Linux Plumbers Conference

Vienna, Austria | September 18-20, 2024

Atomic code patching and ftrace

Puranjay Mohan and Tao Chiu

The Ftrace mechanism for dummies

```
<vfs_open>:
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
[...]
[...]
[...]
[...]
[...]
[...]
[...]
```

Aim: To call my_tracer for all calls to vfs_open.

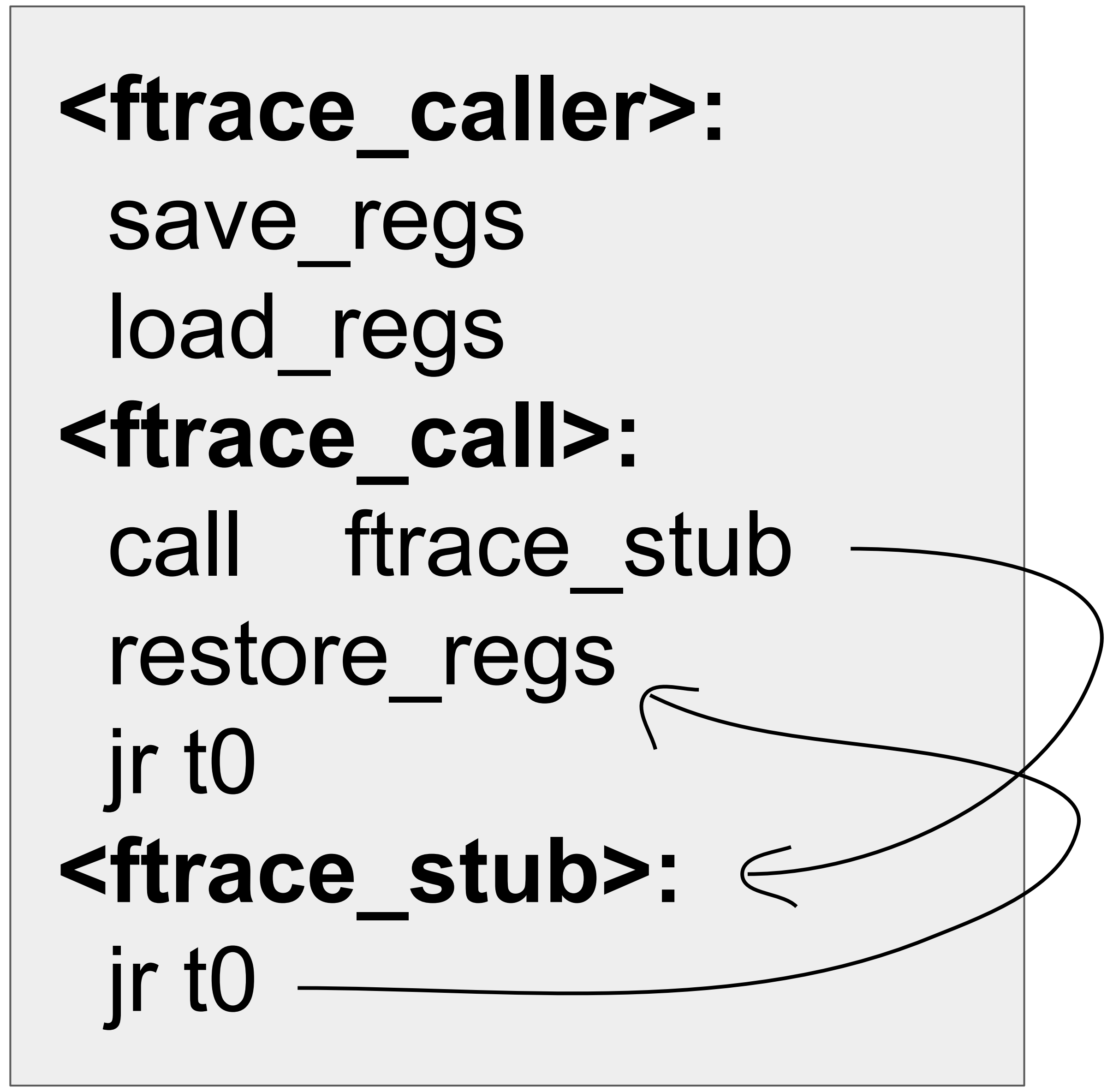
```
<my_tracer>:
mv      a5, a0
mv      a0, a1
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
[...]
```

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
  [...]
  [...]
  [...]
  [...]
  [...]
```

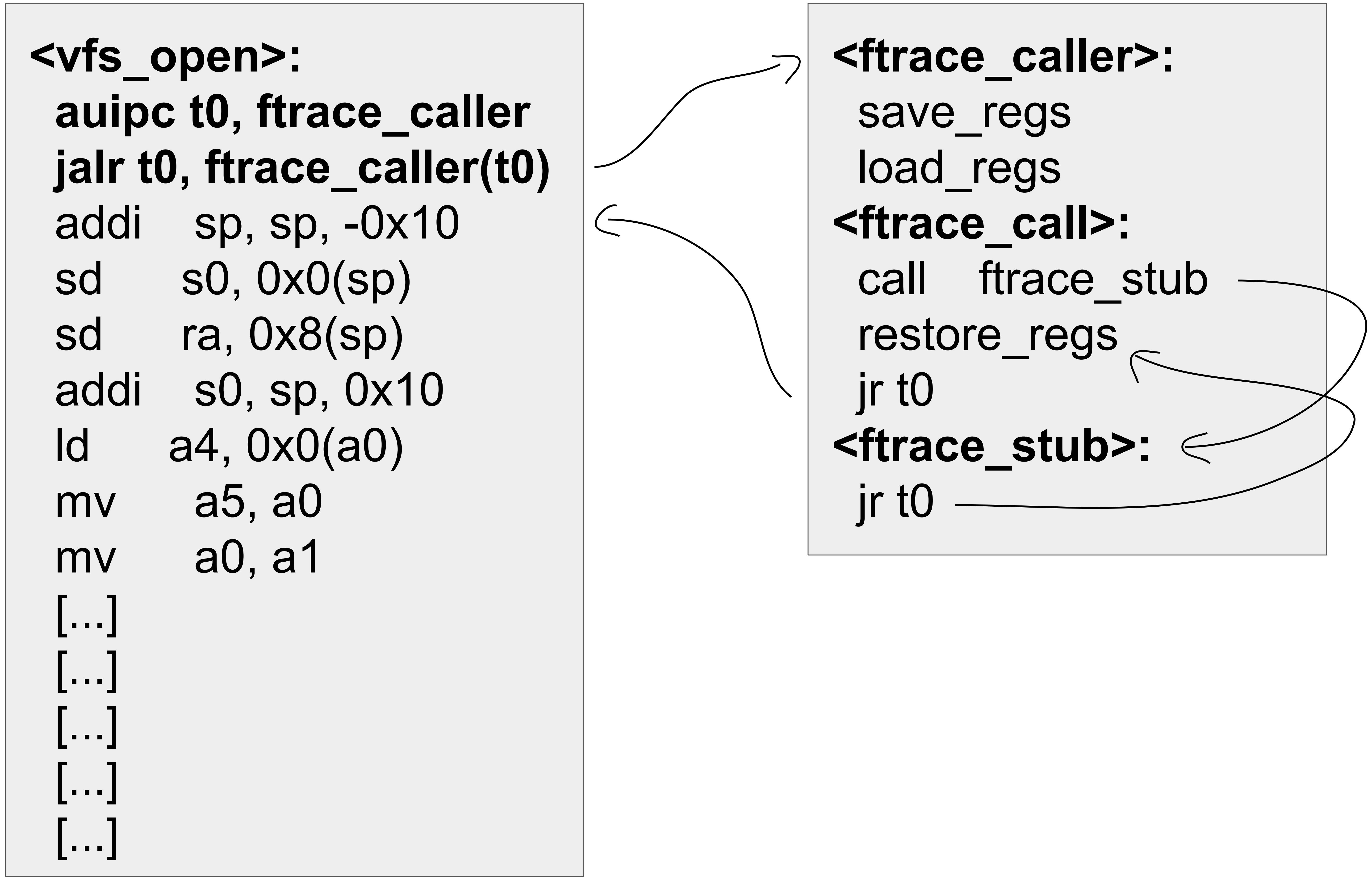
The Ftrace mechanism for dummies

