Google

Unifying return hooks

Simplify kernel interface

LPC23 - Tracing micro conference

Masami Hiramatsu (Google) <mhiramat@kernel.org>

Current function return hooks in the Linux kernel

We have 3 different return hooks in the Linux kernel:

- Function-graph-tracer
- Kretprobe
- Fprobe

But implementations are different.

Shadow Stack

What is the shadow stack here?

- A space for saving original return address and other info (e.g. frame pointer, time-stamp, private-data)
- As same as the real stack, each context (task) has the shadow stack **in use**.

There are 3 shadow stacks

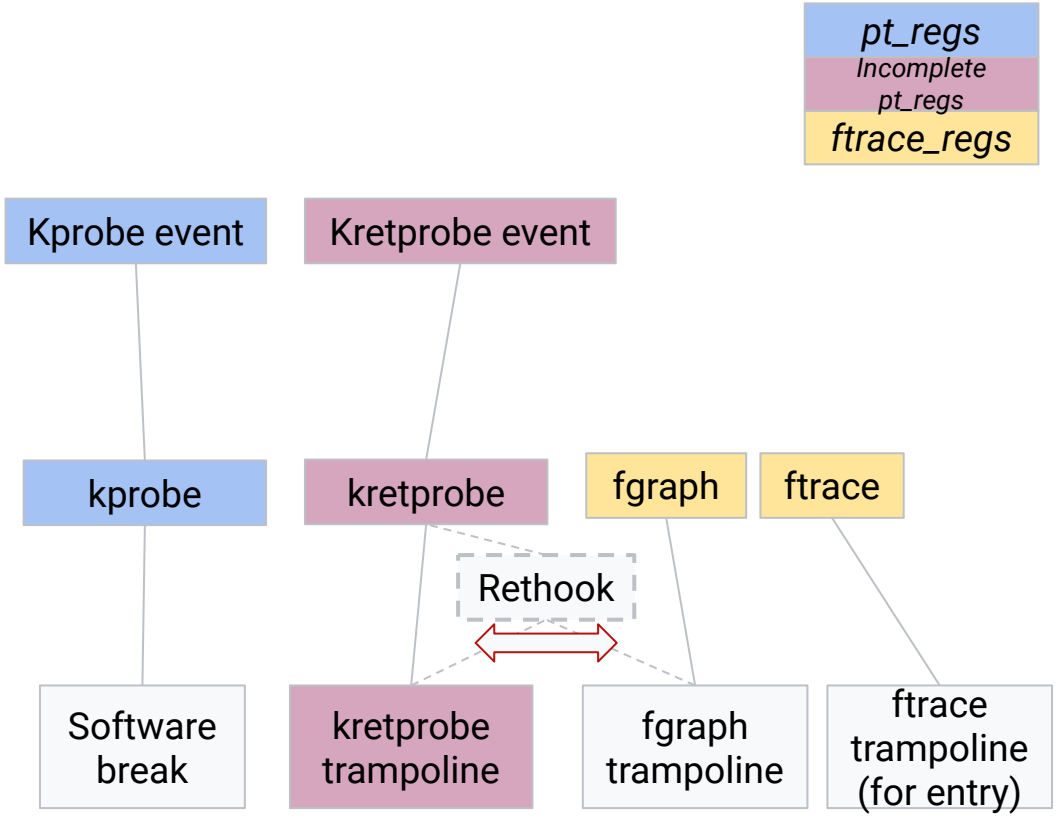
- Function_graph tracer
  - Task::ret_stack
- Fprobe (rethook)
  - Task::rethooks
- Kretprobe
  - Task::kretprobe_instances

Per-task stack

Global pool + per-task list

The old plan

Introduce the **rethook** interface to switch the kretprobe trampoline and fgraph trampoline.

