我们经常看到很多driver会或者cpufreq governor等等会创建一些进程,并在创建之后使用wake_up_process(struct task_struct *p)函数直接wakeup这个task,下面看看这个wake_up_process函数是怎么实现进程调度,怎么实现放在哪个CPU上执行调度的?
先列出起源码如下:

```
wake_up_process(p)---->try_to_wake_up(p,TASK_NORMAL,0,1)---->

/**
 * wake_up_process - Wake up a specific process
 * @p: The process to be woken up.
 *
 * Attempt to wake up the nominated process and move it to the set of
runnable
 * processes.
 *
 * Return: 1 if the process was woken up, 0 if it was already running.
 *
 * It may be assumed that this function implies a write memory barrier
before
 * changing the task state if and only if any tasks are woken up.
 */
int wake_up_process(struct task_struct *p)
{   /*wake_up_process直接调用try_to_wake_up函数,并添加三个限定参数*/
    return try_to_wake_up(p, TASK_NORMAL, 0, 1);
}
/* Convenience macros for the sake of wake_up */
#define TASK_NORMAL     (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)

/**
 * try_to_wake_up - wake up a thread
 * @p: the thread to be awakened
 * @state: the mask of task states that can be woken
 * @wake_flags: wake modifier flags (WF_*)
 * @sibling_count_hint: A hint at the number of threads that are being woken
up
 *                      in this event.
 *
 * Put it on the run-queue if it's not already there. The "current"
 * thread is always on the run-queue (except when the actual
 * re-schedule is in progress), and as such you're allowed to do
 * the simpler "current->state = TASK_RUNNING" to mark yourself
 * runnable without the overhead of this.
 *
 * Return: %true if @p was woken up, %false if it was already running.
 * or @state didn't match @p's state.
 */
static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags,
           int sibling_count_hint)
{
    unsigned long flags;
    int cpu, success = 0;
#ifdef CONFIG_SMP
    struct rq *rq;
    u64 wallclock;
```

```
#endif

    /*
     * If we are going to wake up a thread waiting for CONDITION we
     * need to ensure that CONDITION=1 done by the caller can not be
     * reordered with p->state check below. This pairs with mb() in
     * set_current_state() the waiting thread does.
     */  /*很有可能需要唤醒一个thread的函数,某个条件必须成立,为了取到最新的没有没优化
   的条件数值,使用内存屏障来实现.*/
    smp_mb__before_spinlock();
    raw_spin_lock_irqsave(&p->pi_lock, flags);
    /*如果进程不是在:TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE下,则就不是normal
  task,直接退出wakeup流程.所以在内核里面看到的wake_up_process,可以看到起主函数都会
   将进程设置为TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE这两种状态之一*/
    if (!(p->state & state))
        goto out;

    trace_sched_waking(p);

    success = 1; /* we're going to change ->state */
    /*获取这个进程当前处在的cpu上面,并不是时候进程就在这个cpu上运行,后面会挑选cpu*/
    cpu = task_cpu(p);

    /*
     * Ensure we load p->on_rq _after_ p->state, otherwise it would
     * be possible to, falsely, observe p->on_rq == 0 and get stuck
     * in smp_cond_load_acquire() below.
     *
     * sched_ttwu_pending()                 try_to_wake_up()
     *   [S] p->on_rq = 1;                  [L] P->state
     *       UNLOCK rq->lock  -----.
     *                              \
     *               +---   RMB
     * schedule()                   /
     *       LOCK rq->lock    -----'
     *       UNLOCK rq->lock
     *
     * [task p]
     *   [S] p->state = UNINTERRUPTIBLE     [L] p->on_rq
     *
     * Pairs with the UNLOCK+LOCK on rq->lock from the