```
<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
```

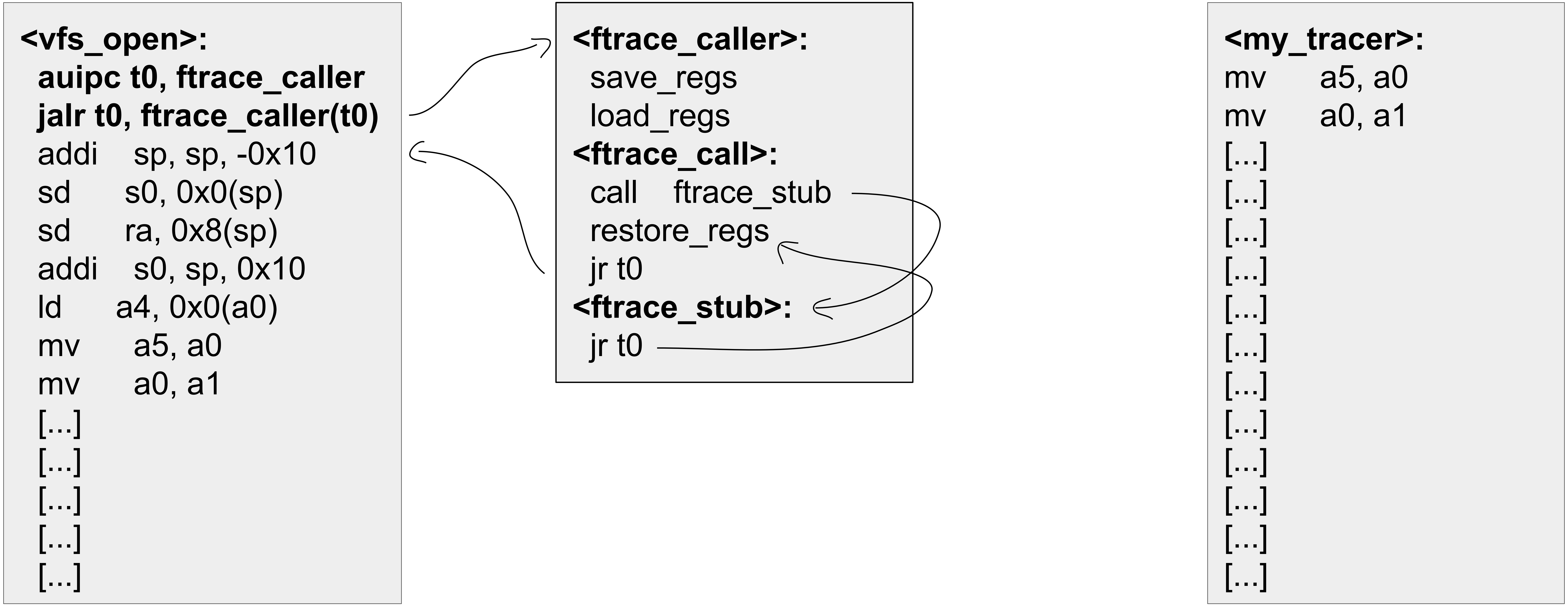
The Ftrace mechanism for dummies



```
<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_stub
  restore_regs
  jr t0
<ftrace_stub>:
  jr t0
```

The Ftrace mechanism for dummies

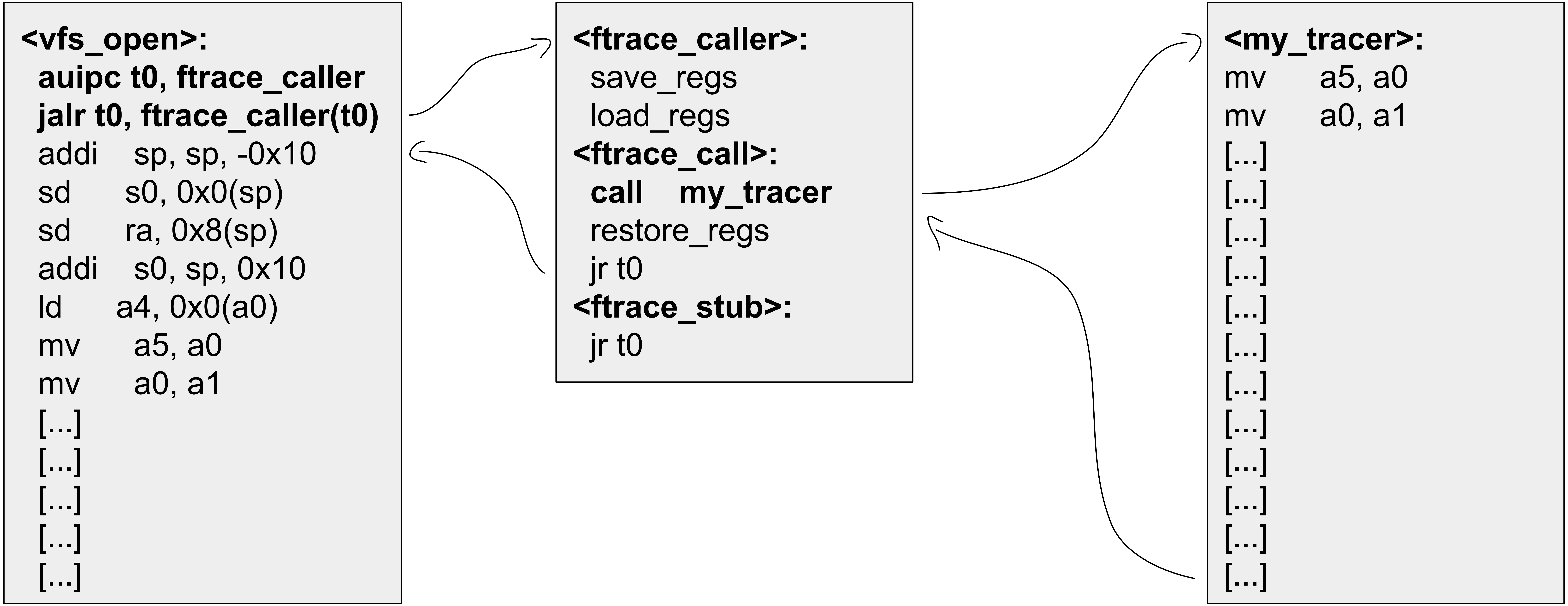


```
<vfs_open>:
  auipc t0, ftrace_caller
  jalr t0, ftrace_caller(t0)
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

