统信软件技术有限公司
UnionTech Software Technology Co., Ltd.

BPF赋能CPUIDLE调节器

内核研发部

2025年2月24日

洪奥

目录

CONTENTS

01 CPUIDLE子系统

01 解决什么问题？

02 如何解决？

03 框架

04 GOVERNOR

01 CPUIDLE子系统 解决什么问题？

- 在非计算密集型的系统上，CPU大部分时间处于空闲状态 ， 以我的个人电脑为例，cpu空闲时间到达了81.9%

```
top - 08:55:33 up 11 days, 12:25,  2 users,  load average: 1.22, 1.35, 1.26
任务: 535 total,   2 running, 532 sleeping,   0 stopped,   1 zombie
%Cpu(s): 11.4 us,  5.1 sy,  0.0 ni, 81.9 id,  0.0 wa,  0.0 hi,  0.8 si,  0.0 st
MiB Mem :  31972.1 total,    766.6 free,  18140.8 used,  14917.2 buff/cache
MiB Swap:  16084.0 total,   9403.7 free,   6680.2 used.  13831.3 avail Mem
```

- CPU空闲时会执行idle任务，CPU依然在执行指令，同样会耗电

```
kernel > sched > C idle.c > do_idle(void)
237   static void do_idle(void)
257
258       while (!need_resched()) {
259           rmb();
260
261           local_irq_disable();
262
```

- 能否让CPU在空闲时，不执行指令呢？

01 CPUIDLE子系统 如何解决？

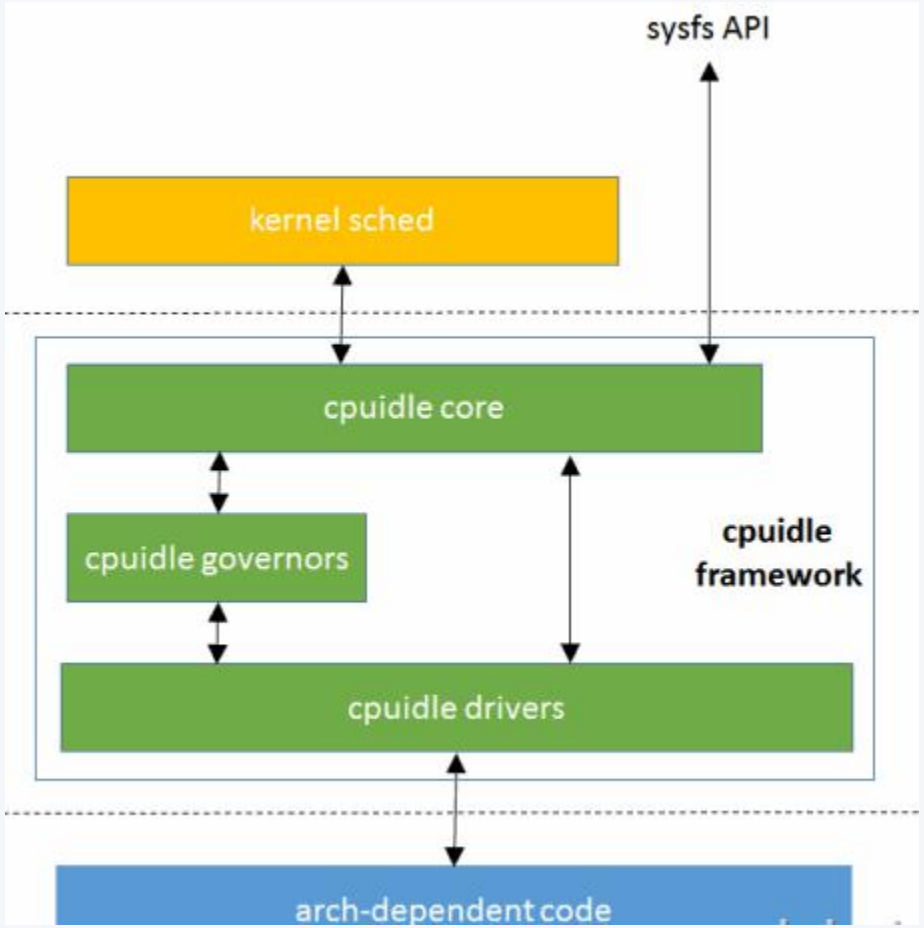
- 硬件层面：CPU的设计者在提供了不同的运行模式C-state, 以intel为例

