

# Linux Plumbers Conference

Vienna, Austria | September 18-20, 2024



# Atomic code patching and ftrace

Puranjay Mohan and Tao Chiu

LINUX PLUMBERS CONFERENCE

Vienna, Austria  
Sept. 18-20, 2024

# The Ftrace mechanism for dummies

# <vfs\_open>:

addi sp, sp, -0x10

sd s0, 0x0(sp)

sd ra, 0x8(sp)

addi s0, sp, 0x10

Id a4, 0x0(a0)

mv a5, a0

mv

[...]

[...]

[...]

[...]

[...]

[...]

[...]

Aim: To call my\_tracer for all calls to vfs\_open.

# <my\_tracer>:

mv a5, a0

mv a0, a1

[ ... ]

[ ... ]

[ ... ]

[ ... ]

[ ... ]

[...]

[...]

[...]

[...]

[...]

[...]

# The Ftrace mechanism for dummies

```
<vfs_open>:  
nop  
nop  
addi    sp, sp, -0x10  
sd      s0, 0x0(sp)  
sd      ra, 0x8(sp)  
addi    s0, sp, 0x10  
ld      a4, 0x0(a0)  
mv      a5, a0  
mv      a0, a1  
[...]  
[...]  
[...]  
[...]  
[...]
```

# The Ftrace mechanism for dummies

```
<vfs_open>:  
nop  
nop  
addi sp, sp, -0x10  
sd s0, 0x0(sp)  
sd ra, 0x8(sp)  
addi s0, sp, 0x10  
ld a4, 0x0(a0)  
mv a5, a0  
mv a0, a1  
[...]  
[...]  
[...]  
[...]  
[...]
```

```
<ftrace_caller>:  
save_regs  
load_regs  
<ftrace_call>:  
call ftrace_stub  
restore_regs  
jr t0  
<ftrace_stub>:  
jr t0
```

A diagram illustrating the control flow between the ftrace\_caller and ftrace\_stub functions. It shows three main components: ftrace\_caller, ftrace\_call, and ftrace\_stub. The ftrace\_caller contains save\_regs, load\_regs, and a call to ftrace\_stub. The ftrace\_caller also contains a restore\_regs instruction and a jr t0 instruction that loops back to the save\_regs instruction. The ftrace\_call contains a call to ftrace\_stub. The ftrace\_stub contains a jr t0 instruction that loops back to the call in the ftrace\_call.

# The Ftrace mechanism for dummies

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi    sp, sp, -0x10  
    sd      s0, 0x0(sp)  
    sd      ra, 0x8(sp)  
    addi    s0, sp, 0x10  
    ld      a4, 0x0(a0)  
    mv      a5, a0  
    mv      a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

The diagram illustrates the control flow between two assembly code snippets. A curved arrow originates from the 'ftrace\_caller' label in the first snippet and points to the start of the second snippet. Another curved arrow originates from the 'jr t0' instruction in the second snippet and points back to the 'ftrace\_caller' label in the first snippet, forming a loop.

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call   ftrace_stub  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