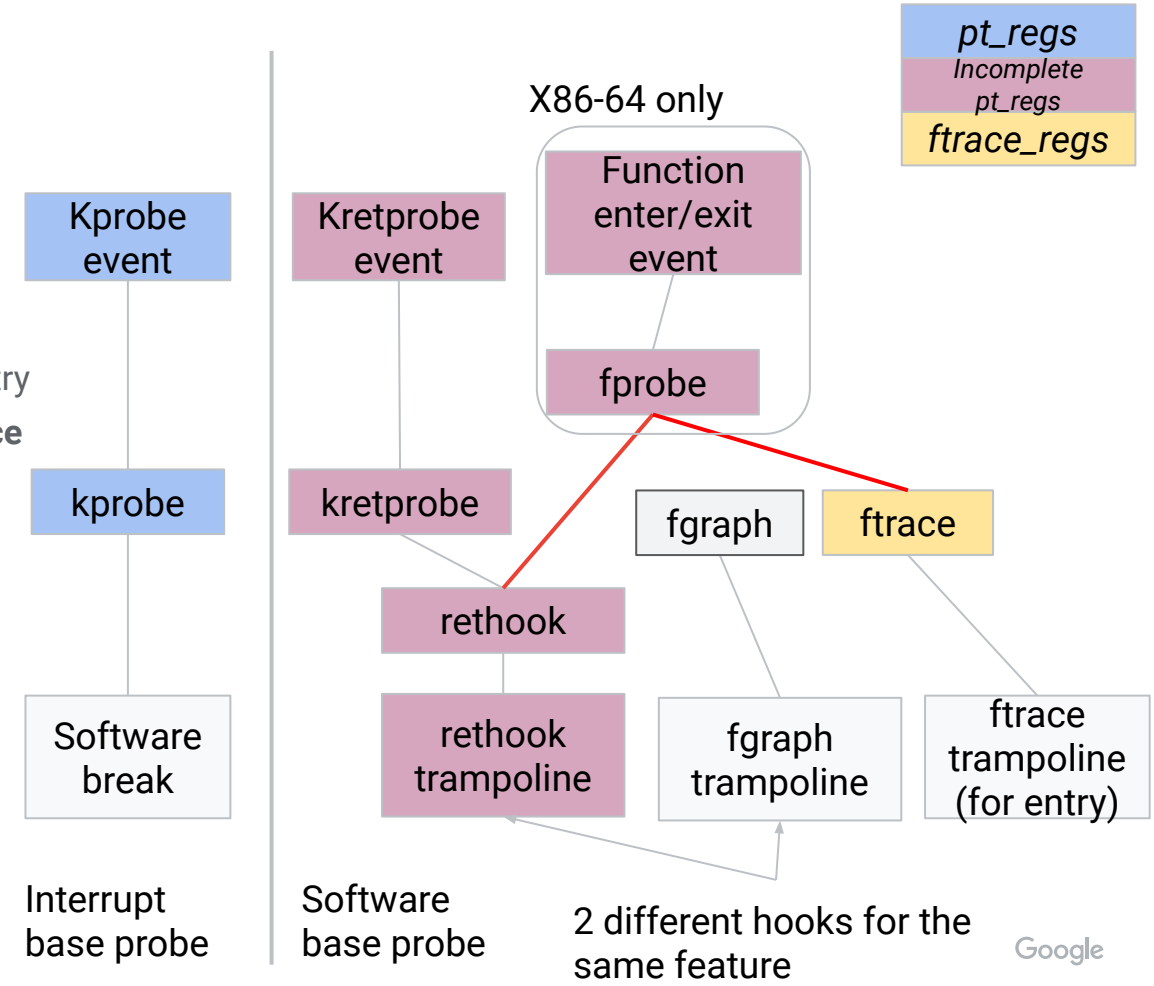


Current

Fprobe has been introduced for function entry and exit events. But fprobe is based on **ftrace** which provides **ftrace_regs**.

