前有schedutil governor来调节cpu频率，现有schedfreq governor来调节频率，现在来讲解它的来龙去脉。
源代码参考AOSP kernel的这个分支：remotes/origin/android-msm-wahoo-4.4-pie，kernel version：4.4.116。源代码下载地址：https://aosp.tuna.tsinghua.edu.cn/kernel/msm.git
使用我的分析的kernel code方式如下：

- git clone https://aosp.tuna.tsinghua.edu.cn/kernel/msm.git
- git branch -a 查看有什么分支

```
remotes/origin/android-msm-sturgeon-3.10-n-preview-1-wear-release
remotes/origin/android-msm-sturgeon-3.10-n-preview-2-wear-release
remotes/origin/android-msm-sturgeon-3.10-n-preview-3-wear-release
remotes/origin/android-msm-sturgeon-3.10-n-preview-4-wear-release
remotes/origin/android-msm-sturgeon-3.10-nougat-dr1-wear
remotes/origin/android-msm-sturgeon-3.10-nougat-mr1-wear-release
remotes/origin/android-msm-sundial-3.18-nougat-mr1-wear-release
remotes/origin/android-msm-sundial-3.18-oreo-wear-dr
remotes/origin/android-msm-swift-3.18-marshmallow-mr1-wear-release
remotes/origin/android-msm-swift-3.18-nougat-dr-release
remotes/origin/android-msm-swordfish-3.18-nougat-mr1-wear-release
remotes/origin/android-msm-swordfish-3.18-nougat-wear-release
remotes/origin/android-msm-swordfish-3.18-o-wear-preview-4
remotes/origin/android-msm-swordfish-3.18-oreo-wear-dr
remotes/origin/android-msm-vega-4.4-fsi-oreo
remotes/origin/android-msm-vega-4.4-oreo-daydream
remotes/origin/android-msm-wahoo-2018.07-oreo-m2
remotes/origin/android-msm-wahoo-2018.07-oreo-m4
remotes/origin/android-msm-wahoo-4.4-o-mr1-preview1
remotes/origin/android-msm-wahoo-4.4-oreo-dr1
remotes/origin/android-msm-wahoo-4.4-oreo-m2
remotes/origin/android-msm-wahoo-4.4-oreo-m4
remotes/origin/android-msm-wahoo-4.4-oreo-mr1
remotes/origin/android-msm-wahoo-4.4-oreo-mr1-preview2
remotes/origin/android-msm-wahoo-4.4-p-preview-1
remotes/origin/android-msm-wahoo-4.4-p-preview-2
remotes/origin/android-msm-wahoo-4.4-p-preview-3
remotes/origin/android-msm-wahoo-4.4-p-preview-4
remotes/origin/android-msm-wahoo-4.4-p-preview-5
remotes/origin/android-msm-wahoo-4.4-pie
remotes/origin/android-msm-wren-3.10-marshmallow-dr1-wear-release
remotes/origin/android-msm-wren-3.10-marshmallow-mr1-wear-release
remotes/origin/master
```

- git checkout -b remotes/origin/android-msm-wahoo-4.4-pie
- 正确切换到我所看的kernel 分支，Android 9.0

闲话少扯。。。。
按照老样子，先看governor相关的结构体成员变量：

```
/*降频和升频的最小间隔，可以修改*/
#define THROTTLE_DOWN_NSEC  50000000 /* 50ms default */
#define THROTTLE_UP_NSEC    500000 /* 500us default */

static DEFINE_PER_CPU(unsigned long, enabled);
DEFINE_PER_CPU(struct sched_capacity_reqs, cpu_sched_capacity_reqs);
```

```
/*tunable的参数，对于升频和降频的时间限制，attr_set设置sys接口，可供userspace调节*/
struct gov_tunables {
    struct gov_attr_set attr_set;
    unsigned int up_throttle_nsec;
    unsigned int down_throttle_nsec;
};

/**
 * gov_data - per-policy data internal to the governor
 * @up_throttle: next throttling period expiry if increasing OPP
 * @down_throttle: next throttling period expiry if decreasing OPP
 * @up_throttle_nsec: throttle period length in nanoseconds if increasing
OPP
 * @down_throttle_nsec: throttle period length in nanoseconds if decreasing
OPP
 * @task: worker thread for dvfs transition that may block/sleep
 * @irq_work: callback used to wake up worker thread
 * @requested_freq: last frequency requested by the sched governor
 *
 * struct gov_data is the per-policy cpufreq_sched-specific data structure.
A
 * per-policy instance of it is created when the cpufreq_sched governor
receives
 * the CPUFREQ_GOV_START condition and a pointer to it exists in the
gov_data
 * member of struct cpufreq_policy.
 *
 * Readers of this data must call down_read(policy->rwsem). Writers must
 * call down_write(policy->rwsem).
 */
struct gov_data {
    ktime_t up_throttle; /*升频的时间节点*/
    /*降频的时间节点，是当前时间+门限数值，即为下次降频的最近时间节点*/
    ktime_t down_throttle;
    /*tunable参数*/
    struct gov_tunables *tunables;
    struct list_head tunables_hook;
    /*频率修改的进程*/
    struct task_struct *task;
    /*slow adjust freq的worker*/
    struct irq_work irq_work;
    /*更加capacity数值设定的请求频率，在通过频率table挑选idx并获取对于
     freq_table的freq_value,最后通过dvfs进行频率的调整*/
    unsigned int requested_freq;
};
```

