Recent Tracing Updates

CELF Jamboree #78

Masami Hiramatsu <mhiramat@kernel.org>

Speaker

Masami Hiramatsu

- Works for Linaro as a Senior Tech Lead.
- Linux kernel maintainer of
  - Kprobes
  - X86 instruction decoder
  - Ftrace’s kprobe events and dynamic events
  - Perf-probe tool
  - Extra Boot Configuration
- U-Boot
  - SynQuacer platform support

Topics

- Boot-time tracing
- Eprobe
- User-space events
- Kernel object tracer
- fprobe/rethook
- Kprobe event BTF support
- Kretinsn probe

Boot time tracing

“Tracing boot time with most of ftrace features”

Boot-time tracing feature gets improved with Extra Boot Config

boottrace-LF-live-2021

Event probe

“Dynamic event on another event” (Tzvetomir Stoyanov (VMware))

Eprobe-event can define a new event on top of another static event.

- Dereference the pointers in static event arguments.
  - Record the field of the data structure.
  - Record the string from the pointer.
- Run a callback function as a hidden trigger.

Event probe usage

Record the filename string from openat syscall event

(1) Check the ‘format’ of an event

```
# # cat events/syscalls/sys_enter_openat/format
name: sys_enter_openat
…
      field:const char * filename;  offset:24;
size:8;     signed:0;
…
```

(2) Define ‘e’ probe on the event.

```
# echo "e:openat syscalls/sys_enter_openat
file=\$filename:ustring" >> dynamic_events
```

(3) Enable the eprobe event

```
# echo 1 > events/eprobes/openat/enable
```

(4) Dump the trace file

```
# cat trace
#           TASK-PID     CPU#  |||||  TIMESTAMP
FUNCTION
#              | |         |   |||||     |         |
              sh-135     [000] ...1.   130.319467:
openat: (syscalls.sys_enter_openat)
file="/etc/passwd"
```

Linaro

osnoise tracers

“Per-cpu latency statistics from OS” (Daniel Bristot de Oliveira (RedHat))

Very precise statistics

```
/sys/kernel/debug/tracing # cat trace
# tracer: osnoise
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable                              MAX
#                              |||| /     delay                                        SINGLE      Interference counters:
#                              |||||               RUNTIME      NOISE  %% OF CPU  NOISE    +-----------------------------+
#           TASK-PID      CPU# |||||   TIMESTAMP    IN US       IN US  AVAILABLE  IN US      HW    NMI    IRQ   SIRQ THREAD
#              | |         |   |||||      |           |             |      |          |      |      |      |      |      |
       osnoise/0-154     [000] .....   389.402628: 1000000      34157  96.58430    600    225      0   1000     20      3
       osnoise/2-156     [002] .....   389.404770: 1000000      61377  93.86230   7231    330      0   1014     30     24
       osnoise/1-155     [001] .....   389.409279: 1000000      26729  97.32710    295    239      0   1000     21      0
       osnoise/4-158     [004] .....   389.417794: 1000000      22383  97.76170    125    179      0   1000     29      2
       osnoise/6-160     [006] .....   389.421173: 1000000      31039  96.89610    633    423      0   1000     21      0
       osnoise/5-159     [005] .....   389.433969: 1000000      56918  94.30820   2945    363      0    986     27      0
       osnoise/7-161     [007] .....   389.436098: 1000000      25307  97.46930     89    272      0   1000     20      2
```

Linaro

timerlat tracer

Per-timer latency statistics

```
# tracer: timerlat
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| / _-=> migrate-disable
#                              |||| /     delay
#                              |||||            ACTIVATION
#            TASK-PID      CPU# |||||   TIMESTAMP   ID            CONTEXT                 LATENCY
#               | |         |   |||||      |         |                  |                       |
      timerlat/4-171     [004] .....   706.895227: #58508 context thread timer_latency    106167 ns
      timerlat/5-172     [005] .....   706.895470: #58508 context thread timer_latency    105815 ns
          <idle>-0       [004] d.h1.   706.896158: #58509 context    irq timer_latency     36692 ns
      timerlat/4-171     [004] .....   706.896227: #58509 context thread timer_latency    105876 ns
          <idle>-0       [005] d.h1.   706.896401: #58509 context    irq timer_latency     37409 ns
      timerlat/5-172     [005] .....   706.896469: #58509 context thread timer_latency    105578 ns
          <idle>-0       [004] d.h1.   706.897158: #58510 context    irq timer_latency     36712 ns
      timerlat/4-171     [004] .....   706.897226: #58510 context thread timer_latency    105802 ns
```

New features for-next

Already merged but not released yet.