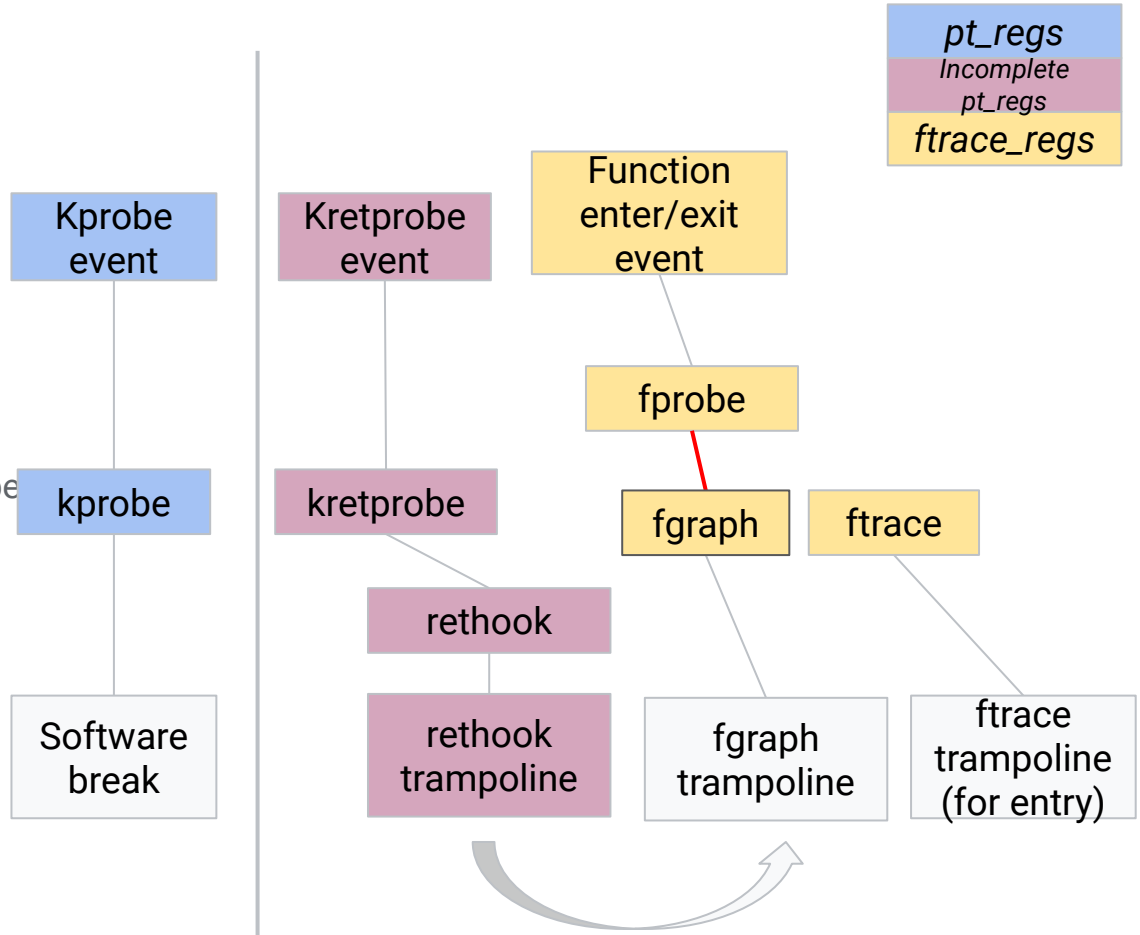
Thus, function enter/exit event uses incomplete pt_regs.

And we still have **2 different return hooks**!