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

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

The Ftrace mechanism for dummies



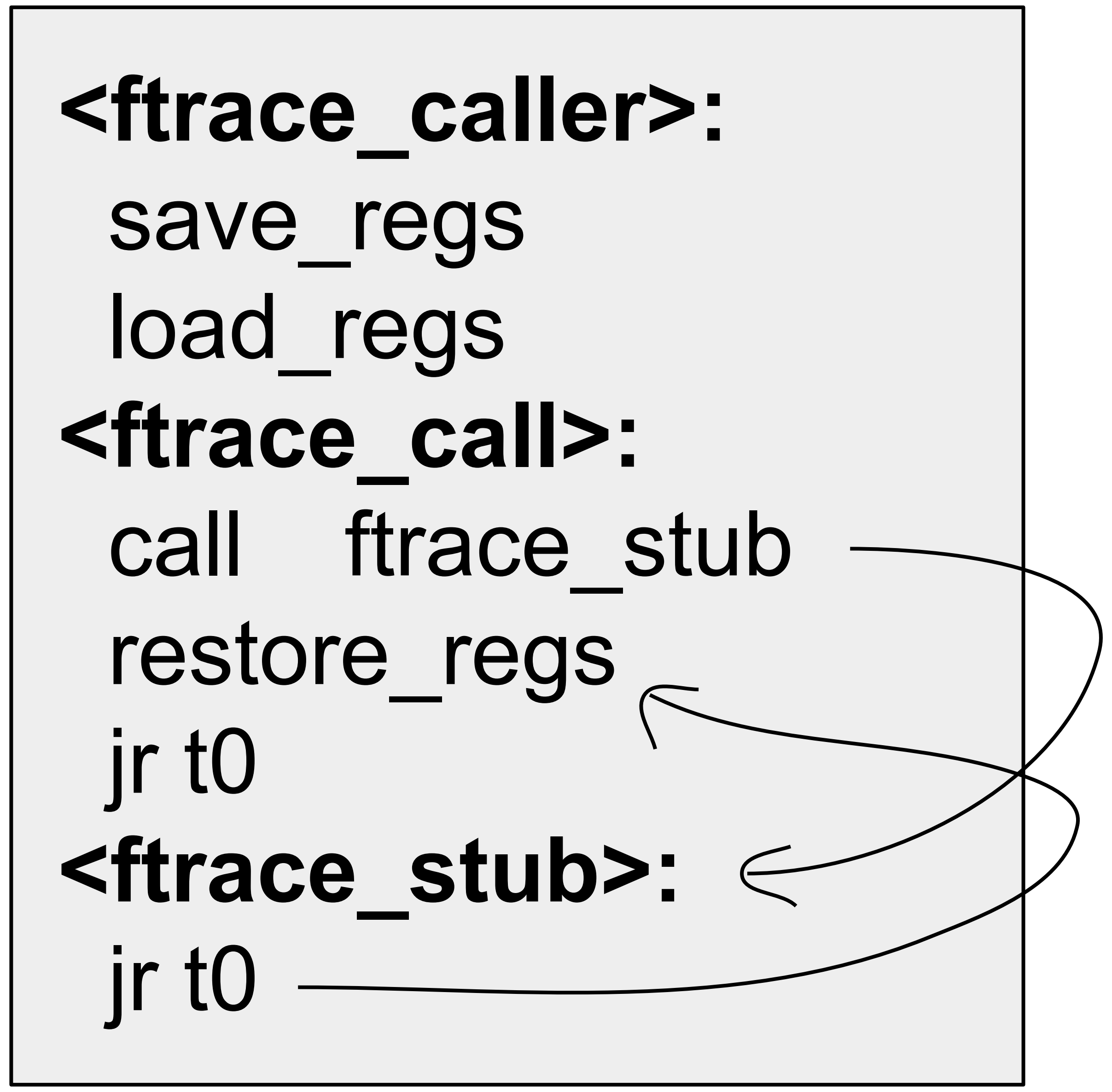
```
<vfs_open>:
  auipc t0, ftrace_caller
  jalr t0, ftrace_caller(t0)
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

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

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

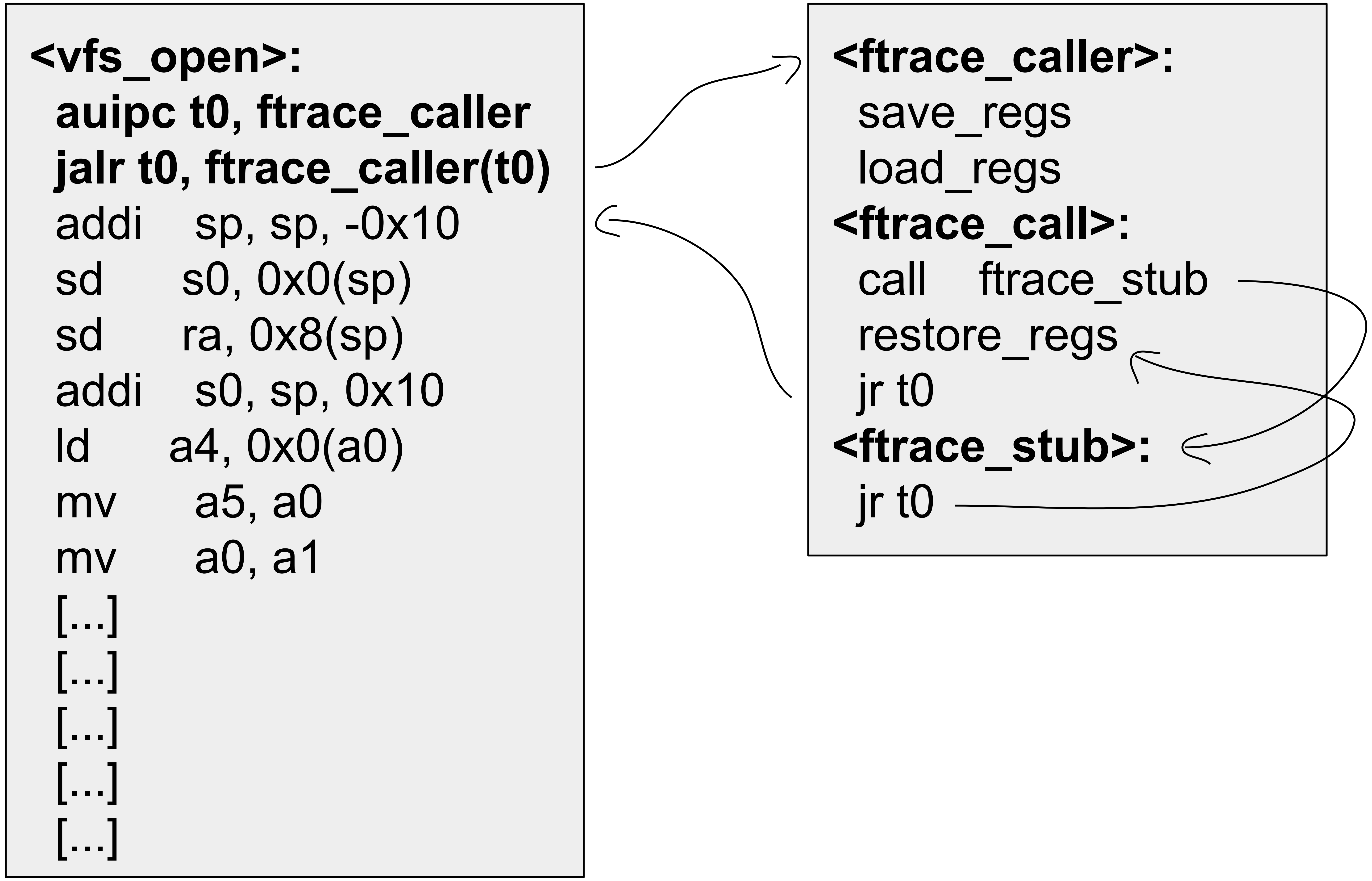
The Ftrace mechanism for dummies

```
<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
  [...]
  [...]
  [...]
  [...]
  [...]
```



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

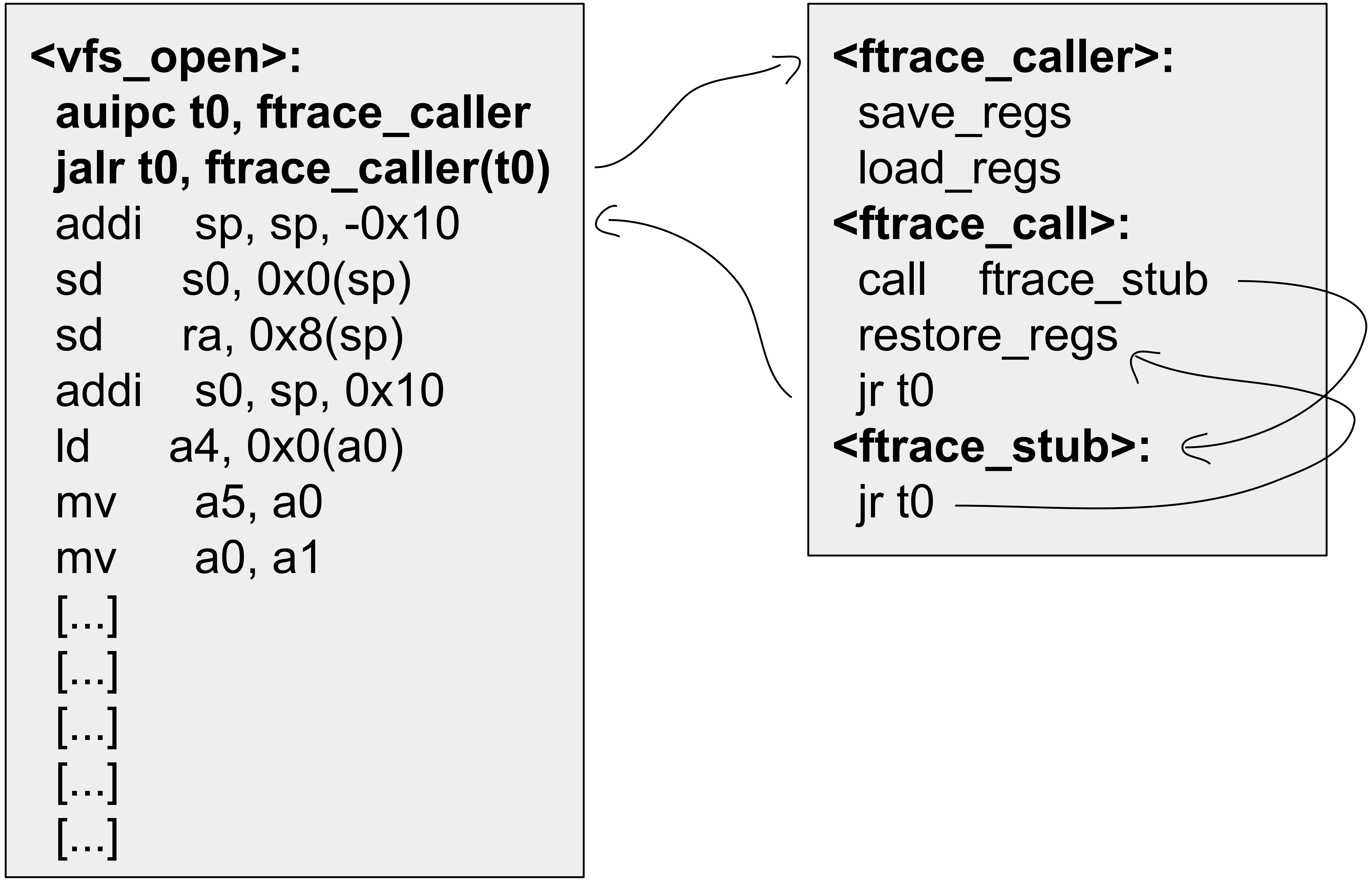
The Ftrace mechanism for dummies



