SiFive

RISC-V ftrace
Working with Preemption

September 14th, 2022 ● RISC-V MC in Linux Plumbers”22 Dublin

About

Name: Andy

Origin: Taiwan

Working at: System Software, SiFive (Since Oct., 2021)

Outline

- Introduction
- Current Implementation of RISC-V ftrace
- RISC-V ftrace code patching and stop_machine()
- Mixing with Kernel Preemption
- Reviews of ftrace Implementations on other Architectures
- How to patch code atomically in RISC-V
- Possible Solutions to Enable ftrace on a Preemptible Kernel
- Proposed Solution

Introduction

```
iscv64:/sys/kernel/tracing# echo nop > current_tracer
iscv64:/sys/kernel/tracing# echo function > current_tracer
iscv64:/sys/kernel/tracing# cat trace
function

in-buffer/entries-written: 34421/34421   #P:2

                       _-----=> irqs-off/BH-disabled
                      / _----=> need-resched
                     | / _---=> hardirq/softirq
                     || / _--=> preempt-depth
                     ||| / _-=> migrate-disable
                     |||| /     delay
  TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
     | |         |   |||||     |         |
    sh-433     [001] ...1.   167.410206: mutex_unlock <-tracing_set_
    sh-433     [001] ...1.   167.410284: preempt_count_add <-vfs_wri
    sh-433     [001] ...2.   167.410288: preempt_count_sub <-vfs_wri
    sh-433     [001] ...1.   167.410321: sys_dup3 <-ret_from_syscall
    sh-433     [001] ...1.   167.410325: ksys_dup3 <-sys_dup3
    sh-433     [001] ...1.   167.410327: _raw_spin_lock <-ksys_dup3
```

ftrace is a tracing framework aiming to

- trace Linux kernel at function calls level
- enable/disable dynamically, without recompiling the kernel
- introduce minimal overhead

Introduction

```
iscv64:/sys/kernel/tracing# echo nop > current_tracer
iscv64:/sys/kernel/tracing# echo function > current_tracer
iscv64:/sys/kernel/tracing# cat trace
function

in-buffer/entries-written: 34421/34421   #P:2

                            _-----=> irqs-off/BH-disabled
                           / _----=> need-resched
                          | / _---=> hardirq/softirq
                          || / _--=> preempt-depth
                          ||| / _-=> migrate-disable
                          |||| /     delay
  TASK-PID      CPU#     |||||  TIMESTAMP  FUNCTION
     | |          |      |||||     |         |
    sh-433      [001] ...1.   167.410206: mutex_unlock <-tracing_set_
    sh-433      [001] ...1.   167.410284: preempt_count_add <-vfs_wri
    sh-433      [001] ...2.   167.410288: preempt_count_sub <-vfs_wri
    sh-433      [001] ...1.   167.410321: sys_dup3 <-ret_from_syscall
    sh-433      [001] ...1.   167.410325: ksys_dup3 <-sys_dup3
    sh-433      [001] ...1.   167.410327: _raw_spin_lock <-ksys_dup3
```

ftrace is a tracing framework aiming to

- trace Linux kernel at function calls level
- enable/disable dynamically, without recompiling the kernel
- introduce minimal overhead

Introduction

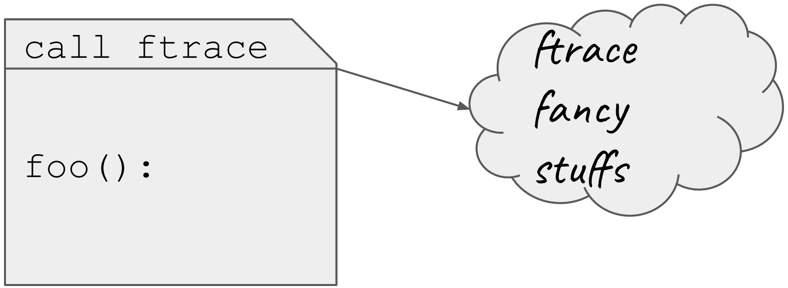
```
iscv64:/sys/kernel/tracing# echo nop > current_tracer
iscv64:/sys/kernel/tracing# echo function > current_tracer
iscv64:/sys/kernel/tracing# cat trace
function

in-buffer/entries-written: 34421/34421   #P:2

                            _-----=> irqs-off/BH-disabled
                           / _----=> need-resched
                          | / _---=> hardirq/softirq
                          || / _--=> preempt-depth
                          ||| / _-=> migrate-disable
                          |||| /     delay
  TASK-PID      CPU#      |||||  TIMESTAMP  FUNCTION
     | |          |       |||||     |          |
    sh-433      [001] ...1.   167.410206: mutex_unlock <-tracing_set_
    sh-433      [001] ...1.   167.410284: preempt_count_add <-vfs_wri
    sh-433      [001] ...2.   167.410288: preempt_count_sub <-vfs_wri
    sh-433      [001] ...1.   167.410321: sys_dup3 <-ret_from_syscall
    sh-433      [001] ...1.   167.410325: ksys_dup3 <-sys_dup3
    sh-433      [001] ...1.   167.410327: _raw_spin_lock <-ksys_dup3
```