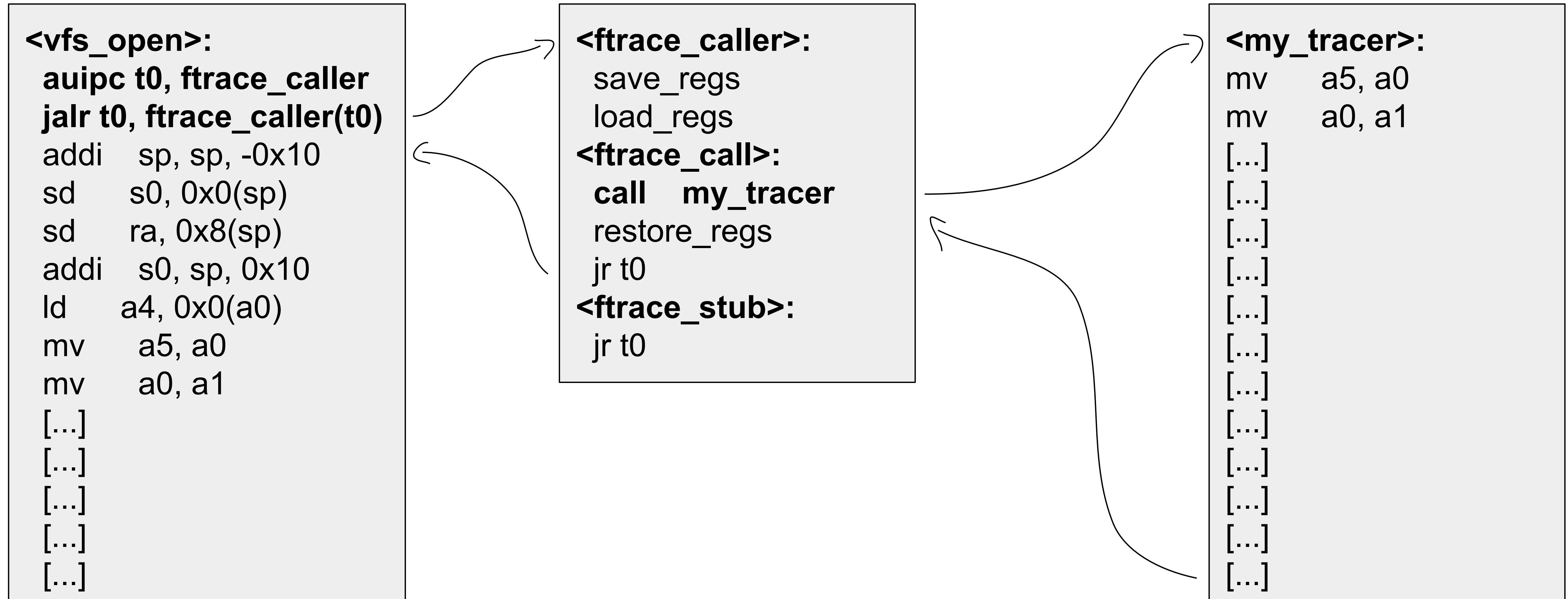
# The Ftrace mechanism for dummies

```
<vfs_open>:  
auipc t0, ftrace_caller  
jalr t0, ftrace_caller(t0)  
addi    sp, sp, -0x10  
sd      s0, 0x0(sp)  
sd      ra, 0x8(sp)  
addi    s0, sp, 0x10  
ld      a4, 0x0(a0)  
mv      a5, a0  
mv      a0, a1  
[...]  
[...]  
[...]  
[...]  
[...]
```

The diagram illustrates the control flow between three kernel functions: **ftrace\_caller**, **ftrace\_call**, and **ftrace\_stub**. The flow starts at the top left, enters the **ftrace\_caller** box, which contains the instructions `save_regs` and `load_regs`. An arrow points from the right side of the **ftrace\_caller** box to the left side of the **ftrace\_call** box. Inside the **ftrace\_call** box, the sequence `call ftrace_stub`, `restore_regs`, and `jr t0` is shown. From the right side of the **ftrace\_call** box, two arrows point to the right side of the **ftrace\_stub** box. Inside the **ftrace\_stub** box, the instruction `jr t0` is shown again, with an arrow pointing back to the right side of the **ftrace\_call** box.

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call    ftrace_stub  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

# The Ftrace mechanism for dummies



# The Ftrace mechanism for dummies

```
<vfs_open>:  
nop  
nop  
addi sp, sp, -0x10  
sd s0, 0x0(sp)  
sd ra, 0x8(sp)  
addi s0, sp, 0x10  
ld a4, 0x0(a0)  
mv a5, a0  
mv a0, a1  
[...]  
[...]  
[...]  
[...]  
[...]
```

```
<ftrace_caller>:  
save_regs  
load_regs  
<ftrace_call>:  
call ftrace_stub  
restore_regs  
jr t0  
<ftrace_stub>:  
jr t0
```

# The Ftrace mechanism for dummies

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi    sp, sp, -0x10  
    sd      s0, 0x0(sp)  
    sd      ra, 0x8(sp)  
    addi    s0, sp, 0x10  
    ld      a4, 0x0(a0)  
    mv      a5, a0  
    mv      a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

The diagram illustrates the control flow between two code snippets. On the left, the `<vfs_open>` function is shown. At the end of this function, there is a call to the `ftrace_caller` stub. This call is annotated with a curved arrow pointing to the `ftrace_caller` stub on the right. The `ftrace_caller` stub itself contains several steps: saving registers, loading registers, performing a `ftrace_call` (which calls the `ftrace_stub`), restoring registers, and finally jumping back to the original `t0` instruction. A second curved arrow points from the `ftrace_stub` back to the `ftrace_caller`, indicating the return path.