接着看cpufreq_governor结构体的填充。

```
static int cpufreq_sched_setup(struct cpufreq_policy *policy,
                               unsigned int event)
{
    switch (event) {
    case CPUFREQ_GOV_POLICY_INIT:
        return cpufreq_sched_policy_init(policy);
    case CPUFREQ_GOV_POLICY_EXIT:
        return cpufreq_sched_policy_exit(policy);
```

```
    case CPUFREQ_GOV_START:
        return cpufreq_sched_start(policy);
    case CPUFREQ_GOV_STOP:
        return cpufreq_sched_stop(policy);
    case CPUFREQ_GOV_LIMITS:
        cpufreq_sched_limits(policy);
        break;
    }
    return 0;
}


#ifndef CONFIG_CPU_FREQ_DEFAULT_GOV_SCHED
static
#endif
struct cpufreq_governor cpufreq_gov_sched = {
    .name            = "sched",
    .governor        = cpufreq_sched_setup,
    .owner           = THIS_MODULE,
};

static int __init cpufreq_sched_init(void)
{
    int cpu;

    for_each_cpu(cpu, cpu_possible_mask)
        per_cpu(enabled, cpu) = 0;
    return cpufreq_register_governor(&cpufreq_gov_sched);
}

/* Try to make this the default governor */
fs_initcall(cpufreq_sched_init);
```

可以看到governor名字为“sched”，顾明思议就是更加调度器的某些变量来调节cpu频率的。我们看起governor callback函数的init：

```
static int cpufreq_sched_policy_init(struct cpufreq_policy *policy)
{
    struct gov_data *gd;
    int cpu;
    int rc;
    /*对每个cpu上的cpu_sched_capacity结构体进行初始化为0*/
    for_each_cpu(cpu, policy->cpus)
        memset(&per_cpu(cpu_sched_capacity_reqs, cpu), 0,
               sizeof(struct sched_capacity_reqs));
    /*为sched governor data分配空间*/
    gd = kzalloc(sizeof(*gd), GFP_KERNEL);
    if (!gd)
        return -ENOMEM;
    /*将sched governor data挂载到cpu policy governor data上，即关联上*/
    policy->governor_data = gd;

    if (!global_tunables) {
        /*对tunable结构体变量分配空间*/
        gd->tunables = kzalloc(sizeof(*gd->tunables), GFP_KERNEL);
        if (!gd->tunables)
```

```
            goto free_gd;
        /*设置频率升高的时间限制，也就是升频率间隔不能小于这个间隔*/
        gd->tunables->up_throttle_nsec =
            policy->cpuinfo.transition_latency ?
            policy->cpuinfo.transition_latency :
            THROTTLE_UP_NSEC;
       /*设置频率降低的时间限制*/
        gd->tunables->down_throttle_nsec =
            THROTTLE_DOWN_NSEC;
       /*初始化tunable结构体成员变量的kobject，并产生sys fs*/
        rc = kobject_init_and_add(&gd->tunables->attr_set.kobj,
                          &tunables_ktype,
                          get_governor_parent_kobj(policy),
                          "%s", cpufreq_gov_sched.name);
        if (rc)
            goto free_tunables;
       /*属性设置*/
        gov_attr_set_init(&gd->tunables->attr_set,
                    &gd->tunables_hook);

        pr_debug("%s: throttle_threshold = %u [ns]\n",
              __func__, gd->tunables->up_throttle_nsec);

        if (!have_governor_per_policy())
            global_tunables = gd->tunables;
    } else {
        gd->tunables = global_tunables;
        gov_attr_set_get(&global_tunables->attr_set,
                   &gd->tunables_hook);
    }
    /*再次update，上面那个是否有点多余哈？？*/
    policy->governor_data = gd;
    if (cpufreq_driver_is_slow()) {
        cpufreq_driver_slow = true;
        /*cpufreq_driver_slow这个参数有点意思。下面创建thread，wakeup函数为：
        cpufreq_sched_thread，最后会创建kschedfreq:0和kschedfreq:4。
        对于两个cluster的cpu架构*/
        gd->task = kthread_create(cpufreq_sched_thread, policy,
                          "kschedfreq:%d",
                          cpumask_first(policy->related_cpus));
        if (IS_ERR_OR_NULL(gd->task)) {
            pr_err("%s: failed to create kschedfreq thread\n",
                  __func__);
            goto free_tunables;
        }
        get_task_struct(gd->task);
        /*绑定相关联的cpu*/
        kthread_bind_mask(gd->task, policy->related_cpus);
        wake_up_process(gd->task);
        /*初始化irq_work*/
        init_irq_work(&gd->irq_work, cpufreq_sched_irq_work);
    }

    set_sched_freq();
```

```c

    return 0;

free_tunables:
    kfree(gd->tunables);
free_gd:
    policy->governor_data = NULL;
    kfree(gd);
    return -ENOMEM;
}
```