     * last wakeup of our task and the schedule that got our task
     * current.
     */
    smp_rmb();
   /*使用内存屏障保证p->on_rq的数值是最新的.如果task已经在rq里面,即进程已经处于
    runnable/running状态.ttwu_remote目的是由于task p已经在rq里面了,并且并没有完全
    取消调度,再次会wakeup的话,需要将task的状态翻转,将状态设置为TASK_RUNNING,这样
    task就一直在rq里面运行了.这种情况直接退出下面的流程,并对调度状态/数据进行统计*/
    if (p->on_rq && ttwu_remote(p, wake_flags))
        goto stat;

#ifdef CONFIG_SMP
    /*
```

```
         * Ensure we load p->on_cpu _after_ p->on_rq, otherwise it would be
         * possible to, falsely, observe p->on_cpu == 0.
         *
         * One must be running (->on_cpu == 1) in order to remove oneself
         * from the runqueue.
         *
         *  [S] ->on_cpu = 1;     [L] ->on_rq
         *      UNLOCK rq->lock
         *          RMB
         *      LOCK   rq->lock
         *  [S] ->on_rq = 0;     [L] ->on_cpu
         *
         * Pairs with the full barrier implied in the UNLOCK+LOCK on rq->lock
         * from the consecutive calls to schedule(); the first switching to our
         * task, the second putting it to sleep.
         */
        smp_rmb();

        /*
         * If the owning (remote) cpu is still in the middle of schedule() with
         * this task as prev, wait until its done referencing the task.
         */
         /*如果拥有（远程）cpu仍然在schedule（）的中间，并且此任务为prev，请等待
         其完成引用任务,意思是说,task p是作为其他cpu上的调度实体被调度,并且没有调度完毕,需要
         等待其完毕,加内屏屏障就是保证on_cpu的数值是内存里面最新的数据
         */
        while (p->on_cpu)
            cpu_relax();/*cpu 忙等待,期望有人修改p->on_cpu的数值,退出忙等待*/

        /*
         * Combined with the control dependency above, we have an effective
         * smp_load_acquire() without the need for full barriers.
         *
         * Pairs with the smp_store_release() in finish_lock_switch().
         *
         * This ensures that tasks getting woken will be fully ordered against
         * their previous state and preserve Program Order.
         */
        smp_rmb();
        /*获取当前进程所在的cpu的rq*/
        rq = cpu_rq(task_cpu(p));

        raw_spin_lock(&rq->lock);
        /*获取当前时间作为walt的wallclock*/
        wallclock = walt_ktime_clock();
        /*更新rq->curr在当前时间对进程负载/对累加的runnable_time的影响*/
        walt_update_task_ravg(rq->curr, rq, TASK_UPDATE, wallclock, 0);
       /*更新新创建的进程p在当前时间对进程负载/对累加的runnable_time的影响*/
        walt_update_task_ravg(p, rq, TASK_WAKE, wallclock, 0);
        raw_spin_unlock(&rq->lock);
        /*根据进程p的状态来确定,如果调度器调度这个task是否会对load有贡献.*/
        p->sched_contributes_to_load = !!task_contributes_to_load(p);
        /*设置进程状态为TASK_WAKING*/
        p->state = TASK_WAKING;
```

```
    /*根据进程的所属的调度类调用相关的callback函数,这里进程会减去当前所处的cfs_rq的最小vrunime,因为这里还没有确定进程会在哪个cpu上运行,等确定之后,会在入队的时候加上新cpu的cfs_rq的最小vruntime*/
    if (p->sched_class->task_waking)
        p->sched_class->task_waking(p);
    /*根据进程p相关参数设定和系统状态,为进程p选择合适的cpu供其运行*/
    cpu = select_task_rq(p, p->wake_cpu, SD_BALANCE_WAKE, wake_flags,
                sibling_count_hint);
    /*如果选择的cpu与进程p当前所在的cpu不相同,则将进程的wake_flags标记添加需要迁移
     ,并将进程p迁移到cpu上.*/
    if (task_cpu(p) != cpu) {
        wake_flags |= WF_MIGRATED;
        set_task_cpu(p, cpu);
    }

#endif /* CONFIG_SMP */
    /*进程p入队操作并标记p为runnable状态,同时执行wakeup preemption,即唤醒抢占.*/
    ttwu_queue(p, cpu);
stat:
    /*调度相关的统计*/
    ttwu_stat(p, cpu, wake_flags);
out:
    raw_spin_unlock_irqrestore(&p->pi_lock, flags);

    return success;
}
```