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call    ftrace_stub  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

# The Ftrace mechanism for dummies

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

The diagram illustrates the control flow between four code regions:

- <vfs\_open>:** This region contains assembly instructions for opening a file system. It ends with a jump to **ftrace\_caller**.
- <ftrace\_caller>:** This region contains assembly instructions for saving and loading registers, followed by a jump to **ftrace\_call**.
- <ftrace\_call>:** This region contains assembly instructions for calling the **ftrace\_stub**, restoring registers, and jumping back to the original **t0** address.
- <ftrace\_stub>:** This region contains a jump back to the original **t0** address.

Curved arrows indicate the flow of control from **vfs\_open** to **ftrace\_caller**, from **ftrace\_caller** to **ftrace\_call**, and from **ftrace\_call** to **ftrace\_stub**.

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_stub  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
<my_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<my_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

# The Ftrace mechanism for dummies

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

The diagram illustrates the control flow between four code sections. An arrow points from the `auipc` instruction in `<vfs_open>` to the `ftrace_caller` section. Inside `<ftrace_caller>`, arrows point from `save_regs` to `load_regs`, and from `load_regs` to the `ftrace_call` section. Within `<ftrace_call>`, arrows point from `call ftrace_stub` to `restore_regs`, and from `restore_regs` to `jr t0`. Finally, an arrow points from `jr t0` back to the `ftrace_stub` section. Inside `<ftrace_stub>`, an arrow points from `jr t0` back to the `ftrace_caller` section.

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_stub  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
<my_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<my_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<ftrace_ops_list_func>:  
/* Iterate, check, and call */
```

# The Ftrace mechanism for dummies

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_ops_list_func  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

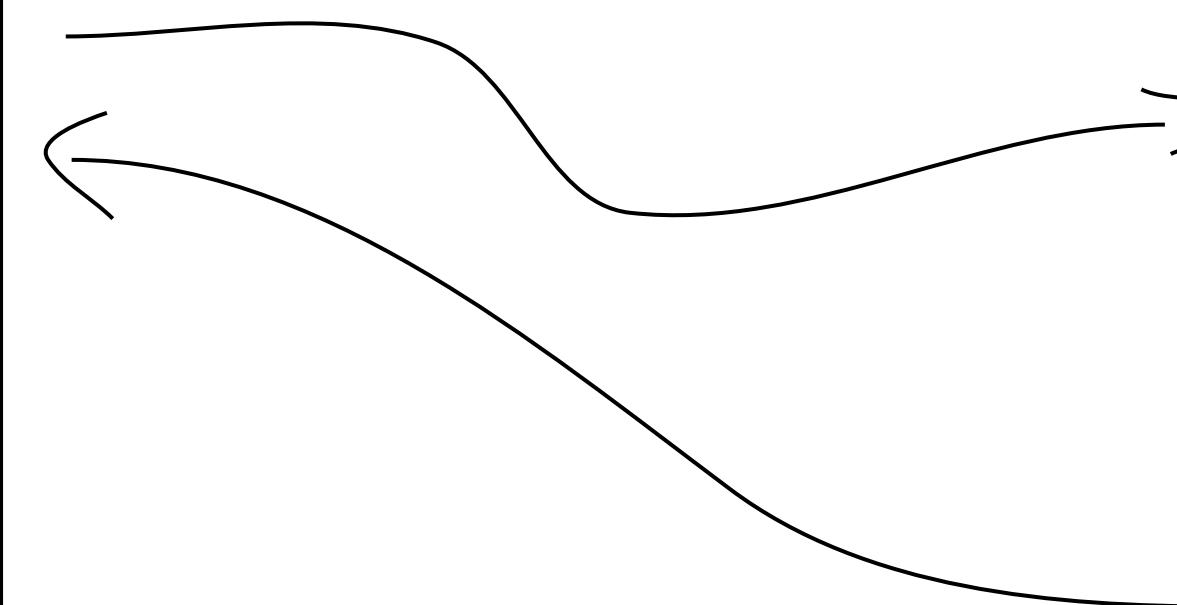
```
<ftrace_ops_list_func>:  
/* Iterate check and call */
```