我们能够看到，上面最重要的信息如下：

- 频率升高的时间限制
- 频率下降的时间限制
- 创建的thread，cpufreq_sched_thread，频率调节的进程
- 初始化一个irq_work，callback函数为cpufreq_sched_irq_work，最后还是wakeupgd->task，callback cpufreq_sched_thread这个函数。

接下来看一下，cpufreq_sched_thread这个函数的实现过程：

```c
/*
 * we pass in struct cpufreq_policy. This is safe because changing out the
 * policy requires a call to __cpufreq_governor(policy, CPUFREQ_GOV_STOP),
 * which tears down all of the data structures and
__cpufreq_governor(policy,
 * CPUFREQ_GOV_START) will do a full rebuild, including this kthread with
the
 * new policy pointer
 */
static int cpufreq_sched_thread(void *data)
{
    struct sched_param param;
    struct cpufreq_policy *policy;
    struct gov_data *gd;
    unsigned int new_request = 0;
    unsigned int last_request = 0;
    int ret;
    /*获取当前cpufreq_policy*/
    policy = (struct cpufreq_policy *) data;
    /*获取sched governor data*/
    gd = policy->governor_data;

    param.sched_priority = 50;
    ret = sched_setscheduler_nocheck(gd->task, SCHED_FIFO, ¶m);
    if (ret) {
        pr_warn("%s: failed to set SCHED_FIFO\n", __func__);
        do_exit(-EINVAL);
    } else {
        pr_debug("%s: kthread (%d) set to SCHED_FIFO\n",
                __func__, gd->task->pid);
    }

    do {
        /*governor请求的频率*/
        new_request = gd->requested_freq;
        /*如果频率一致，则进程进入TASK_INTERRUPTIBLE状态并sleep*/
```

```c
        if (new_request == last_request) {
            set_current_state(TASK_INTERRUPTIBLE);
            if (kthread_should_stop())
                break;
            schedule();  /*放弃cpu的运行, sleep*/
        } else {
            /*
             * if the frequency thread sleeps while waiting to be
             * unthrottled, start over to check for a newer request
             */
            /*是否最后启动频率请求, */
            if (finish_last_request(gd, policy->cur))
                continue;
            last_request = new_request;
            /*update 升和降频率的时间=当前时间+升/降频率时间间隔, 同时调用DVFS
            进行频率的调节*/
            cpufreq_sched_try_driver_target(policy, new_request);
        }
    } while (!kthread_should_stop());

    return 0;
}
```

我们来看一下finish_last_request函数的实现：

```c
static bool finish_last_request(struct gov_data *gd, unsigned int cur_freq)
{
    ktime_t now = ktime_get();

    ktime_t throttle = gd->requested_freq < cur_freq ?
        gd->down_throttle : gd->up_throttle;
    /*当前时间与下个周期频率变化的时间, 如果现在时间已经超过本应该频率调节的时间
    节点, 那么就必须进行频率调节了*/
    if (ktime_after(now, throttle))
        return false;

    while (1) {
        /*由于throtle > now*/
        int usec_left = ktime_to_ns(ktime_sub(throttle, now));
         /*将差值转换为us*/
        usec_left /= NSEC_PER_USEC;
        trace_cpufreq_sched_throttled(usec_left);
        /*休眠[usec_left,usec_left+100]这个时间间隔*/
        usleep_range(usec_left, usec_left + 100);
        now = ktime_get();   /*当前时间*/
        /*再次比较当前时间与throttle的差值, 如果now + sleep 时间 > throttle,
         则在thread里面重新计算是否需要频率调整。即休眠[usec_left,usec_left+100]
          这个时间段之后, 判断finish_last_update为false。。     */
        if (ktime_after(now, throttle))
            return true;
    }
}
```