```
<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
  [...]
  [...]
  [...]
  [...]
  [...]
```

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

The Ftrace mechanism for dummies



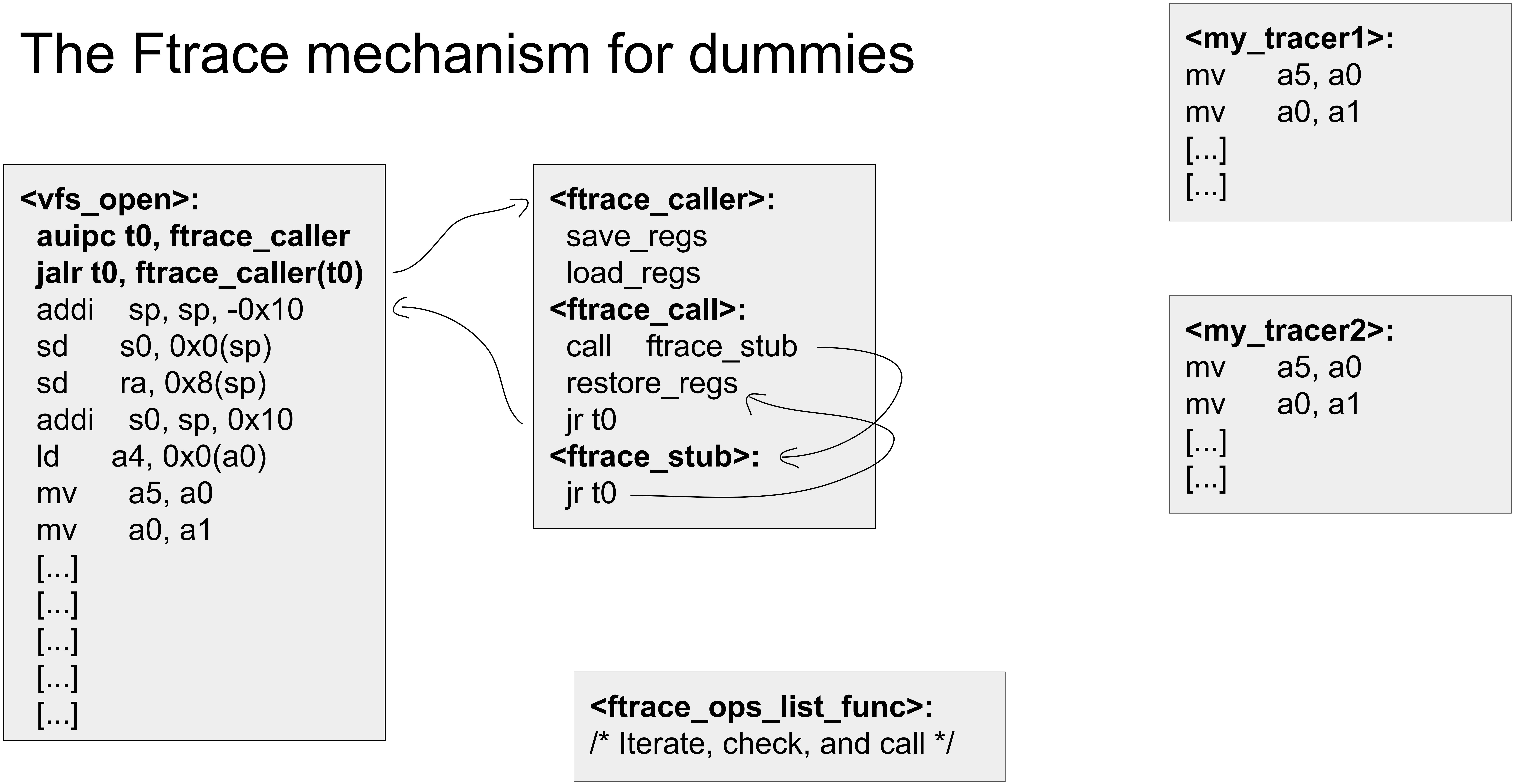
```
<vfs_open>:
  auipc t0, ftrace_caller
  jalr t0, ftrace_caller(t0)
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

```
<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
  [...]
  [...]
  [...]
  [...]
  [...]
```

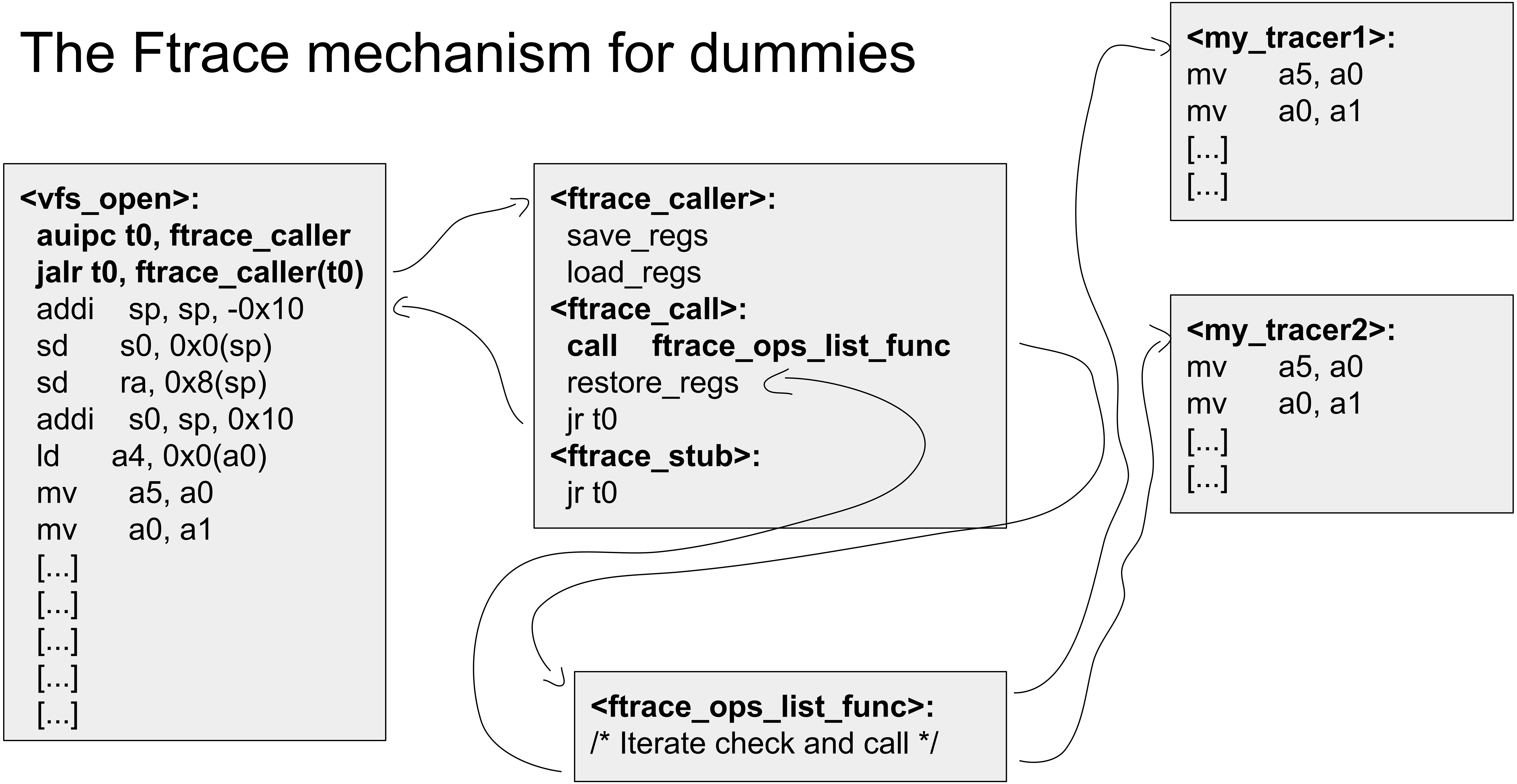
```
<ftrace_caller>:
  save_regs
  load_regs
<ftrace_call>:
  call    ftrace_stub
  restore_regs
  jr t0
<ftrace_stub>:
  jr t0
```

```
<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
  [...]
  [...]
  [...]
  [...]
  [...]
```