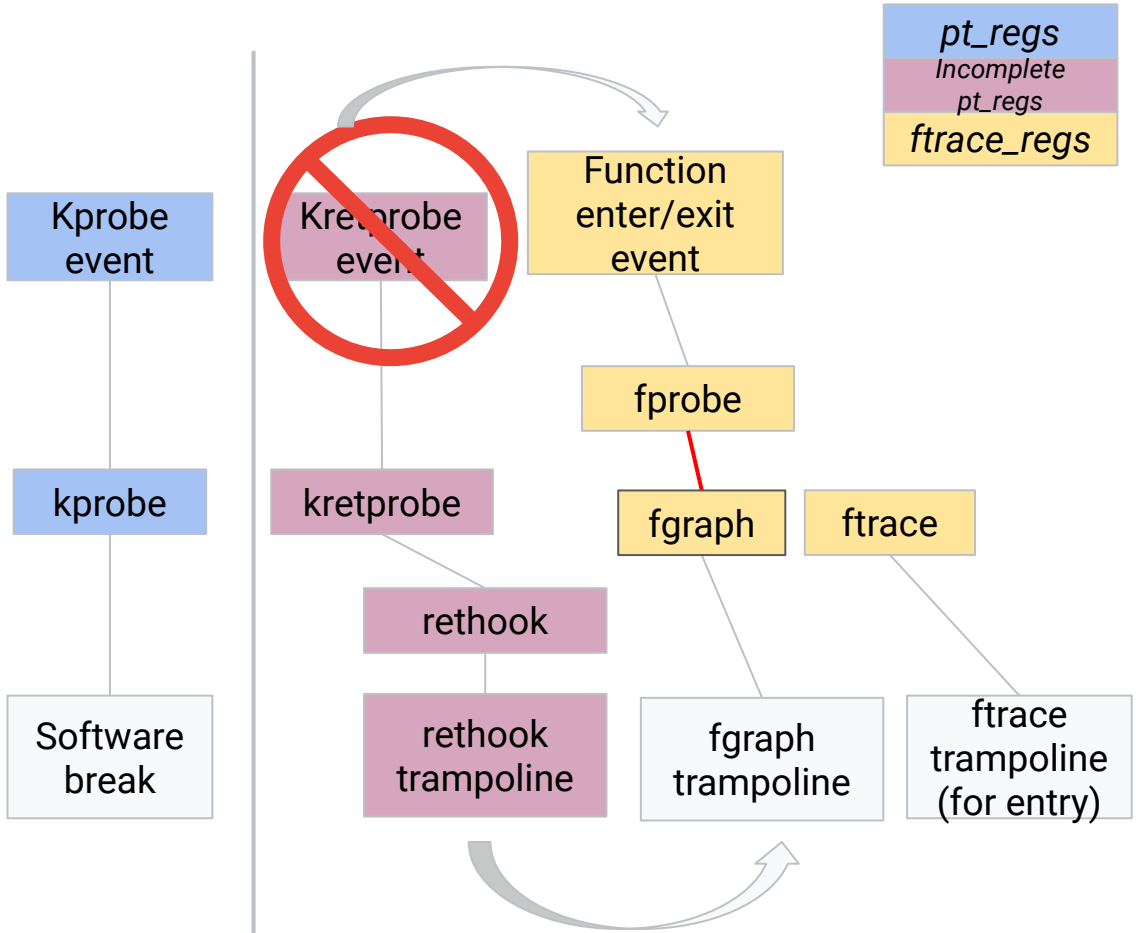
Next (ongoing) plan

(1) Make func-graph use ftrace_regs
(2) Move fprobe on the func-graph
(3) And ask all kretprobe user to use fprobe instead.