ftrace is a tracing framework aiming to

- trace Linux kernel at function calls level
- enable/disable dynamically, without recompiling the kernel
- introduce minimal overhead



Introduction

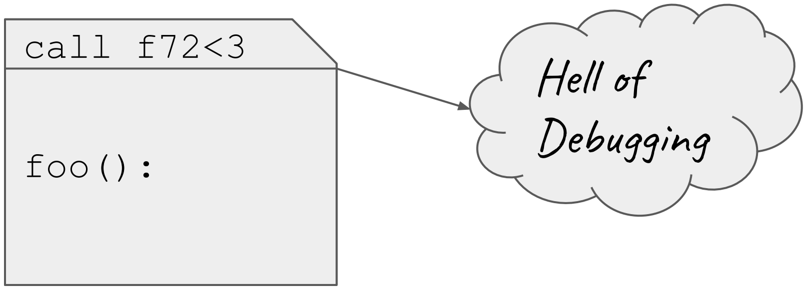
```
iscv64:/sys/kernel/tracing# echo nop > current_tracer
iscv64:/sys/kernel/tracing# echo function > current_tracer
iscv64:/sys/kernel/tracing# cat trace
function

in-buffer/entries-written: 34421/34421   #P:2

                          _-----=> irqs-off/BH-disabled
                         / _----=> need-resched
                        | / _---=> hardirq/softirq
                        || / _--=> preempt-depth
                        ||| / _-=> migrate-disable
                        |||| /     delay
  TASK-PID      CPU#    |||||  TIMESTAMP  FUNCTION
     | |          |     |||||     |         |
    sh-433      [001] ...1.   167.410206: mutex_unlock <-tracing_set_
    sh-433      [001] ...1.   167.410284: preempt_count_add <-vfs_wri
    sh-433      [001] ...2.   167.410288: preempt_count_sub <-vfs_wri
    sh-433      [001] ...1.   167.410321: sys_dup3 <-ret_from_syscall
    sh-433      [001] ...1.   167.410325: ksys_dup3 <-sys_dup3
    sh-433      [001] ...1.   167.410327: _raw_spin_lock <-ksys_dup3
```