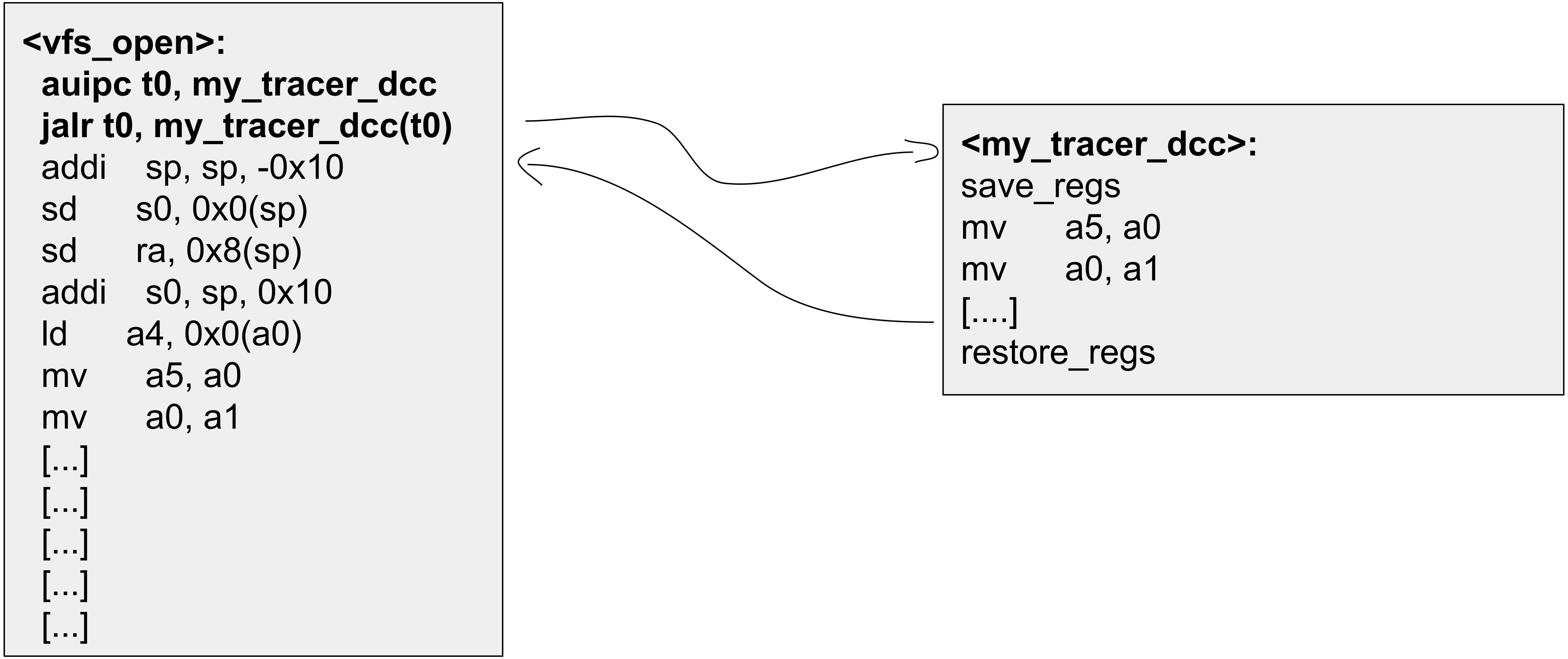
```
<ftrace_caller>:
  save_regs
  load_regs
<ftrace_call>:
  call    ftrace_ops_list_func
  restore_regs
  jr t0
<ftrace_stub>:
  jr t0
```

```
<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.

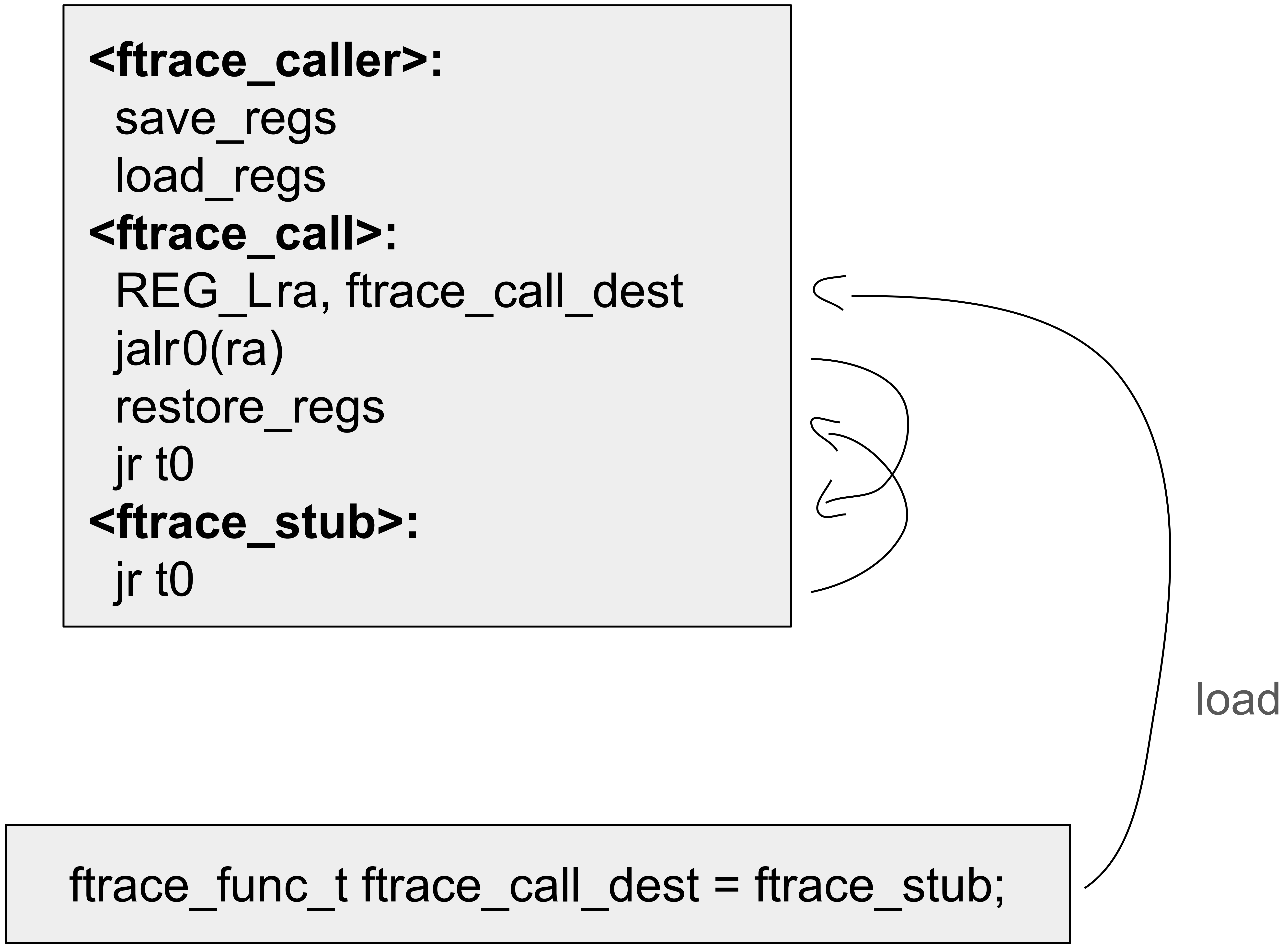
- Looking at this from a lower level:
    - Function entries have an `auipc+jalr` pair [To be atomically patched]
    - Ftrace_caller trampoline has an `auipc+jalr` pair [To be atomically patched]

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` ←→ `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

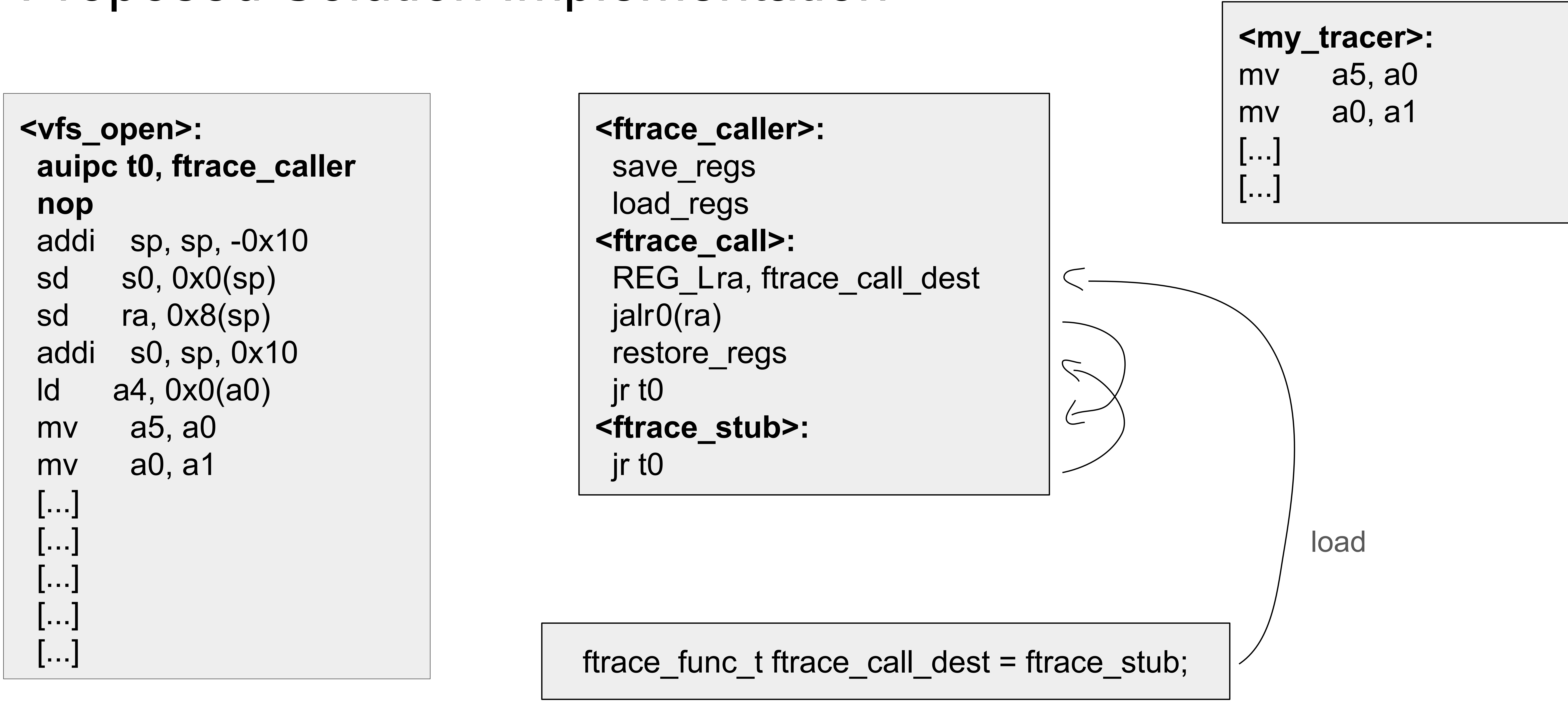
```
<vfs_open>:
  auipc t0, ftrace_caller
  nop
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```



```
<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



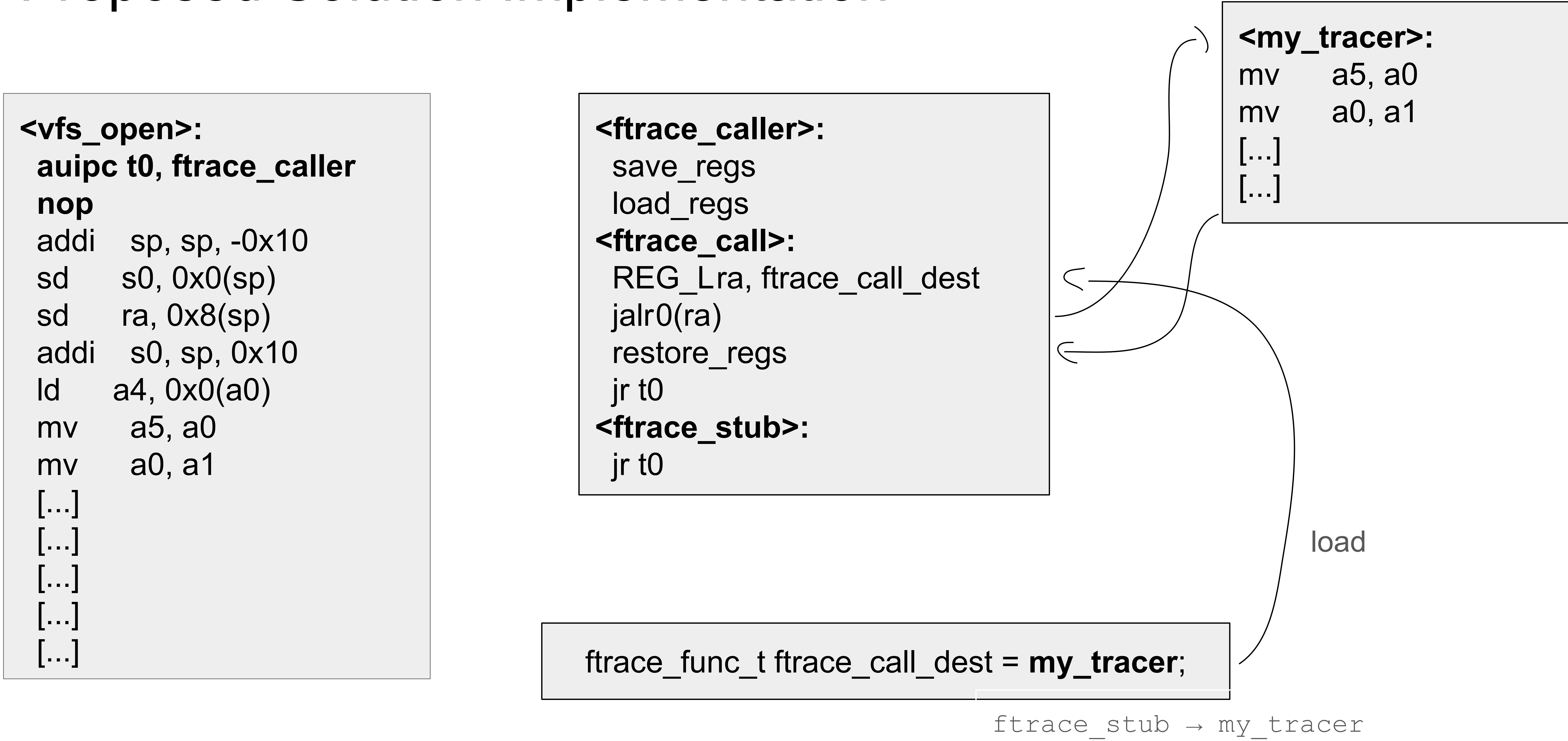
```
<vfs_open>:
  auipc t0, ftrace_caller
  nop
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

```
<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
[...]
[...]
```

```
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
  [...]
  [...]
  [...]
  [...]
  [...]
```