- User-space event

User_events

"User application can send raw event data to ftrace" (Beau Belgrave (Microsoft))

Application can define new event and send the event to kernel.

- Event status (enabled/disabled) is exposed via mapped page.
  - Application can change the behavior of the event.
    - E.g. skip event parameter preparation when it is disabled.
  - Application event can be analyzed by ftrace histogram/filters.
- Much faster than uprobes.
  - Only one writev systemcall is needed.

User_events interfaces

2 special tracefs interfaces added for user_events

- <tracefs>/user_events_status
  - Used for sharing the event status “page” with kernel.
  - A char-array page shared by mmap().
- <tracefs>/user_events_data
  - Used for ;
    - Define a new user-event via ioctl()
    - Write user-event data from application via writev()
- Event definition ioctl(DIAG_IOCSREG)
  - Event definition passed via “struct user_reg”
  - This returns write-index and status-index.
- Event status check
  - Check the status-index byte of mmapped “user_event_status”. (!0 == enabled)
- Event data
  - Write [write-index][event-data] data via writev()

User_events usage

(1) open status file and mmap the data

(2) open data file and ioctl() the new event definition

(3) prepare the event data

(4) write the data if enabled(traced)

```
struct user_reg reg;

int page_fd = open("user_events_status", O_RDWR);
char *page_data = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_SHARED, page_fd, 0);
close(page_fd);

int data_fd = open("user_events_data", O_RDWR);

reg.size = sizeof(reg);
reg.name_args = (__u64)"test int payload";

ioctl(data_fd, DIAG_IOCSREG, &reg);

struct iovec io[2];
io[0].iov_base = &reg.write_index;
io[0].iov_len = sizeof(uint32_t);
io[1].iov_base = &payload;
io[1].iov_len = sizeof(int);

if (page_data[reg.status_index])
        writev(data_fd, io, 2);
```

User_events and trace_marker

What is the difference between user-events and trace-marker?

Trace_marker

- Pros
  - Easy to use, just write a string to <tracefs>/trace_marker.
- Cons
  - Can not disable the event.
  - Fixed trace event - only get the string from user.
  - Not able to be used with trigger and filter because the data is “string”.

User_events

- Pros
  - Can define multiple events.
  - Can disable each event.
  - Can set the trigger and filters by user-data.
- Cons
  - Need to define user_event (name and fields) before use.
  - Need to write with the event index.

New features under development

These are currently under development

- Kernel object tracer
- fprobe/rethook

Linaro

Kernel Object Tracer

“**Online object tracking tracer**” (Jeff Xie (Individual contributor))

Trace all function calls involving target object (by address).

- The target object address is specified by “objtrace” event trigger.
- The object is checked at every function entry, and recorded with the value.
  - This is a kind of function tracer filter based on the parameter.

Trigger syntax

objtrace:add:FIELD[,OFFSET][:TYPE][:COUNT][if COND]

- Currently only “add” is supported.
- “remove” will be added. (e.g. for object delete function)

Kernel object tracer usage

(1)Add an event on alloc_inode()

(2) Set “objtrace” trigger with inode and its “i_mod” field offset (0).

(3) Do something

(4) Dump trace file

```
# echo 'r alloc_inode inode=$retval' >> kprobe_events
#
```

```
# echo ‘objtrace:add:inode,0:u16:1’ >>
events/kprobes/r_alloc_inode_0/trigger
```

```
# ls > /dev/null
```

```
# cat trace
…
              ls-144     [004] ...1.   912.348433:
inode_sb_list_add <-new_inode object:0xffff8880070a89c8
value:0x0
              ls-144     [004] ...1.   912.348474:
current_time <-proc_pid_make_inode
object:0xffff8880070a89c8 value:0x416d
              ls-144     [004] ...1.   912.348480:
timestamp_truncate <-current_time object:0xffff8880070a89c8
value:0x416d
```

Linaro

fprobe/rethook

"Multiple function entry/exit probe" (Masami Hiramatsu (Linaro)/Jiri Olsa(RedHat))

Ftrace + kretprobe based new probe. (kernel API, like kprobes)

- This will speed up eBPF kprobe events for multiple functions.
  - Original Jiri's idea
- Kretprobe will be rewritten with rethook.
- Eventually, graph tracer's shadow stack will be integrated(?)