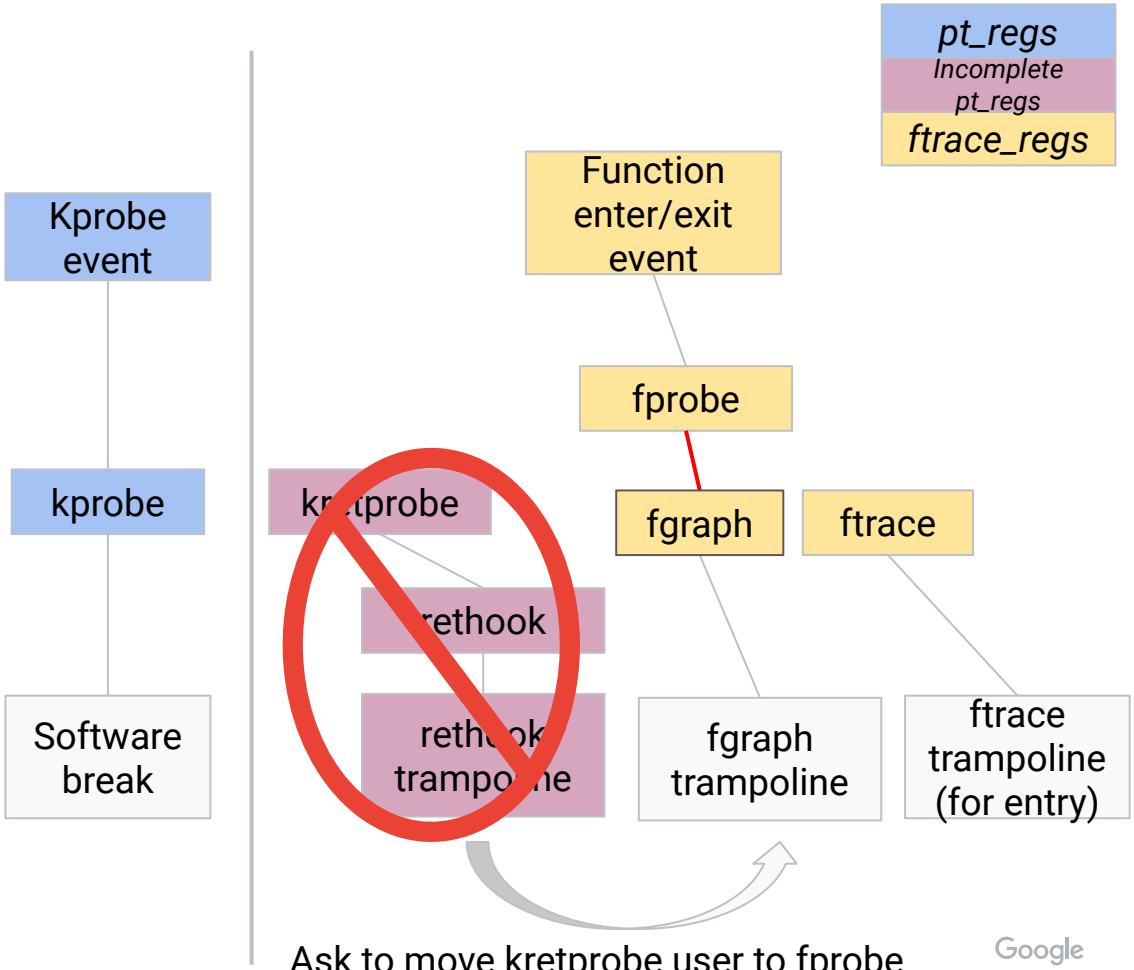
Next (ongoing) plan

Then, remove kretprobe event (compatible feature is provided by fprobe event)



Next (ongoing) plan

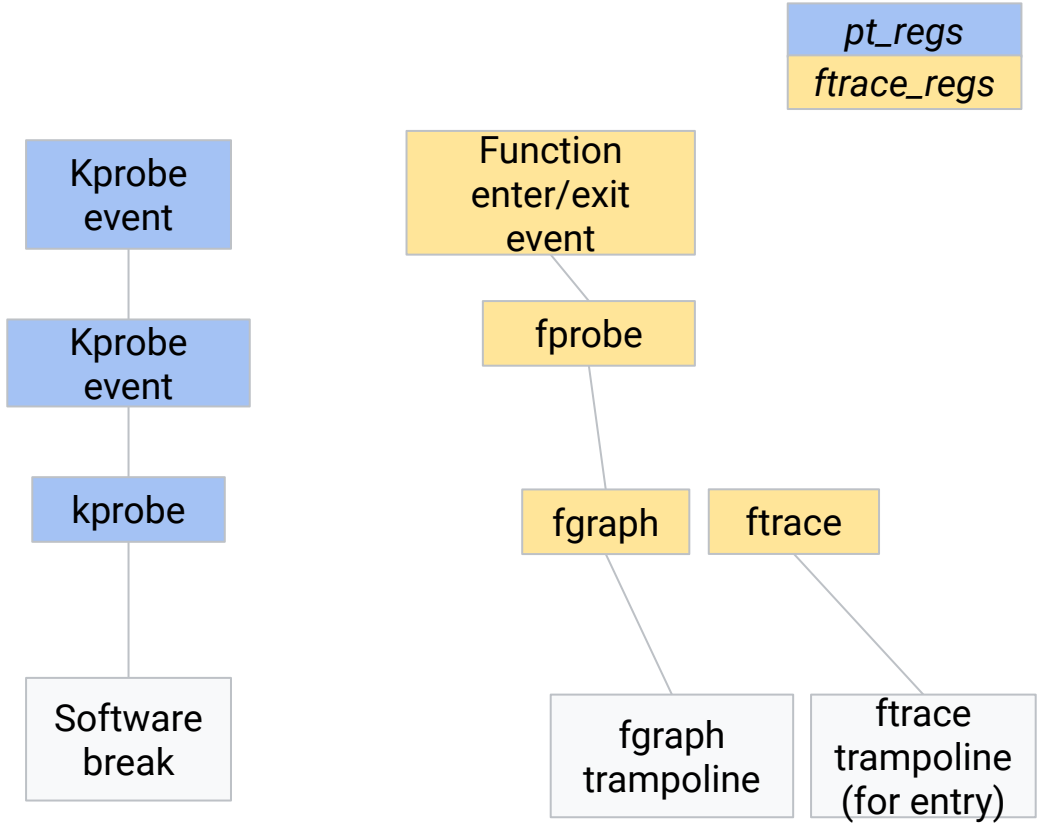
Then, we can deprecate the kretprobe return hooks.



Future proposal

Removing kretprobe makes kprobe simpler and easier to maintain.