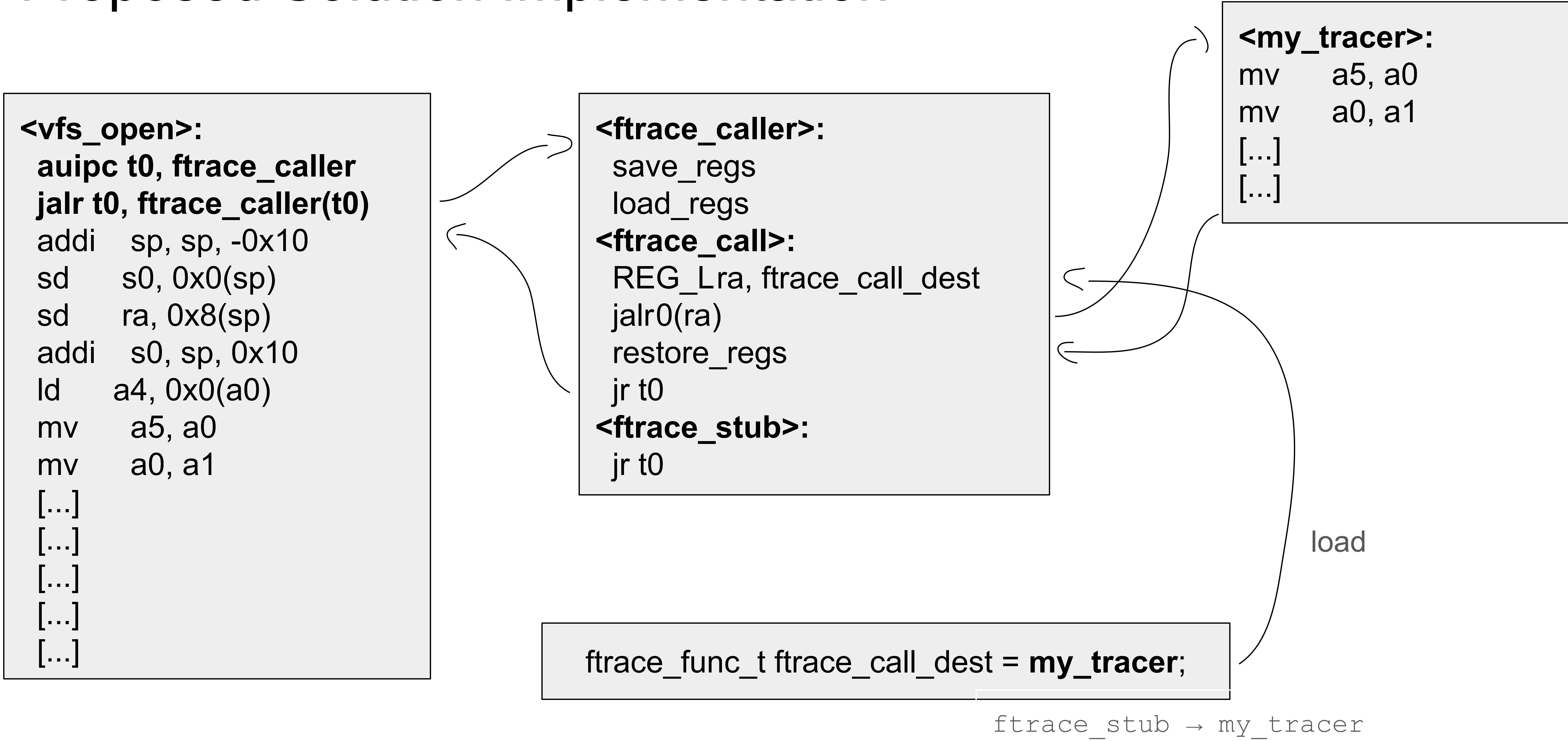
```
<ftrace_caller>:
  save_regs
  load_regs
<ftrace_call>:
  REG_Lra, ftrace_call_dest
  jalr0(ra)
  restore_regs
  jr t0
<ftrace_stub>:
  jr t0
```