如果没有外部触发这个thread, 最后last_request = new_request一直会相等, 导致频率不会update, 那么肯定还有调度算法来trigger governor来调节频率：

```c
void update_cpu_capacity_request(int cpu, bool request)
```

```
{
    unsigned long new_capacity;
    struct sched_capacity_reqs *scr;

    /* The rq lock serializes access to the CPU's sched_capacity_reqs. */
    lockdep_assert_held(&cpu_rq(cpu)->lock);
    /*获取当前cpu的sche_capacity_reqs数据结果数据*/
    scr = &per_cpu(cpu_sched_capacity_reqs, cpu);
    /*capacity受cfs和rt task的影响*/
    new_capacity = scr->cfs + scr->rt;
    /*将capacity增加10%*/
    new_capacity = new_capacity * capacity_margin
        / SCHED_CAPACITY_SCALE;
    new_capacity += scr->dl;
    /*如果当前cpu的需要调整的capacity与源total一致，没有必要频率调节了*/
    if (new_capacity == scr->total)
        return;

    trace_cpufreq_sched_update_capacity(cpu, request, scr, new_capacity);
    /*更新源total数据为最新数据*/
    scr->total = new_capacity;
    /*如果需要频率调整，则调用下面函数进行调整*/
    if (request)
        update_fdomain_capacity_request(cpu);
}
```

看看update_fdomain_capacity_request函数的实现原理（比较简单）：

```
static void update_fdomain_capacity_request(int cpu)
{
    unsigned int freq_new, index_new, cpu_tmp;
    struct cpufreq_policy *policy;
    struct gov_data *gd;
    unsigned long capacity = 0;

    /*
     * Avoid grabbing the policy if possible. A test is still
     * required after locking the CPU's policy to avoid racing
     * with the governor changing.
     *//*是否启用sched governor*/
    if (!per_cpu(enabled, cpu))
        return;
    /*获取当前cpu的cpufreq_policy*/
    policy = cpufreq_cpu_get(cpu);
    if (IS_ERR_OR_NULL(policy))
        return;

    if (policy->governor != &cpufreq_gov_sched ||
        !policy->governor_data)
        goto out;
    /*获取当前sched governor data结构体*/
    gd = policy->governor_data;
    /*对同一个policy的cpu找出最大的capacity数值*/
    /* find max capacity requested by cpus in this policy */
    for_each_cpu(cpu_tmp, policy->cpus) {
        struct sched_capacity_reqs *scr;
```

```
        scr = &per_cpu(cpu_sched_capacity_reqs, cpu_tmp);
        capacity = max(capacity, scr->total);
    }
    /*使用capacity容量归一化最高频率*/
    /* Convert the new maximum capacity request into a cpu frequency */
    freq_new = capacity * policy->max >> SCHED_CAPACITY_SHIFT;
    /*找出符合freq_new频率的index_new索引，即频率table索引号*/
    if (cpufreq_frequency_table_target(policy, policy->freq_table,
                   freq_new, CPUFREQ_RELATION_L,
                   &index_new))
        goto out;
    /*索引号在table里面对应的频率*/
    freq_new = policy->freq_table[index_new].frequency;
    /*频率校正*/
    if (freq_new > policy->max)
        freq_new = policy->max;

    if (freq_new < policy->min)
        freq_new = policy->min;

    trace_cpufreq_sched_request_opp(cpu, capacity, freq_new,
                gd->requested_freq);
    if (freq_new == gd->requested_freq)
        goto out;
    /*更新governor data结构体请求的频率参数*/
    gd->requested_freq = freq_new;

    /*
     * Throttling is not yet supported on platforms with fast cpufreq
     * drivers.
     */
    if (cpufreq_driver_slow)
        /*触发频率调节*/
        irq_work_queue_on(&gd->irq_work, cpu);
    else
        cpufreq_sched_try_driver_target(policy, freq_new);

out:
    cpufreq_cpu_put(policy);
}
```

最后都是执行到如下函数中：

```
static void cpufreq_sched_try_driver_target(struct cpufreq_policy *policy,
                     unsigned int freq)
{
    struct gov_data *gd = policy->governor_data;

    /* avoid race with cpufreq_sched_stop */
    if (!down_write_trylock(&policy->rwsem))
        return;
   /*频率更新*/
    __cpufreq_driver_target(policy, freq, CPUFREQ_RELATION_L);
    /*更新上升和下降频率的下个频率调节时间节点*/
    gd->up_throttle = ktime_add_ns(ktime_get(),
```

```
                        gd->tunables->up_throttle_nsec);
        gd->down_throttle = ktime_add_ns(ktime_get(),
                        gd->tunables->down_throttle_nsec);
        up_write(&policy->rwsem);
    }
```
至此完毕，比较简单。关键还是这个函数update_cpu_capacity_request在哪里调用的。调用关系方框图如下：