Fprobe usage

(1) Set entry/exit handler to fprobe

(2) Set optional flag

(3) Call register_fprobe*() with probe point
(fprobe supports pattern, symbol list and address list)

Do something

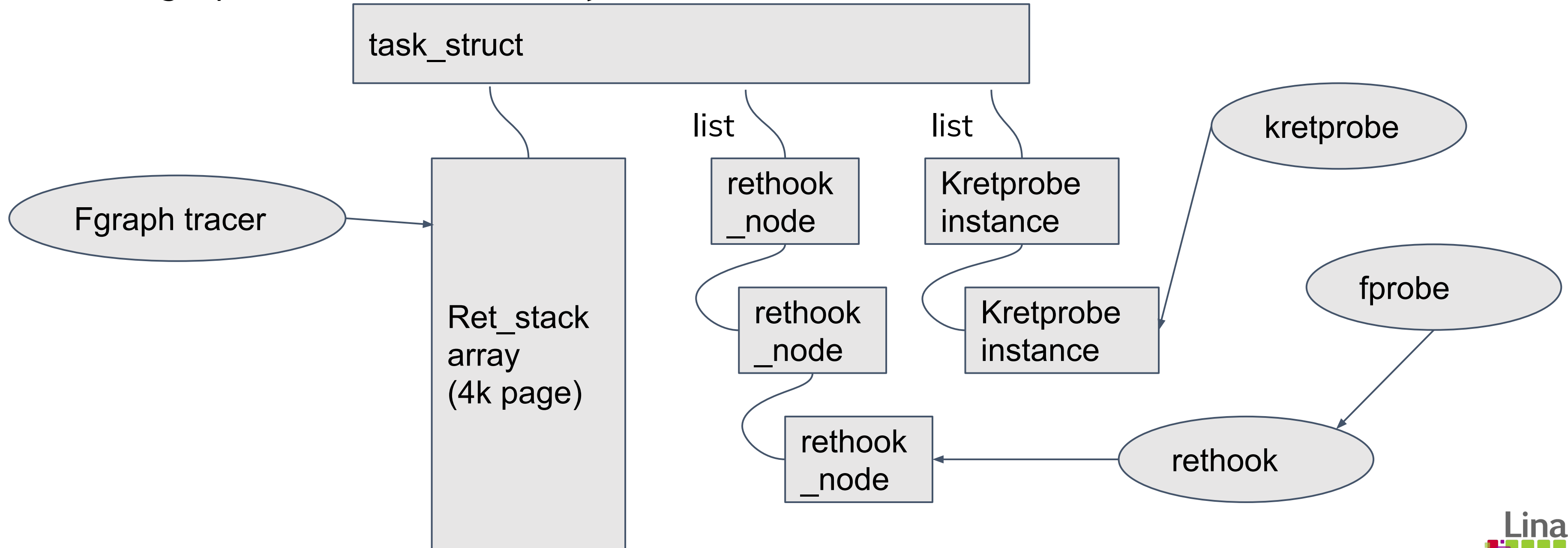
(4) Call unregister_fprobe() to finish.

```
static notrace void sample_entry_handler(struct fprobe *fp, unsigned long ip, struct pt_regs *regs)
{
        pr_info("Enter <%pS> ip = 0x%p\n", (void *)ip, (void *)ip);
}
…

fprobe.entry_handler = sample_entry_handler;
fprobe.exit_handler = sample_exit_handler;

fprobe.flags = FPROBE_FL_KPROBE_SHARED;

register_fprobe(&fprobe, “vfs_*”, “vfs_read”);

…

unregister_fprobe(&fprobe);
```

Rethook & fgraph tracer

Rethook (and its origin - kretprobe) and fgraph tracer's ret_stack are the shadow stack.

- Rethook and kretprobe make shadow stack by list of objects
- fgraph tracer uses an array for each task



Shadow Stack - Pros and Cons

List shadow stack

- Pros
  - Flexible memory usage (controllable)
- Cons
  - If the list object is not enough, fails to hook the return.