- 操作系统：通过MWAIT 、 MONITOR 指令让CPU进入特定的C-state

```
arch > x86 > include > asm > C mwait.h > __mwait(unsigned long, unsigned long)
  2   #ifndef _ASM_X86_MWAIT_H
 32                     unsigned long edx)
 37   }
 38
 39   static __always_inline void __monitorx(const void *eax, unsigned long ecx,
 40                                           unsigned long edx)
 41   {
 42       /* "monitorx %eax, %ecx, %edx;" */
 43       asm volatile(".byte 0x0f, 0x01, 0xfa;"
 44                    :: "a" (eax), "c" (ecx), "d"(edx));
 45   }
 46
 47   static __always_inline void __mwait(unsigned long eax, unsigned long ecx)
 48   {
 49       mds_idle_clear_cpu_buffers();
 50
 51       /* "mwait %eax, %ecx;" */
 52       asm volatile(".byte 0x0f, 0x01, 0xc9;"
 53                    :: "a" (eax), "c" (ecx));
 54   }
 55
```

01 CPUIDLE子系统 框架

软件架构



关键元素

- kernel sched：决定何时执行idle任务
- cpuidle core：
  -抽象出cpuidle device、cpuidle driver、cpuidle governor三个实体
  -以函数调用的形式，向上层sched模块提供接口
  -以sysfs的形式，向用户空间提供接口
  -向下层的cpuidle drivers模块，提供统一的driver注册和管理接口
  -向下层的governors模块，提供统一的governor注册和管理接口
- cpuidle governors：
  -决定选用哪种C-state状态（策略）
- cpuidle drivers：
  -架构相关代码，实现进入特定架构CPU的C-state（机制）

代码流程

```
do_idle()->
   cpuidle_idle_call->
      cpuidle_select->
         menu_select
      call_cpuidle->
         cpuidle_enter->
            cpuidle_enter_state->
               intel_idle->
                  mwait_idle_with_hints
```

sysfs 接口

/sys/devices/system/cpu/cpuidle/current_governor

/sys/devices/system/cpu/cpuidle/current_driver

/sys/devices/system/cpu/cpu0/cpuidle/state*/time

01 CPUIDLE子系统 GOVERNOR

governor数据结构：

```
struct cpuidle_governor {
    int (*enable)(struct cpuidle_driver *drv, struct cpuidle_device *dev);
    int (*select)(struct cpuidle_driver *drv, struct cpuidle_device *dev, bool *stop_tick);
    void (*reflect)(struct cpuidle_device *dev, int index);
    ...
};
```

核心函数select：承担着选择最佳C-state的重任。

```
do_idle()->
    cpuidle_idle_call->
        cpuidle_select-> cpuidle_curr_governor->select
            menu_select (governor)
        call_cpuidle->
            cpuidle_enter->
                cpuidle_enter_state->
                    intel_idle-> (driver)
                        mwait_idle_with_hints
```

内核中已有的governor策略：

Ladder
Menu
Teo

迁移select策略到用户态的好处？

- 动态加载
- 深度定制

02 BPF赋能

01 BPF初体验

02 STRUCT_OPS 使用

03 KFUNC使用

04 展望

BPF开发方法

一、使用bpftrace编写eBPF程序

bpftrace -e 'kprobe:do_nanosleep { printf("PID %d sleeping...\n", pid); }'

二、使用libbpf库来开发eBPF程序

- 前置条件：
  -内核编译选项支持CONFIG_DEBUG_INFO_BTF=y
  -依赖安装： sudo apt install clang llvm bpftool build-essential libbpf1 libbpf-dev
- 确认内核支持btf： ls -la /sys/kernel/btf/vmlinux
- 生成vmlinux.h ： bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
- 用户侧编写eBPF程序：（hello.bpf.c）

```
/**
 * 通过使用 kprobe（内核探针）在do_nanosleep函数的入口处放置钩子，实现对该系统调用
的跟踪
*/
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

// 定义一个名为do_nanosleep的 kprobe，当进入do_nanosleep时，它会被触发
SEC("kprobe/do_nanosleep")
int BPF_KPROBE(do_nanosleep)
{
    pid_t pid;

    // 获取当前进程的 PID（进程标识符）
    pid = bpf_get_current_pid_tgid() >> 32;
    // 使用bpf_printk函数在内核日志中打印 PID
    bpf_printk("hello: pid = %d\n", pid);
    return 0;
}
// 定义许可证，以允许程序在内核中运行
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

- 生成ebpf字节码:

clang -g -Wall -D__x86_64__ -O2 -target bpf -I ./vmlinux.h -c hello.bpf.c -o hello.bpf.o

- 生成BPF 脚手架（skeleton）文件：

bpftool gen skeleton hello.bpf.o > hello.skel.h

- 用户侧编写用户空间程序: 用于加载ebpf程序到内核

