_FTRACE_OPS support
  riscv/ftrace: Add DYNAMIC_FTRACE_WITH_REGS support
  riscv/ftrace: Add HAVE_FUNCTION_GRAPH_RET_ADDR_PTR support

Future Work

- Other tracers
- Linker relaxing issue
  - a new framework for __mcount: discussion
  - __fentry__ ???
- Module tracing support
- Integration with other tools
  - kprobe
  - perf

Recap

- Ftrace is a tracing framework with little overhead
- Ftrace Generic Part
  - function_graph as the example
  - from tracefs interfaces to the setting of hooks
- Ftrace Arch-dependent Part
  - design of two prototype tracers
  - dynamic hook stuff in brief
- RISC-V Port
  - Patch highlight
  - (hopefully) interesting stories

References

- Ftrace Kernel Hooks: More than just tracing by Steven Rostedt
- 知乎軟件架構設計專題 在Linux下做性能分析2：ftrace
- 完成　RISC-V 環境架設：官方版本的 busybear；稍微有點outdated的docker版本

Thank You!