Array shadow stack

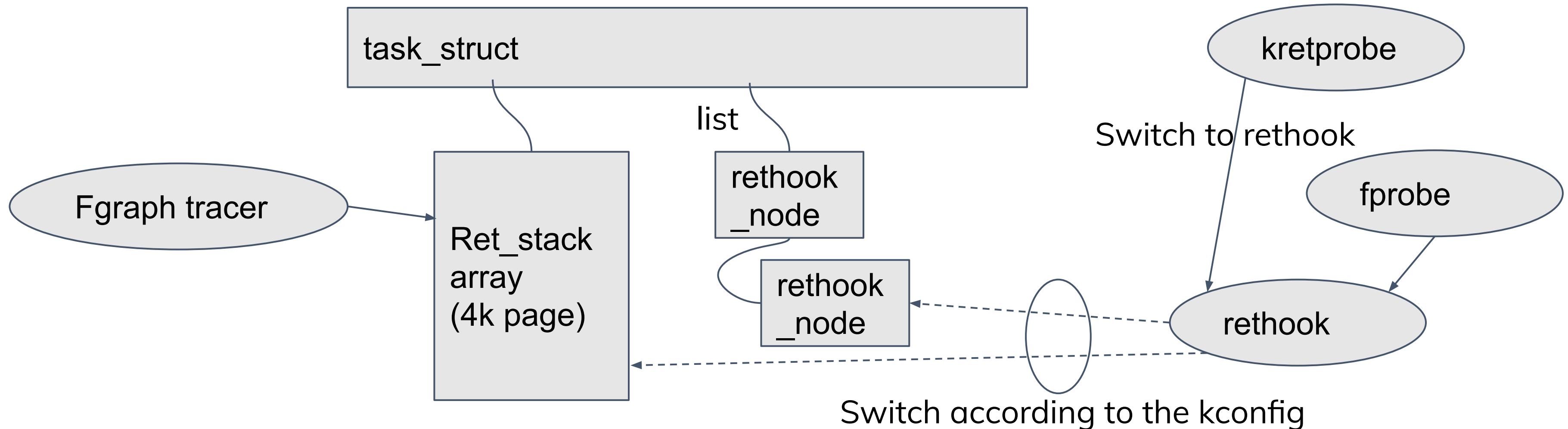
- Pros
  - Fixed memory usage (depends on number of tasks)
- Cons
  - Fixed memory usage (might be high just for a single probe)

But the worst case is enabling both of them!

TODO: Shadow Stack integration

Both of array shadow stack and list shadow stack consumes much memory.

- If the array shadow stack is enabled, use that from rethook.
- Kretprobe should move onto rethook.
- Keep using the same rethook interface but switch implementation!



PoC: Kretinsn probe

“Probe return instruction directly instead of using the shadow stack”

Another idea to solve the shadow stack limitation.

- Kretinsn probe decodes the target function and find “ret” instruction and probe it.
- This does NOT change the stack.

This will reduce memory usage of probing function return.

- Kretprobe (rethook) pools many data nodes for the shadow stack.
- fgraph tracer allocates a page for each tasks as a shadow stack.

Problem:

- It doesn’t work for the function which is tail-call to jump optimization.

Linaro

Kprobe event BTF support

“Use BTF for accessing function arguments”

Currently we need perf-probe and DEBUGINFO to access function arguments.

BTF allows kernel to analyze the name and the type of functions.

- User can define new kprobe events on function with “argument name”
- Maybe able to access data structure fields without perf-tools

BTF (BPF Type Format)

- DWARF (a.k.a. debuginfo) like binary code information.
  - Limited types are supported.
  - Only function parameter is supported.
    - DWARF supports local variables.
  - Data structure are also described.
    - E.g. the offset of each field.
  - Do not support the assignment
- BPF related tools support this feature.
  - Perf and BPF tools checks the function parameters with this.
- “__user” attribute support is under development.

Kprobe event with BTF

$$args adds all function argument with appropriate types to kprobe event.

(1) add a kprobe on a function entry with **$$args**.

(2) Then it automatically expanded to the function argument

In the future, we can specify structure fields etc. without perf-probe. (but only for the function entry)

```
# echo 'p vfs_read $$args' >> kprobe_events
```

```
# cat kprobe_events
p:kprobes/p_vfs_read_0 vfs_read file=$arg1:x64
buf=$arg2:x64 count=$arg3:u64 pos=$arg4:x64
```

```
# echo ‘p vfs_read file->f_pos buf:string’ >>
kprobe_events
# cat kprobe_events
p:kprobes/p_vfs_read_0 vfs_read
f_pos=+280($arg1):s64 buf=$arg2:ustring
```

Thank you!

Refcount leak tracking

In perf, there are many reference counters are used for managing objects.

But keeping use of refcount correctly is hard.

- Some object initialize refcount by 0, others by 1.
- Ian invented a new refcount leak tracker.
- This changes get() into alloc() and put() into free().

```
func(obj)
  _obj = get(obj);
  /* all operation must be done with _obj. */
 put(_obj);
 /* Then use-after-put can be found. */
 method(_obj) -> use after free!
```

Linaro