```
/**
 * ebpf 用户空间程序(loader、read ringbuffer)
*/
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "hello.skel.h"

static int libbpf_print_fn(enum libbpf_print_level level, const char *fo
va_list args)
{
    return vfprintf(stderr, format, args);
}

static volatile sig_atomic_t stop;
static void sig_int(int signo)
{
    stop = 1;
}
```

- 用户程序的加载及运行

gcc hello.c -lbpf -o hello

只能在kprobe放钩子显然不能满足迁移cpuidle select策略到用户侧eBPF程序的需求

bpf struct_ops被引入！
它可以动态地替换内核中任意的 "operations structure"（a structure full of function pointers）。

借助struct_ops 替换任意函数指针的特性，可以将内核的代码逻辑迁移到bpf程序。

02 BPF STRUCT_OPS 特性使用

内核侧

- 在内核中定义新的bpf_struct_ops，cfi_stubs成员赋值为你想替换operations structure

```
static struct bpf_struct_ops new_bpf_struct_ops = {
                .verifier_ops = &new_bpf_struct_verifier_ops,
                .reg = new_bpf_struct_reg,
                .unreg =new_bpf_struct_unreg,
                .update = new_bpf_struct_update,
                .check_member =new_bpf_struct_member,
                .init_member = new_bpf_struct_member,
                .init =new_bpf_struct_nit,
                .validate = new_bpf_struct_validate,
                .name = "new_bpf_struct_ops",
                .cfi_stubs = &struct_ops_to_be_replace,
                .owner = THIS_MODULE,
};
```

- 迁移任意内核策略到bpf程序时， struct_ops_to_be_replace的选取？

BPF tcpca的做法：实例网络框架中已有的结构体tcp_congestion_ops做为 struct_ops_to_be_replace

```
static struct tcp_congestion_ops __bpf_ops_tcp_congestion_ops = {
    .ssthresh = bpf_tcp_ca_ssthresh,
    .cong_avoid = bpf_tcp_ca_cong_avoid,
    .set_state = bpf_tcp_ca_set_state,
    .cwnd_event = bpf_tcp_ca_cwnd_event,
    .in_ack_event = bpf_tcp_ca_in_ack_event,
    .pkts_acked = bpf_tcp_ca_pkts_acked,
    .min_tso_segs = bpf_tcp_ca_min_tso_segs,
    .cong_control = bpf_tcp_ca_cong_control,
    .undo_cwnd = bpf_tcp_ca_undo_cwnd,
    .sndbuf_expand = bpf_tcp_ca_sndbuf_expand,

    .init = __bpf_tcp_ca_init,
    .release = __bpf_tcp_ca_release,
};

static struct bpf_struct_ops bpf_tcp_congestion_ops = {
    .verifier_ops = &bpf_tcp_ca_verifier_ops,
    .reg = bpf_tcp_ca_reg,
    .unreg = bpf_tcp_ca_unreg,
    .update = bpf_tcp_ca_update,
    .check_member = bpf_tcp_ca_check_member,
    .init_member = bpf_tcp_ca_init_member,
    .init = bpf_tcp_ca_init,
    .validate = bpf_tcp_ca_validate,
    .name = "tcp_congestion_ops",
    .cfi_stubs = &__bpf_ops_tcp_congestion_ops,
    .owner = THIS_MODULE,
};
```

BPF scheduler的做法：定义一个与调度类相似的结构体sched_ext_ops做为 struct_ops_to_be_replace，
并且新增ext调度类做中间层，在调度类ext的回调函数中调用sched_ext_ops的回调函数。

```
struct sched_ext_ops {
    /**...
    s32 (*select_cpu)(struct task_struct *p, s32 prev_cpu, u64 wake_flags);

    /**...
    void (*enqueue)(struct task_struct *p, u64 enq_flags);

    /**...
    void (*dequeue)(struct task_struct *p, u64 deq_flags);