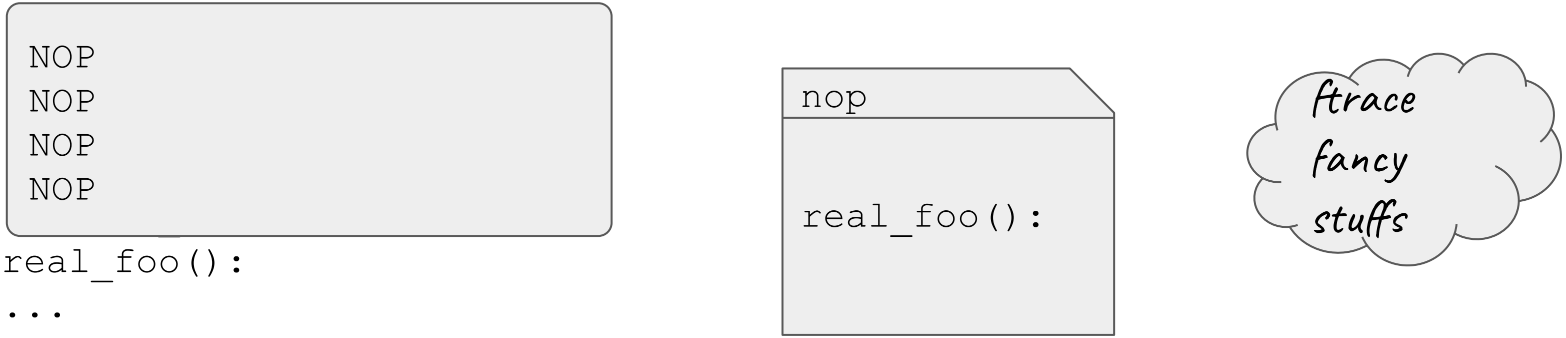
ftrace is a tracing framework aiming to

- trace Linux kernel at function calls level
- enable/disable dynamically, without recompiling the kernel
- introduce minimal overhead



Current Implementation of RISC-V ftrace

Use `-fpatchable-function-entry=8` to reserve nop paddings on each traceable function

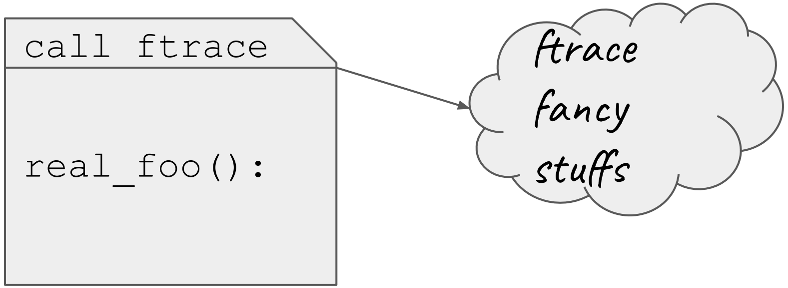


Current Implementation of RISC-V ftrace

Use `-fpatchable-function-entry=8` to reserve nop paddings on each traceable function

- In general, RISC-V uses a pair of `auipc` and `jalr` to perform a call

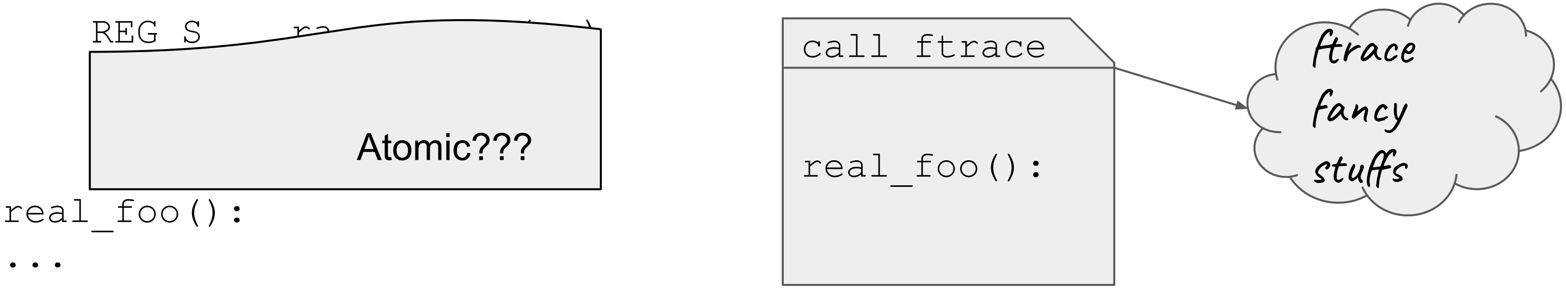
```
    REG_S     ra, -SZREG(sp)
    auipc     ra, 0x?????
    jalr      0x???(ra)
    REG_L     ra, -SZREG(sp)
real_foo():
...
```