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

```
ftrace_func_t ftrace_call_dest = my_tracer;
```

`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
[...]
[...]
```

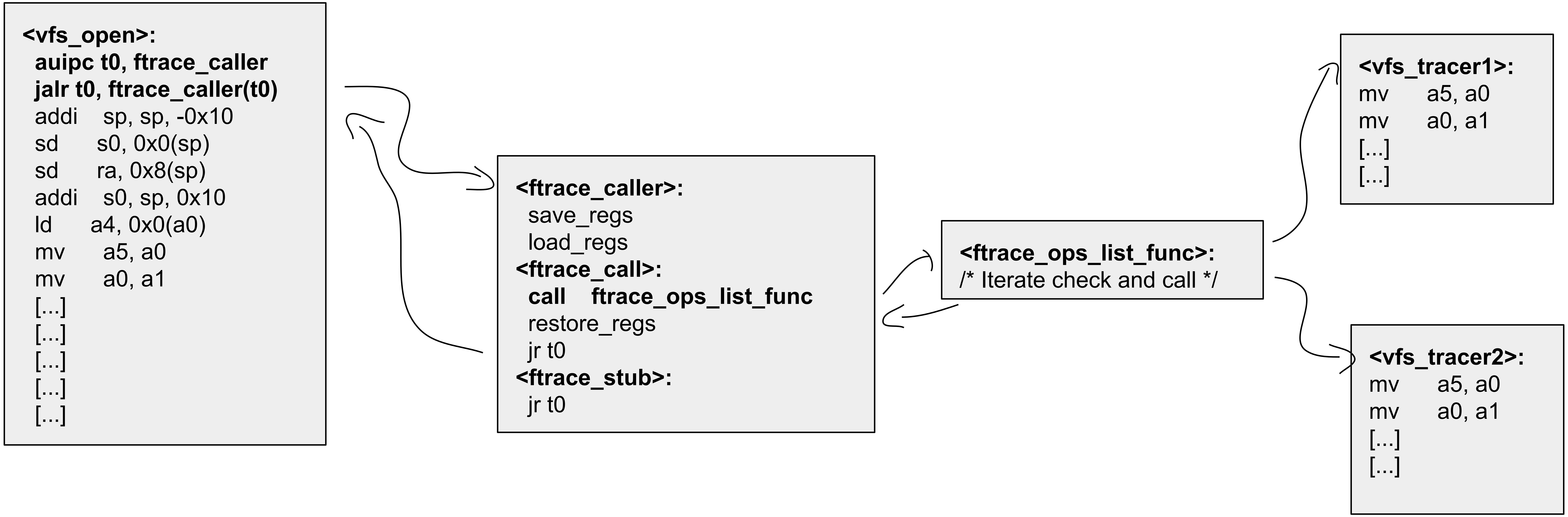
load

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.



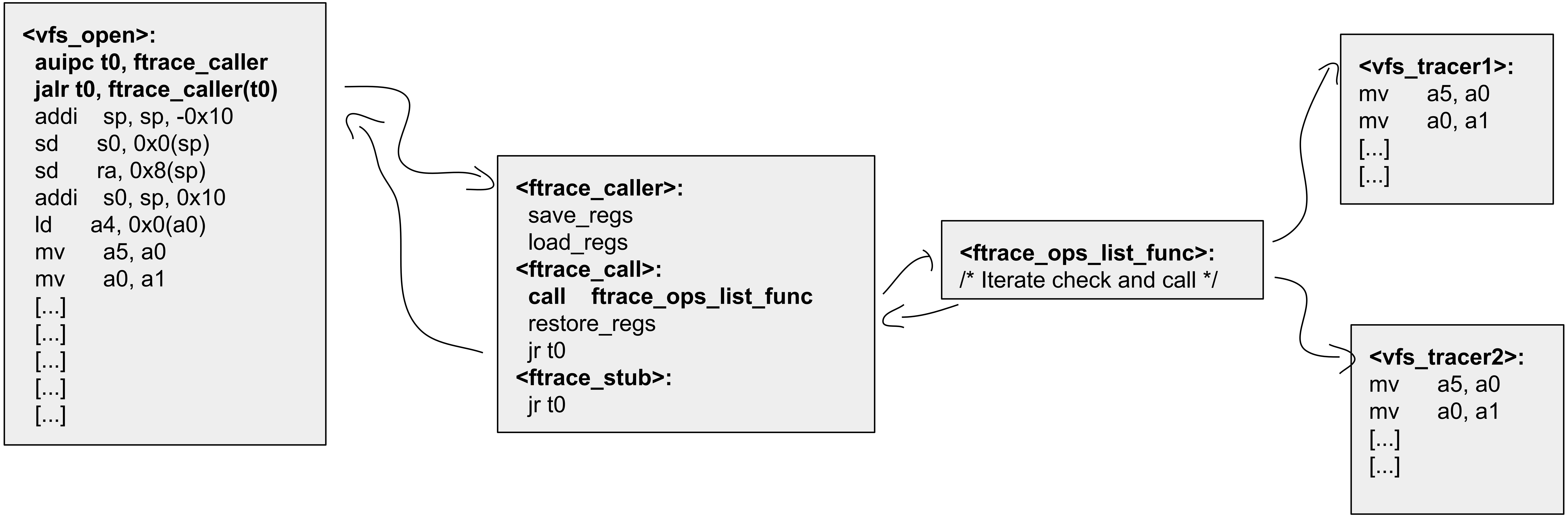
```
<vfs_open>:
  auipc t0, ftrace_caller
  jalr t0, ftrace_caller(t0)
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

```
<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
  [...]
  [...]
  [...]
  [...]
  [...]
```

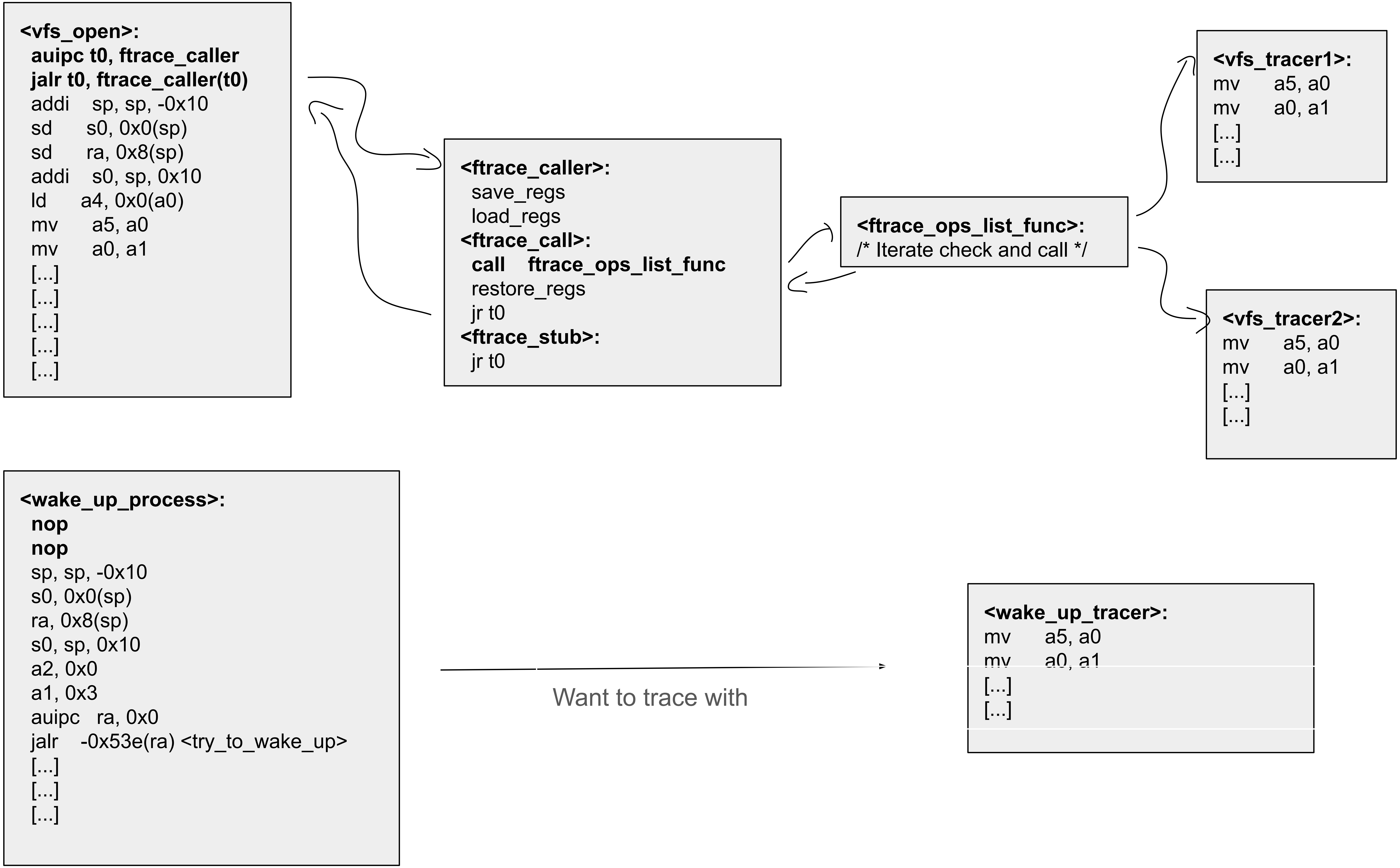
```
<ftrace_caller>:
  save_regs
  load_regs
<ftrace_call>:
  call    ftrace_ops_list_func
  restore_regs
  jr t0
<ftrace_stub>:
  jr t0
```

```
<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>
  [...]
  [...]
  [...]
```