    /**...
    void (*dispatch)(s32 cpu, struct task_struct *prev);
```

```
static struct sched_ext_ops __bpf_ops_sched_ext_ops = {...

static struct bpf_struct_ops bpf_sched_ext_ops = {
    .verifier_ops = &bpf_scx_verifier_ops,
    .reg = bpf_scx_reg,
    .unreg = bpf_scx_unreg,
    .check_member = bpf_scx_check_member,
    .init_member = bpf_scx_init_member,
    .init = bpf_scx_init,
    .update = bpf_scx_update,
    .validate = bpf_scx_validate,
    .name = "sched_ext_ops",
    .owner = THIS_MODULE,
    .cfi_stubs = &__bpf_ops_sched_ext_ops
};
```

02 BPF STRUCT_OPS 特性使用

内核侧

- 迁移cpuidle governor策略到bpf程序时， struct_ops_to_be_replace的选取？

方案一：
参考BPF tcpca，实例化cpuilde框架中已有的结构体cpuidle_governor做struct_ops_to_be_replace

```
static struct cpuidle_governor ext_cpuidel_governor;

static struct bpf_struct_ops bpf_idle_gover_ext_ops = {
        ...
       .cfi_stubs = &ext_cpuidel_governor,
        ...
}
```

缺点：需修改cpuilde的框架代码做兼容处理

```
do_idle()->
   cpuidle_idle_call->
      cpuidle_select->
            if( ext_cpuidel_governor_exsit)
                do_ext_cpuidel_governor();
            else
                do_other_governor();
      call_cpuidle->
         cpuidle_enter->
            cpuidle_enter_state->
               intel_idle-> (driver)
                  mwait_idle_with_hints
```

✗

方案二：
参考BPF scheduler，定义一个与cpuilde_governor相似的结构体__bpf_ops_idle_gover_ext_ops做struct_ops_to_be_replace

```
struct idle_gover_ext_ops{
   void (*ops_select)(void)
}
static struct idle_gover_ext_ops __bpf_ops_idle_gover_ext_ops = {
      .ops_select = ops_select_stub,
}
static struct bpf_struct_ops bpf_idle_gover_ext_ops = {
       ...
      .cfi_stubs = &__bpf_ops_idle_gover_ext_ops
};
```

新增ext governor类

```
static struct cpuidle_governor ext_governor = {
      .select =      ext_select,
      ...
}
```

在ext governor的select回调函数中调用idle_gover_ext_ops的回调

```
static int ext_select(struct cpuidle_driver *drv, struct cpuidle_device *dev,
                      bool *stop_tick)
{
      __bpf_ops_idle_gover_ext_ops.ops_select();
      ...
}
```

✓

02 BPF STRUCT_OPS 特性使用

内核侧

- struct_ops注册

  在新增的cpuidle_governor ext_governor类初始化时注册

```
static int __init init_ext(void)
{
    ...
    register_bpf_struct_ops(&bpf_idle_gover_ext_ops, idle_gover_ext_ops);
    return cpuidle_register_governor(&ext_governor);
}
```

- struct_ops的替换

  在.reg回调中执行 eBPF struct_ops替换内核struct ops

```
static int bpf_igx_reg(void *kdata)
{
    printk("%s called\n", __func__);
    __bpf_ops_idle_gover_ext_ops = *(struct idle_gover_ext_ops *)kdata;
    return 0;
}
```

- struct_ops的还原

  在.unreg回调中 还原struct_ops，在用户态终止eBPF程序时触发

```
static void bpf_igx_unreg(void *kdata)
{
    __bpf_ops_idle_gover_ext_ops.ops_select = ops_select_stub;
    return;
}
```

```
static struct cpuidle_governor ext_governor = {
    .name =     "ext",
    .rating =   20,
    .enable =   ext_enable_device,
    .select =   ext_select,
    .reflect =  ext_reflect,
};
```

```
static struct idle_gover_ext_ops __bpf_ops_idle_gover_ext_ops = {
    .ops_select = ops_select_stub,
}

static struct bpf_struct_ops bpf_idle_gover_ext_ops = {
    ...
    .reg = bpf_igx_reg,
    .unreg = bpf_igx_unreg,
    .cfi_stubs = &__bpf_ops_idle_gover_ext_ops,
    ...
}
```

02 BPF STRUCT_OPS 特性使用

用户侧

- eBPF程序

```
SEC("struct_ops/simple_dummy")
void BPF_PROG(simple_dummy)
{
  bpf_printk("simple dummy ENTRY \n");
  return;
}