Current Implementation of RISC-V ftrace

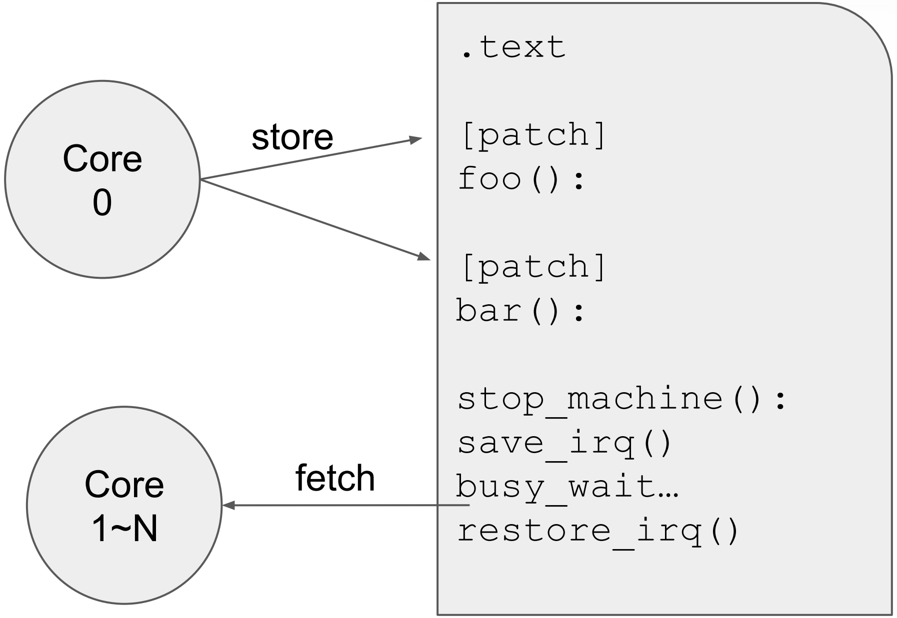
Use `-fpatchable-function-entry=8` to reserve nop paddings on each traceable function

- In general, RISC-V uses a pair of `auipc` and `jalr` to perform a call
- The key is, the change have to be seen atomically to other cores.
- In other words, patching must “happen before” cores running on it.



RISC-V ftrace code patching and `stop_machine()`

forcing all other cores into a busy wait loop while a core is performing the critical job.



RISC-V ftrace code patching and `stop_machine()`

forcing all other cores into a busy wait loop while a core is performing the critical job.

```
	/* Simple state machine */
	do {
		/* Chill out and ensure we re-read multi_stop_state. */
		stop_machine_yield(cpumask);
		newstate = READ_ONCE(msdata->state);
		if (newstate != curstate) {
			curstate = newstate;
			switch (curstate) {
			case MULTI_STOP_DISABLE_IRQ:
...
			case MULTI_STOP_RUN:
				if (is_active)
					err = msdata->fn(msdata->data);
				break;
...
			}
			ack_state(msdata);
		}
...
		rcu_momentary_dyntick_idle();
	} while (curstate != MULTI_STOP_EXIT);
```

RISC-V ftrace code patching and `stop_machine()`

forcing all other cores into a busy wait loop while a core is performing the critical job.

```
	/* Simple state machine */
	do {
		/* Chill out and ensure we re-read multi_stop_state. */
		stop_machine_yield(cpumask);
		newstate = READ_ONCE(msdata->state);
		if (newstate != curstate) {
			curstate = newstate;
			switch (curstate) {
			case MULTI_STOP_DISABLE_IRQ:
...
			case MULTI_STOP_RUN:
				if (is_active)
					err = msdata->fn(msdata->data);
				break;
...
			}
			ack_state(msdata);
		}
...
		rcu_momentary_dyntick_idle();
	} while (curstate != MULTI_STOP_EXIT);
```

RISC-V ftrace code patching and `stop_machine()`

forcing all other cores into a busy wait loop while a core is performing the critical job.

```
notrace void rcu_momentary_dyntick_idle(void)
{
        int seq;

        raw_cpu_write(rcu_data.rcu_need_heavy_qs, false);
        seq = rcu_dynticks_inc(2);
        /* It is illegal to call this from idle state. */
        WARN_ON_ONCE(!(seq & 0x1));
        rcu_preempt_deferred_qs(current);
}
EXPORT_SYMBOL_GPL(rcu_momentary_dyntick_idle);
```

RISC-V ftrace code patching and `stop_machine()`

forcing all other cores into a busy wait loop while a core is performing the critical job.

```
notrace void rcu_momentary_dyntick_idle(void)
{
        int seq;

        raw_cpu_write(rcu_data.rcu_need_heavy_qs, false);
        seq = rcu_dynticks_inc(2);
        /* It is illegal to call this from idle state. */
        WARN_ON_ONCE(!(seq & 0x1));
        rcu_preempt_deferred_qs(current);
}
EXPORT_SYMBOL_GPL(rcu_momentary_dyntick_idle);
```

```
static void rcu_preempt_deferred_qs(struct task_struct *t)
{
        unsigned long flags;

        if (!rcu_preempt_need_deferred_qs(t))
                return;
        local_irq_save(flags);
        rcu_preempt_deferred_qs_irqrestore(t, flags);
}
```