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

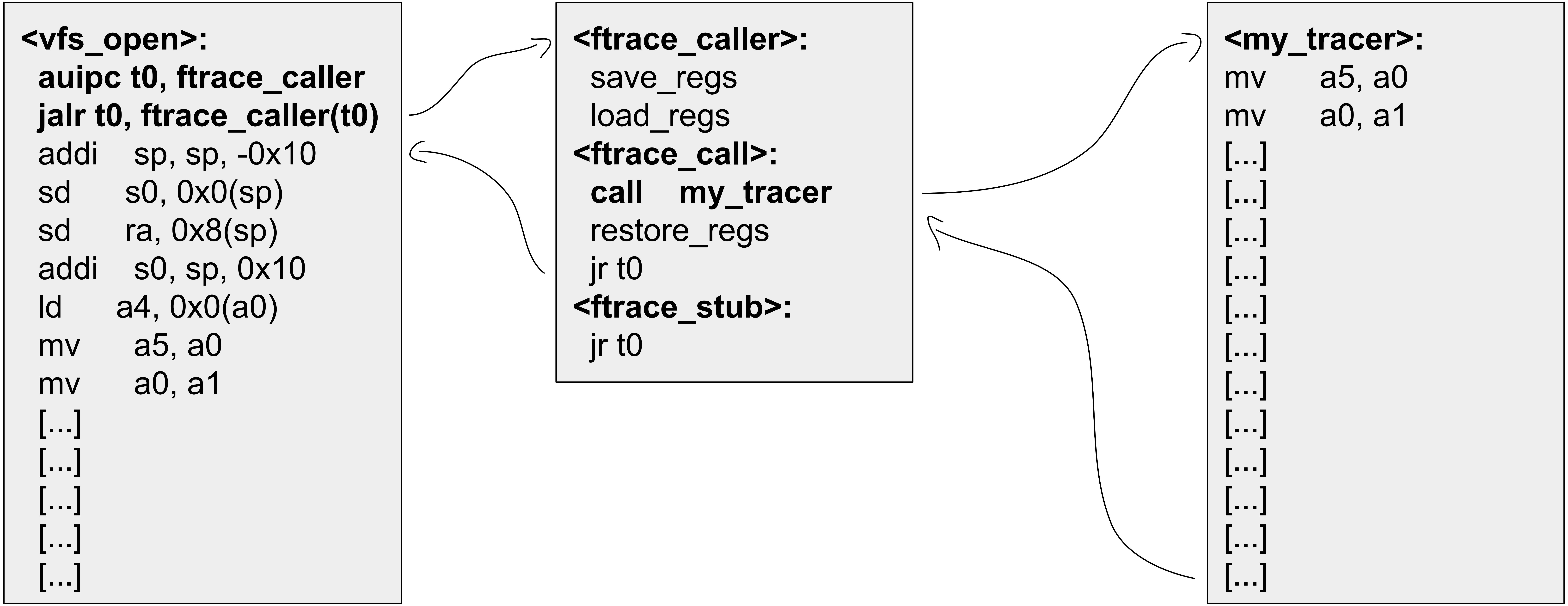
Possible solutions?

- 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

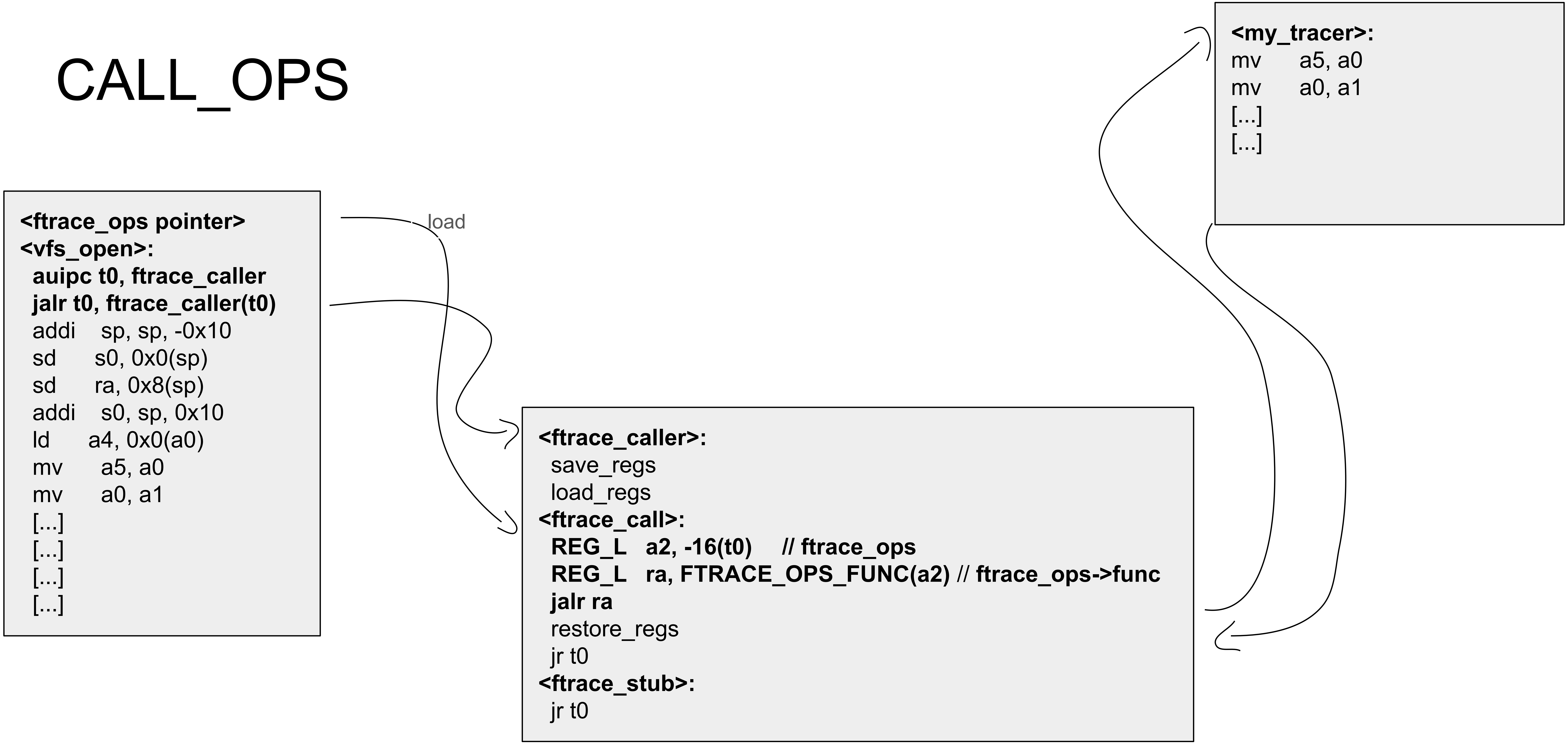


```
<vfs_open>:
  auipc t0, ftrace_caller
  jalr t0, ftrace_caller(t0)
  addi    sp, sp, -0x10
  sd      s0, 0x0(sp)
  sd      ra, 0x8(sp)
  addi    s0, sp, 0x10
  ld      a4, 0x0(a0)
  mv      a5, a0
  mv      a0, a1
  [...]
  [...]
  [...]
  [...]
  [...]
```

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

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

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: atmoic 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(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

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