SEC(".struct_ops")
struct idle_gover_ext_ops simple = {
      .ops_select = (void *)simple_dummy,
};
```

代码解析：
- SEC(".struct_ops")
用于 BPF 中要替换的 struct_ops 结构，
例如当前实现的 idle_gover_ext_ops simple。
- SEC("struct_ops/xyz")
定义待替换结构体
idle_gover_ext_ops simple的回调函数

- 用户程序

```
int main(){
      struct simple_igx_bpf *skel;
      struct bpf_link *link;

      /* 脚手架生成的函数*/
      /* 加载并验证 ebpf 应用程序 */
      skel = simple_igx_bpf__open_and_load();

     // bpf_map__attach_struct_ops 增加注册一个 struct_ops map 到内核子系统
     link =bpf_map__attach_struct_ops(skel->maps.simple}

      while (!stop) {
            fprintf(stderr, ".");
            sleep(1);
      }

cleanup:
    //清理函数
    bpf_link__destroy(link);
     simple_igx_bpf__destroy(skel);
}
```

借助 BPF struct_ops ， 我们已经可以将cpuidle select策略迁移到用户侧，但依然有局限性：
- 功能差距： eBPF 运行时的现有功能/辅助函数无法提供你所需的特定能力。
- 复杂需求： 某些任务需要更复杂的内核交互，而 eBPF 无法开箱即用地处理这些需求。

Kfunc（BPF 内核函数）被引入！
通过在内核中定义你自己的 kfunc，可以将 eBPF 的能力扩展到默认限制之外:
- 增强功能： 引入标准 eBPF 运行时中不可用的新操作。
- 定制行为： 根据你的特定需求定制内核交互。

02 BPF KFUNC 特性使用

内核侧

```
/* 开始 kfunc 定义 */
__bpf_kfunc_start_defs();

/* 定义 kfunc  my_bpf_kfunc*/
__bpf_kfunc int my_bpf_kfunc(int val)
{
	do_something();
	trace_printk("%s %d\n", __func__, val);
	return 0;
}

/* 结束 kfunc 定义 */
__bpf_kfunc_end_defs();

/* 定义 BTF kfuncs ID 集 */
BTF_KFUNCS_START(my_bpf_kfunc_example_ids_set)
BTF_ID_FLAGS(func, my_bpf_kfunc)
BTF_KFUNCS_END(my_bpf_kfunc_example_ids_set)

static const struct btf_kfunc_id_set my_bpf_kfunc_example_set = {
	.owner = THIS_MODULE,
	.set = &my_bpf_kfunc_example_ids_set,
};

 /**
  * init_ext - initializes the governor
  */
 static int __init init_ext(void)
 {
	 int ret;
	/* 注册 BPF_PROG_TYPE_KPROBE 的 BTF kfunc ID 集 */
	ret = register_btf_kfunc_id_set(BPF_PROG_TYPE_STRUCT_OPS,
&my_bpf_kfunc_example_set);
}
```

用户侧

- eBPF程序

```
/* 声明外部 kfunc */
extern int my_bpf_kfunc(int val) __ksym;

SEC("struct_ops/simple_dummy")
int BPF_PROG(simple_dummy)
{
	int ret;
	bpf_printk("simple dummy ENTRY \n");
	ret = my_bpf_kfunc(123456);
	return ret;
}
```

02 BPF KFUNC 特性使用

一个更有趣的功能：
通过ebpf程序的灵活控制，让指定的cpu在选择C-state时强制进入最深C-state

内核侧

```c
/* 定义 my_bpf_kfunc kfunc */
__bpf_kfunc int my_bpf_kfunc(int val, struct cpuidle_driver *drv, struct cpuidle_device *dev)
{
    if(dev->cpu <= val )
        return drv->state_count - 1;
    else
        trace_printk("%s: val = %d\n", __func__, val);


    return 0;
}
```

用户侧

- eBPF程序

```c
SEC("struct_ops/simple_dummy")
int BPF_PROG(simple_dummy, struct cpuidle_driver *drv, struc
{
        int ret;
        bpf_printk("simple dummy ENTRY \n");
        ret = my_bpf_kfunc(4, drv, dev);
        return ret;
}
```

- 用户程序

…

- BPF赋能cpufreq governor
- BPF赋能io schdule
- BPF赋能gpu schdule

...

Q & A