```
<my_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<my_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

# The Ftrace mechanism for dummies: Direct calls

```
<vfs_open>:  
    auipc t0, my_tracer_dcc  
    jalr t0, my_tracer_dcc(t0)  
    addi   sp, sp, -0x10  
    sd     s0, 0x0(sp)  
    sd     ra, 0x8(sp)  
    addi   s0, sp, 0x10  
    ld     a4, 0x0(a0)  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```



```
<my_tracer_dcc>:  
    save_regs  
    mv   a5, a0  
    mv   a0, a1  
    [...]  
    restore_regs
```

# RISC-V: Drawbacks of the current implementation

- 2 instructions (auipc, jalr) are to be patched at runtime.
  - stop\_machine() on all cpus except one that does the patching.
- Can't work with kernel preemption as kernel preemption allows process to be scheduled out while executing on one of these instruction pairs.
  - Ftrace + PREEMPT not supported.
- Looking at this from a lower level:
  - Function entries have an auipc+jalr pair [To be atomically patched]
  - Ftrace\_caller trampoline has an auipc+jalr pair [To be atomically patched]

# Proposed Solution

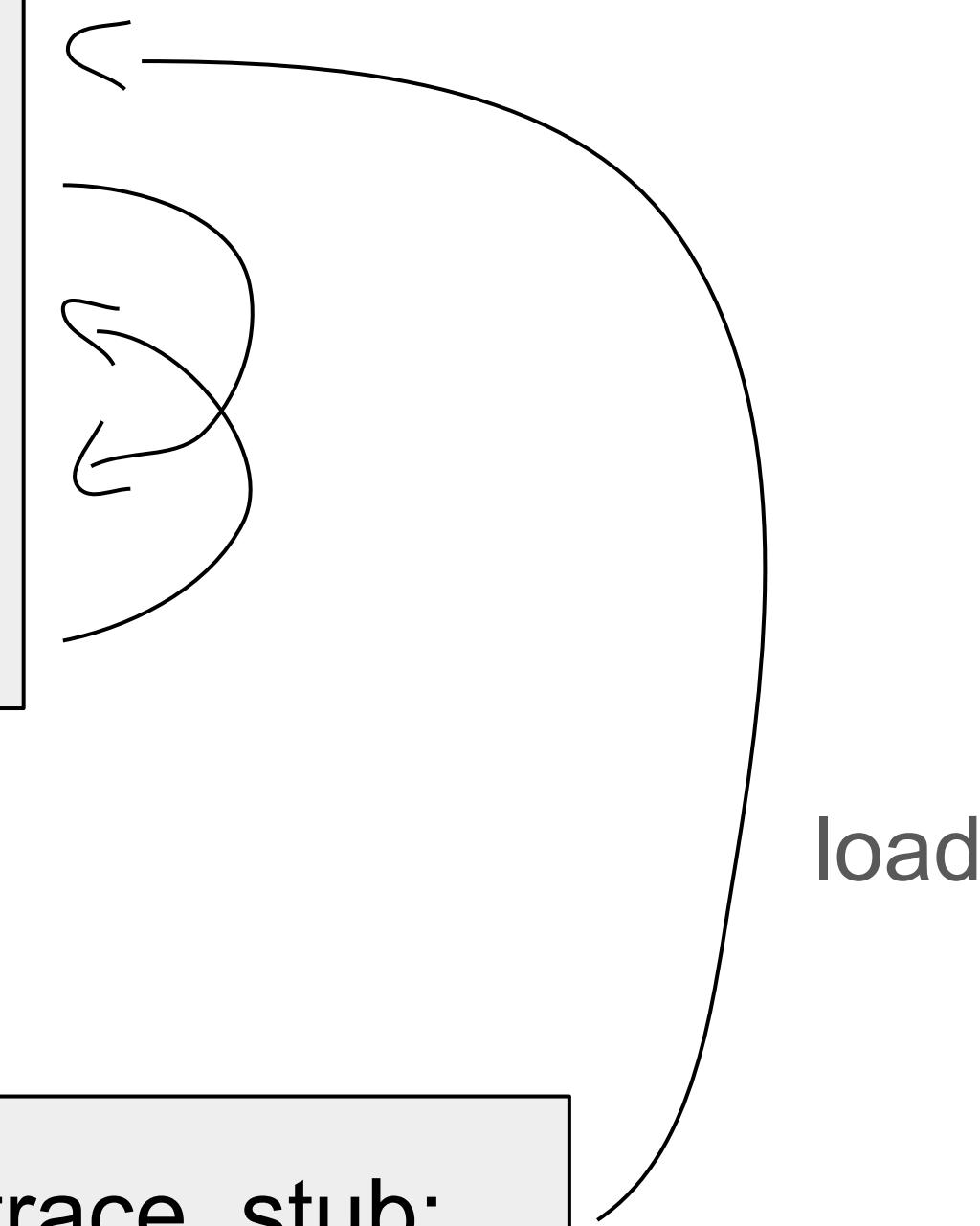
- We know that we are calling/not–calling `ftrace_caller` from function entry.  
So, we can leave the auipc and only change `nop`  $\longleftrightarrow$  `jalr`
  - This assumes we don't support direct calls. [More on that later]
- Modifying a single instruction can be done atomically if it is aligned.

# Proposed Solution Implementation

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    nop  
    addi   sp, sp, -0x10  
    sd     s0, 0x0(sp)  
    sd     ra, 0x8(sp)  
    addi   s0, sp, 0x10  
    ld     a4, 0x0(a0)  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    REG_Lra, ftrace_call_dest  
    jalr0(ra)  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
ftrace_func_t ftrace_call_dest = ftrace_stub;
```