- “Kprobe == software breakpoint”
- Only fgraph trampoline hooks function return.



Discussion

E.g.

- Do we really want to unify return hooks?
  - Keeping 2 different return hooks in kernel?
  - Performance differences?
- Is it OK to use ftrace_regs? -> next talk
- Can we remove kretprobe?
- Can’t we just share a trampoline?

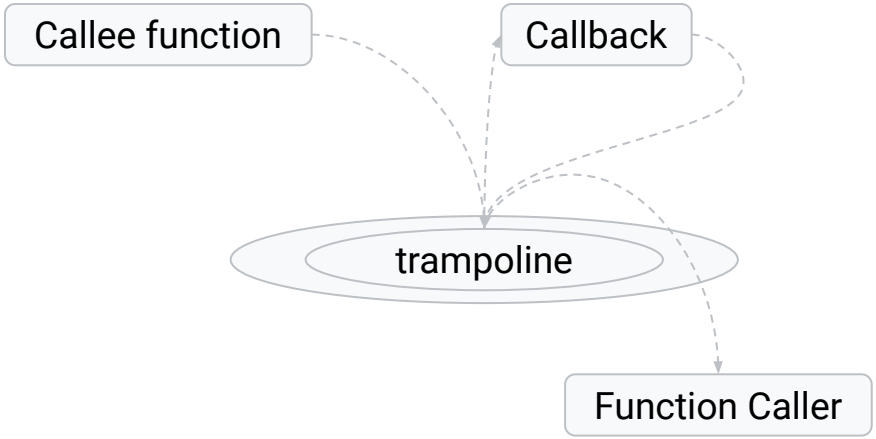
Appendix

Return Hooks

What is the return hook?

“Function return hook” hooks the function exit to call a callback.

- Callback at the function entry
  - Modifies the return address to a trampoline.
  - Save original return address to **shadow stack.**
- Callback from the trampoline
  - Use assembler code to save registers or, use a software breakpoint.
- Recover the original address
  - Restore it from **shadow stack.**



Shadow Stack

Per-task stack v.s. Global pool

Per-task stack (function-graph)

- Allocate stack page(s) for each task (thread)
- Simple array of the saved entries

Pros

- **Simple and fast**
- Scalable (in performance)

Cons

- Consume memory even if the task is not involved.

Global pool (rethook)

- Allocate fixed number of entries in system-wide pool.
- Make a linked list for each task

Pros

- Object size is controllable.
- **Usually smaller memory consumption**

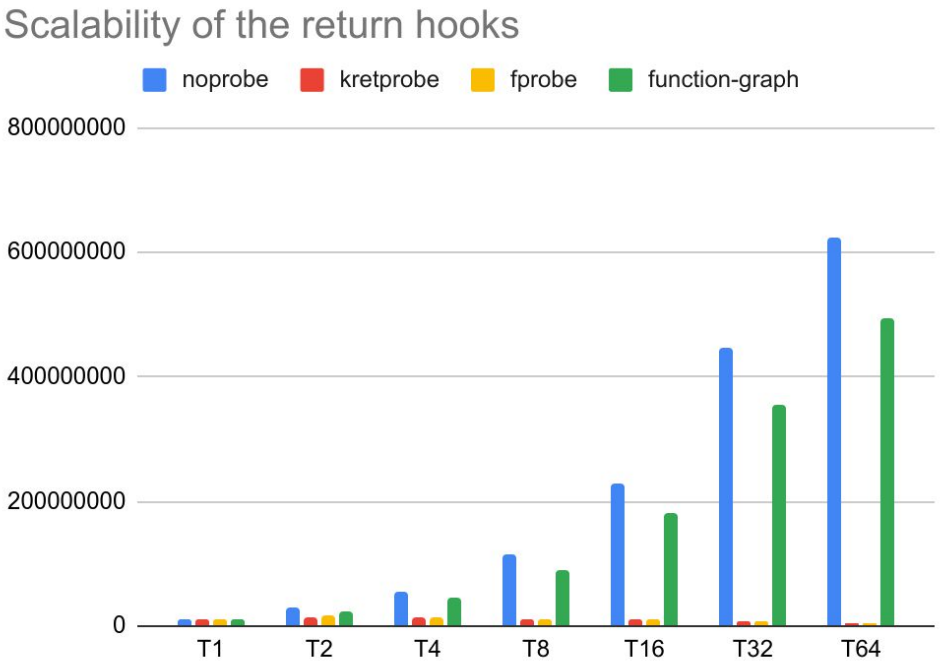
Cons

- User needs to tune the number of objects to avoid **miss-hit**
- Consuming memory if many objects selected.
- Not scalable (in performance) -> will be solved by objpool

Scalability of the shadow stacks

Current rethook is completely no scalability of the performance (overhead).