Mixing with Kernel Preemption

All sub-function calls being made in stop_machine() must be marked as notrace

- Or we may panic the kernel easily:
  - Illegal instructions
  - Corrupted frame
- As of 5.17, we got these symbols that could be called in the idle loop of waiting cores after turning on CONFIG_PREEMPT:
  - `__rcu_report_exp_rnp()`
  - `rcu_report_exp_cpu_mult()`
  - `rcu_preempt_deferred_qs()`
  - `rcu_preempt_need_deferred_qs()`
  - `rcu_preempt_deferred_qs_irqrestore()`

```
REG_S    ra, -SZREG(sp)
auipc    ra, 0x?????
jalr     0x???(ra)
REG_L    ra, -SZREG(sp)
```

Mixing with Kernel Preemption

Even if we got this right, ftrace still messes up, but why?

```
[  222.460512] status: 0000000200000100 badaddr: 00000000000003f1 cause: 000000000000000d
[  222.461026] [<ffffffff8088b41a>] spi_finalize_current_message+0x38/0x24c
[  222.461242] [<ffffffff8088b3ee>] spi_finalize_current_message+0xc/0x24c
[  222.461418] [<ffffffff8088dfba>] __spi_pump_messages+0x2c2/0x6d2
[  222.461584] [<ffffffff8088e658>] __spi_sync+0x260/0x282
[  222.461730] [<ffffffff8088e6f6>] spi_sync_locked+0x20/0x28
[  222.461878] [<ffffffff809867e8>] mmc_spi_readbytes+0x48/0x72
[  222.462029] [<ffffffff80986a3e>] mmc_spi_set_ios+0xc4/0x210
[  222.462176] [<ffffffff80972fd8>] mmc_power_up.part.0+0x110/0x198
[  222.462335] [<ffffffff80973cc2>] mmc_rescan+0x16c/0x2ca
[  222.462479] [<ffffffff8003960a>] process_one_work+0x18a/0x388
[  222.462639] [<ffffffff80039890>] worker_thread+0x88/0x354
[  222.462792] [<ffffffff80040bb6>] kthread+0xe0/0x10a
[  222.462932] [<ffffffff8000376a>] ret_from_exception+0x0/0xc
[  222.464839] ---[ end trace 0000000000000000 ]---
[  222.465084] note: kworker/0:2[43] exited with preempt_count 1
```

Mixing with Kernel Preemption

Even if we got this right, ftrace still messes up due to preemption itself

```
0:REG_S  ra, -SZREG(sp)
4:auipc  ra, 0x?????
8:jalr   0x???(ra)
-------------------------------> preempted
c:REG_L  ra, -SZREG(sp)
```

```
[  222.460512] status: 0000000200000100 badaddr: 00000000000003f1 cause: 000000000000000d
[  222.461026] [<ffffffff8088b41a>] spi_finalize_current_message+0x38/0x24c
[  222.461242] [<ffffffff8088b3ee>] spi_finalize_current_message+0xc/0x24c
[  222.461418] [<ffffffff8088dfba>] __spi_pump_messages+0x2c2/0x6d2
```

Reviews of ftrace Implementations on other Architectures

- Most architectures do not use stop_machine() to perform runtime code patching:
  - x86, x86_64
  - ARM64
  - MIPS
  - powerPC
  - s390
- And they update only one instruction to enable/disable ftrace except for x86
- The key point is to make code-patching seen atomic for running cores
  - Could we?

```
REG_S    ra, -SZREG(sp)
auipc    ra, 0x?????
jalr     0x???(ra)
REG_L    ra, -SZREG(sp)
```

ftrace Code Patching on ARM64

- Concurrent modification and execution of instructions:
  - ARMv7 explicitly state that the effect of the concurrent modification and execution of an instruction is unpredictable except for the following instruction words:
    - `B, BL, NOP, BKPT, SVC, HVC, SMC`
  - In RISC-V, we get Ziccif
- For each function entry:

Compiled	Disabled	Enabled
`NOP` `NOP`	`MOV X9, LR` `NOP`	`MOV X9, LR` `BL <entry>`

ftrace Code Patching on x86

- Things are tricky on x86 due to its variable-length instructions:
  - It uses a 5-byte CALL instruction for each ftrace entry
  - Opcode: 0xE8
  - Immediate: rel32
- Steps to perform runtime code patching, in a nutshell:
  - add an int3 trap to the address that will be patched
  - update all but the first byte of the patched range
  - replace the first byte (int3) by the first byte of replacing opcode

```
NOP, NOP, NOP, NOP, NOP
```

E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.

ftrace Code Patching on x86

- Things are tricky on x86 due to its variable-length instructions:
  - It uses a 5-byte CALL instruction for each ftrace entry
  - Opcode: 0xE8
  - Immediate: rel32
- Steps to perform runtime code patching, in a nutshell:
  - add an int3 trap to the address that will be patched
  - update all but the first byte of the patched range
  - replace the first byte (int3) by the first byte of replacing opcode

```
INT3, NOP, NOP, NOP, NOP
```

E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.

ftrace Code Patching on x86

- Things are tricky on x86 due to its variable-length instructions:
  - It uses a 5-byte CALL instruction for each ftrace entry
  - Opcode: 0xE8
  - Immediate: rel32
- Steps to perform runtime code patching, in a nutshell:
  - add an int3 trap to the address that will be patched
  - update all but the first byte of the patched range
  - replace the first byte (int3) by the first byte of replacing opcode

```
INT3, ftrace_caller
```

E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.

ftrace Code Patching on x86

- Things are tricky on x86 due to its variable-length instructions:
  - It uses a 5-byte CALL instruction for each ftrace entry
  - Opcode: 0xE8
  - Immediate: rel32
- Steps to perform runtime code patching, in a nutshell:
  - add an int3 trap to the address that will be patched
  - update all but the first byte of the patched range
  - replace the first byte (int3) by the first byte of replacing opcode

```
CALL ftrace_caller
```

E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.

How Should We Patch Code Atomically in RISC-V

- RISC-V must use 2 instructions to call a function at a practical distance:
  - AUIPC + JALR, forming a 4GB relative range
  - The relative address must be splitted into these 2 instructions.
- Even if we could make updating of the 2 instructions to be seen at once, we cannot make them execute together.

```
REG_S    ra, -SZREG(sp)
auipc    ra, 0x?????
-----------------------> preempted??
jalr     0x???(ra)
REG_L    ra, -SZREG(sp)
```

Architecture	Branch Range (both directions)
RISC-V	4GB (AUIPC + JALR) 2MB (JAL) 4KB (JALR)
x86	4GB
ARM64	256MB
MIPS	256MB
PowerPC	16MB

Possible solution

- Disable preemption and re-enable preemption on each function entry (x
- Limit the jump offset to 4K and change only JALR instruction (x
- Use LUI (Load Upper Immediate, imm20) to encode the jump (x
- Use trampolines to jump indirectly (?

```
REG_S    ra, -SZREG(sp)
auipc    ra, 0x?????
----------------------> preempted??
jalr     0x???(ra)
REG_L    ra, -SZREG(sp)
```

Architecture	Branch Range (both directions)
RISC-V	4GB (AUIPC + JALR) 2MB (JAL) 4KB (JALR)
x86	4GB
ARM64	256MB
MIPS	256MB
PowerPC	16MB

Proposed Solution

- Similar to the trampoline approach, but we encode the trampoline into each function entry
- 4-byte align, and reserve for 24 bytes at each function (0.56 MB code size increased for 122K funcs)
- Similar to x86, we use control flow redirection.
- We may load/store the jump target atomically.
- Limitation: function alignment

Compiled	Disabled, aligned to 8 bytes	Disabled, aligned to 4 but not 8	Enabled

`nop`
`nop`
`nop`
`nop`
`nop`
`nop`
`func:`
	`00: j func`
	`04: j 0x10`
	`08: dest_addr.lower`
	`0c: dest_addr.upper`
	`10: ld t0,`
	`0x8(t0)`
	`14: jalr t0, t0`
	`func:`
		`04: j func`
		`08: ld t0, 0xc(t0)`
		`0c: j 0x18`
		`10: dest_addr.lower`
		`14: dest_addr.upper`
		`18: jalr t0, t0`
		`func:`
			`auipc t0, 0`
			`...`