# Proposed Solution Implementation

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    nop  
    addi   sp, sp, -0x10  
    sd     s0, 0x0(sp)  
    sd     ra, 0x8(sp)  
    addi   s0, sp, 0x10  
    ld     a4, 0x0(a0)  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
  
<ftrace_call>:  
    REG_Lra, ftrace_call_dest  
    jalr0(ra)  
    restore_regs  
    jr t0  
  
<ftrace_stub>:  
    jr t0
```

```
<my_tracer>:  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]
```

load

```
ftrace_func_t ftrace_call_dest = ftrace_stub;
```

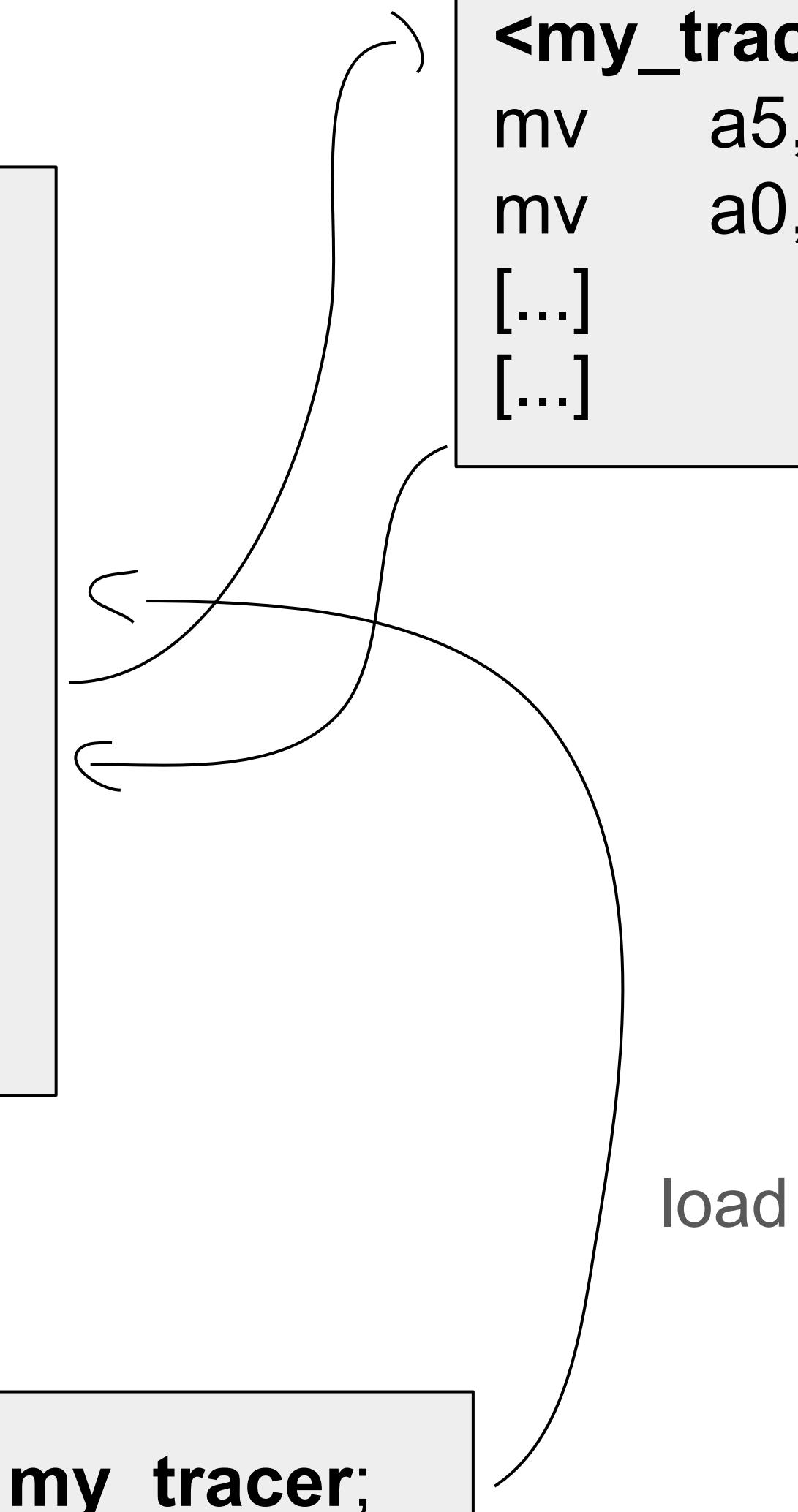
# Proposed Solution Implementation

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    nop  
    addi   sp, sp, -0x10  
    sd     s0, 0x0(sp)  
    sd     ra, 0x8(sp)  
    addi   s0, sp, 0x10  
    ld     a4, 0x0(a0)  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
  
<ftrace_call>:  
    REG_Lra, ftrace_call_dest  
    jalr0(ra)  
    restore_regs  
    jr t0  
  
<ftrace_stub>:  
    jr t0
```

```
ftrace_func_t ftrace_call_dest = my_tracer;
```

```
<my_tracer>:  
    mv     a5, a0  
    mv     a0, a1  
    [...]  
    [...]
```



ftrace\_stub → my\_tracer

# Proposed Solution Implementation

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
  
<ftrace_call>:  
    REG_Lra, ftrace_call_dest  
    jalr0(ra)  
    restore_regs  
    jr t0  
  
<ftrace_stub>:  
    jr t0
```

```
<my_tracer>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
ftrace_func_t ftrace_call_dest = my_tracer;
```

ftrace\_stub → my\_tracer