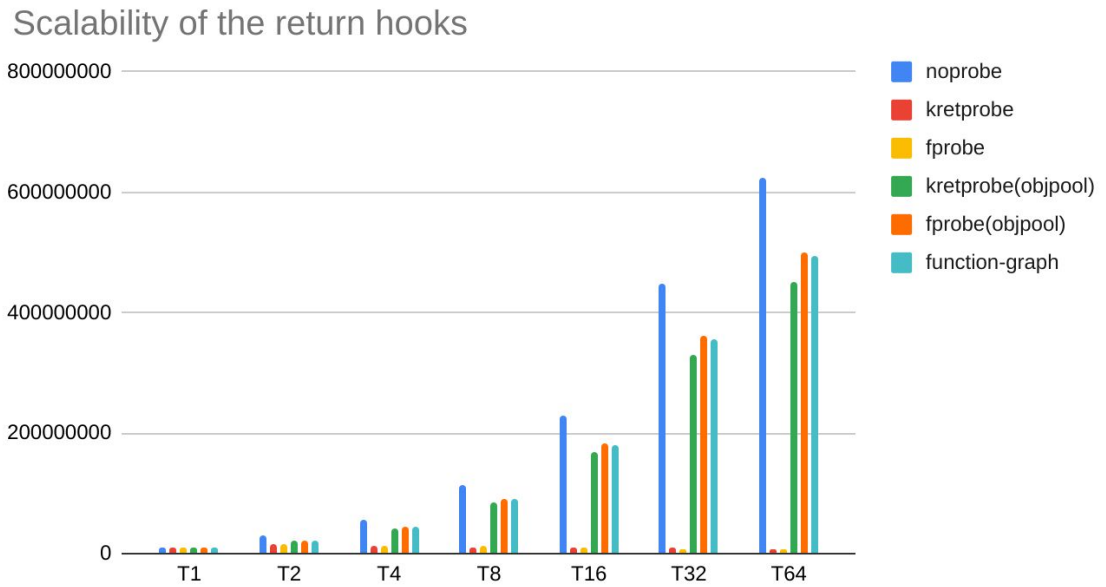
Unifying it to function-graph return hook will solve this problem.



Scalability of the shadow stacks (solved)

Objpool (from v6.7) will fix this performance issue.

So performance may not be the issue anymore.



Memory usage and tuning

Rethook: (N: # of pre-allocated nodes, a.k.a. nr_maxactive)

- N * rethook_node(=48byte)

Rethook will use less memory if it is used for a few probes, but it will be increased if

- Use many probes
- Use many pre-allocated node / probe to avoid miss-hit.
- N=# of tasks is safe number of nodes.

10 CPU, 500 tasks	1 probe (N=cpu)	100 probe (N=cpu)	1 probe (N=task)	100 probe (N=task)
Rethook	480B	48KB	24KB	**2.4MB**
Ftrace retstack	**2MB**	**2MB**	**2MB**	2MB

User has to fine tune the pre-allocated objects.
(nr_maxactive)

Comparison of the memory usage

Note that objpool will increase the memory footprint.

Rethook: (N: # of pre-allocated nodes)
- N * rethook_node(=64B)

Rethook + objpool: (N: # of pre-allocated nodes, M: # of CPUs)
- (roundup_power_of_2(N+1) * ptr) * M + N * rethook_node

10 CPU, 500 tasks	1 probe (N=cpu)	100 probe (N=cpu)	1 probe (N=task)	100 probe (N=task)
Rethook	480B	48KB	24KB	2.4MB
Rethook + objpool	1.6KB	160KB	65KB	**6.5MB**
Ftrace retstack	**2MB**	**2MB**	**2MB**	2MB