## Two problems:

- Our solution added overhead to direct calls as the function entry can only call `ftrace_caller`. All direct calls have to go through `ftrace_caller`.
- Because `ftrace_caller` trampoline is common to all traced functions, we have to call `ftrace_ops_list_func` even for functions that are only traced by one function.

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_ops_list_func  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
<ftrace_ops_list_func>:  
/* Iterate check and call */
```

```
<vfs_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<vfs_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_ops_list_func  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
<ftrace_ops_list_func>:  
/* Iterate check and call */
```

```
<vfs_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<vfs_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<wake_up_process>:  
    nop  
    nop  
    sp, sp, -0x10  
    s0, 0x0(sp)  
    ra, 0x8(sp)  
    s0, sp, 0x10  
    a2, 0x0  
    a1, 0x3  
    auipc ra, 0x0  
    jalr -0x53e(ra) <try_to_wake_up>  
    [...]  
    [...]  
    [...]
```

```
<vfs_open>:  
    auipc t0, ftrace_caller  
    jalr t0, ftrace_caller(t0)  
    addi sp, sp, -0x10  
    sd s0, 0x0(sp)  
    sd ra, 0x8(sp)  
    addi s0, sp, 0x10  
    ld a4, 0x0(a0)  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]  
    [...]  
    [...]  
    [...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call ftrace_ops_list_func  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

```
<ftrace_ops_list_func>:  
/* Iterate check and call */
```

```
<vfs_tracer1>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<vfs_tracer2>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

```
<wake_up_process>:  
    nop  
    nop  
    sp, sp, -0x10  
    s0, 0x0(sp)  
    ra, 0x8(sp)  
    s0, sp, 0x10  
    a2, 0x0  
    a1, 0x3  
    auipc ra, 0x0  
    jalr -0x53e(ra) <try_to_wake_up>  
    [...]  
    [...]  
    [...]
```

Want to trace with

```
<wake_up_tracer>:  
    mv a5, a0  
    mv a0, a1  
    [...]  
    [...]
```

# Possible solutions?

- Can't do direct call from entry of `wake_up_process` to `wake_up_tracer` because `wake_up_tracer` can't receive direct calls.
- Let `ftrace_caller` handle it through `ftrace_ops_list_func` but this will add the overhead of tracing `vfs_open` on tracing `wake_up_process` [RISC-V does this currently]
- Dynamically allocate a new trampoline like `ftrace_caller` just for `wake_up_process` and call that. [x86 does this]
- Or implement call ops! [ARM64 does this]

# What is CALL\_OPS

- We allocate 8 bytes before the function entry and put a pointer to ftrace\_ops there
- Every patchable function can put a ftrace\_ops pointer before its entry and ftrace\_caller will fetch this pointer and call ftrace\_ops->func
- Allow each callsite to provide its ftrace\_ops to ftrace\_caller and thus we don't need to patch ftrace\_caller at runtime. [ARM64 does this already]

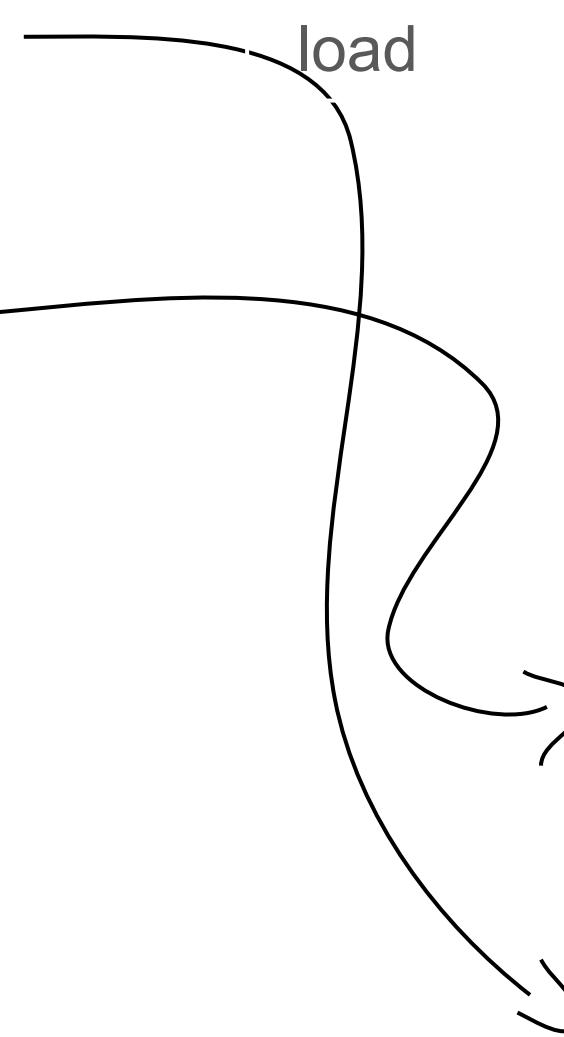
# CALL OPS

```
<vfs_open>:  
auipc t0, ftrace_caller  
jalr t0, ftrace_caller(t0)  
addi    sp, sp, -0x10  
sd      s0, 0x0(sp)  
sd      ra, 0x8(sp)  
addi    s0, sp, 0x10  
ld      a4, 0x0(a0)  
mv      a5, a0  
mv      a0, a1  
[...]  
[...]  
[...]  
[...]  
[...]
```

```
<ftrace_caller>:  
    save_regs  
    load_regs  
<ftrace_call>:  
    call my_tracer  
    restore_regs  
    jr t0  
<ftrace_stub>:  
    jr t0
```