Callback arguments issue

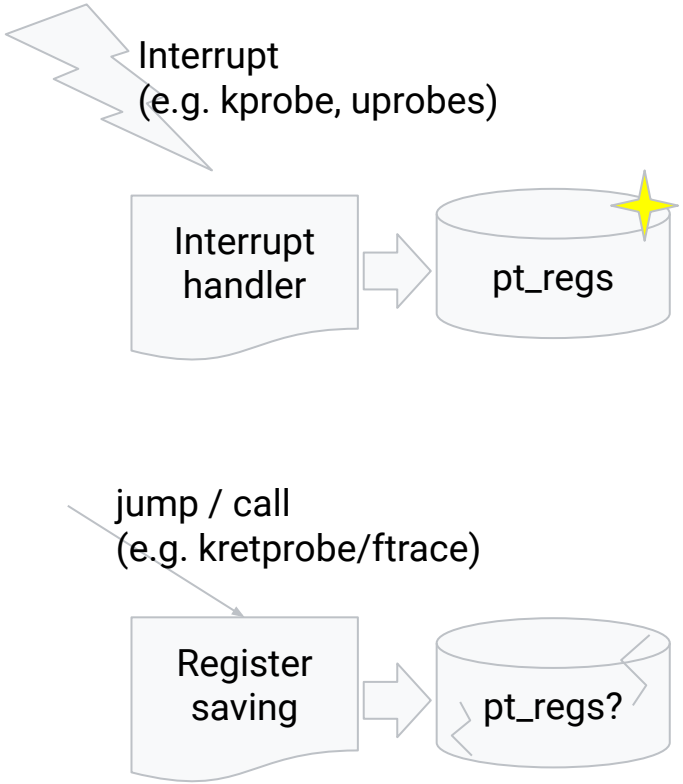
Problem of using pt_regs

pt_regs is designed for storing all registers in the interrupt context (some registers are saved automatically)

- Some registers can not be saved manually (e.g. pstate @arm64)
- Most of the registers are not used but take time to save it.

So unless it is saved by an interrupt, **pt_regs is not correct** and **takes more overhead**.

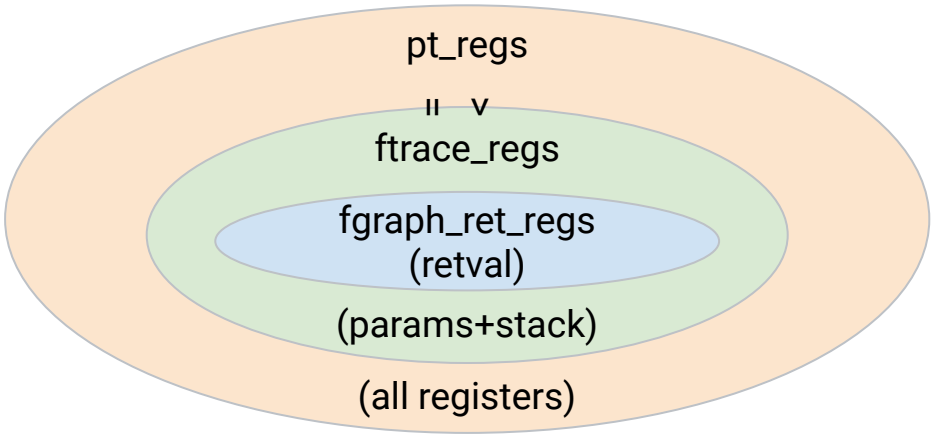
This is the reason why **arm64 doesn't support kprobes on ftrace**. (and it should not support kretprobe too)



Ftrace_regs is a handy option

Ftrace_regs is a partial set of pt_regs (most architectures just wraps pt_regs).

fgraph_ret_regs is a shrunken version of ftrace_regs, but it only has return value.



What is the ftrace_regs?

ftrace_regs only saves the registers for;

- Function **parameters**
- Function **return values**
- Hooking/unwinding **function call** (e.g. frame pointer, link register or stack pointer and instruction pointer)
- (optional) arch implementation dependent

Don’t include state flags, callee-save registers etc.

```
int function_foo(int param1, long param2, void *param3)
{
        ...

        return ret;
}
```

```
[    2.794307]  function_graph_enter_regs+0x184/0x280
[    2.796119]  ? fprobe_selftest_target+0x4/0x20
[    2.797809]  ? test_fprobe_entry+0x91/0x300
[    2.799409]  ? fprobe_selftest_target+0x4/0x20
[    2.801105]  ftrace_graph_func+0xcd/0x170
....
```