# CALL\_OPS

```
<ftrace_ops pointer>
<vfs_open>:
    auipc t0, ftrace_caller
    jalr t0, ftrace_caller(t0)
    addi sp, sp, -0x10
    sd s0, 0x0(sp)
    sd ra, 0x8(sp)
    addi s0, sp, 0x10
    ld a4, 0x0(a0)
    mv a5, a0
    mv a0, a1
    [...]
    [...]
    [...]
    [...]
```



```
<ftrace_caller>:
    save_regs
    load_regs
<ftrace_call>:
    REG_L a2, -16(t0) // ftrace_ops
    REG_L ra, FTRACE_OPS_FUNC(a2) // ftrace_ops->func
    jalr ra
    restore_regs
    jr t0
<ftrace_stub>:
    jr t0
```

```
<my_tracer>:
    mv a5, a0
    mv a0, a1
    [...]
    [...]
```

# Summary

- Move from regs to args [1] [Merged]
  - We only need calls to `ftrace_caller` now.
- Stop using `stop_machine` by patching single instruction [2]
- Move to call ops: don't need to patch `ftrace_caller` [3]

[1] [\[PATCH\] ftrace: riscv: move from REGS to ARGS](#)

[2] [\[PATCH v2 0/6\] riscv: ftrace: atomic patching and preempt improvements](#)

[3] [\[RFC PATCH\] riscv: Implement HAVE\\_DYNAMIC\\_FTRACE\\_WITH\\_CALL\\_OPS](#)

# Discussion + Q/A

- Direct call now need to go through `ftrace_caller`
- `.text` size increases!
- If we put a pointer to `ftrace_ops` above the function entry, and want to atomically modify it, all functions need to be aligned at 8B.
- Some other ways:
  - trigger re-execution on updated AUIPC
  - Indirections
- The Assumption – Ziccif
  - ... Instruction fetches of naturally aligned power-of-2 sizes up to  $\min(\text{ILEN}, \text{XLEN})$  (i.e., 32 bits for RVA20) are atomic.
  - Static branch in riscv Linux assumes Ziccif.
  - Do we want to maintain static branch on a platform that does not support Ziccif?
  - Example of machines that do not implement Ziccif:
    - QEMU:
      - fetch lower 2 bytes to get opcode, then fetch upper 2 bytes if it is a 4 bytes instruction
  - Or, treat it as a hardware bug if not implemented

# The Problem and Current Solution of Atomic Patching

- The Problem:
  - Impossible to concurrently modify and execute 2 instructions (AUIPC + JALR).
- The solution:
  - Only patch 1 instruction and limit the jump range:
    - Point AUIPC to the ftrace trampoline at boot time, and start/stop tracing by patching JALR/NOP to the location of the second instruction.
    - The range is limited to +-2KB (CALL\_OPS should solve this).
- Some other ways:
  - trigger re-execution on updated AUIPC
  - Indirections



## The Assumption – Ziccif

... Instruction fetches of naturally aligned power-of-2 sizes up to  $\min(ILEN, XLEN)$  (i.e., 32 bits for RVA20) are atomic.

- Static branch in riscv Linux assumes Ziccif.
- Do we want to maintain static branch on a platform that does not support Ziccif?
- Example of machines that do not implement Ziccif:
  - QEMU:
    - fetch lower 2 bytes to get opcode, then fetch upper 2 bytes if it is a 4 bytes instruction
  - Or, treat it as a hardware bug if not implemented