try_to_wake_up函数解析

try_to_wakeup_up函数的主要功能解析如下:

- 根据进行状态state来决定是否需要继续唤醒动作,这就是p->state & state的目的
- 获取进程p所在的当前cpu
- 如果当前cpu已经在rq里面了,则需要翻转进程p的状态为TASK_RUNNING,这样进程p就一直在rq里面,不需要继续调度,退出唤醒动作
- 如果进程p在cpu上,则期待某个时刻去修改这个数值on_cpu为0,这样代码可以继续运行.
- 根据WALT算法来实现rq当前task和新唤醒的task p相关的task load和rq相关的runnable_load的数值,具体怎么实现看walt.c文件。
- 为task p挑选合适的cpu
- 如果挑选的cpu与进程p所在的cpu不是同一个cpu,则进程task migration操作
- 将进程p入队操作
- 统计调度相关信息作为debug使用.

对于关键的几个点,详细分析如下:

1 How does the scheduler pick a suitable CPU?

```
cpu = select_task_rq(p, p->wake_cpu, SD_BALANCE_WAKE, wake_flags,
             sibling_count_hint);
```

对函数select_task_rq的实现原理如下:

```
/*
 * The caller (fork, wakeup) owns p->pi_lock, ->cpus_allowed is stable.
 */
static inline
```

```
int select_task_rq(struct task_struct *p, int cpu, int sd_flags, int
wake_flags,
            int sibling_count_hint)
{
    lockdep_assert_held(&p->pi_lock);
    /*nr_cpus_allowed这个变量是进程p可以运行的cpu数量,一般在系统初始化的时候就已经
    设定好了的,或者可以通过设定cpu的亲和性来修改*/
    if (p->nr_cpus_allowed > 1)
        /*核心函数的callback*/
        cpu = p->sched_class->select_task_rq(p, cpu, sd_flags, wake_flags,
                                 sibling_count_hint);

    /*
     * In order not to call set_task_cpu() on a blocking task we need
     * to rely on ttwu() to place the task on a valid ->cpus_allowed
     * cpu.
     *
     * Since this is common to all placement strategies, this lives here.
     *
     * [ this allows ->select_task() to simply return task_cpu(p) and
     *   not worry about this generic constraint ]
     */
    /*1.如果选择的cpu与进程p允许运行的cpu不匹配 或者
      2.如果挑选的cpu offline
     只要满足上面的任何一条,则重新选择cpu在进程p成员变cpus_allowed里面选择*/
    if (unlikely(!cpumask_test_cpu(cpu, tsk_cpus_allowed(p)) ||
             !cpu_online(cpu)))
        cpu = select_fallback_rq(task_cpu(p), p);

    return cpu;
}
```

所以select_task_rq函数分两部分来分析:

1.1 select_task_rq callback函数分析

```
        cpu = p->sched_class->select_task_rq(p, cpu, sd_flags, wake_flags,
                                 sibling_count_hint);
 ----->
/*
 * select_task_rq_fair: Select target runqueue for the waking task in
domains
 * that have the 'sd_flag' flag set. In practice, this is SD_BALANCE_WAKE,
 * SD_BALANCE_FORK, or SD_BALANCE_EXEC.
 *
 * Balances load by selecting the idlest cpu in the idlest group, or under
 * certain conditions an idle sibling cpu if the domain has SD_WAKE_AFFINE
set.
 *
 * Returns the target cpu number.
 *
 * preempt must be disabled.
 */
static int
select_task_rq_fair(struct task_struct *p, int prev_cpu, int sd_flag, int
wake_flags,
             int sibling_count_hint)
```

```
{
    struct sched_domain *tmp, *affine_sd = NULL, *sd = NULL;
    int cpu = smp_processor_id(); /*获取当前运行的cpu id*/
    int new_cpu = prev_cpu;  /*将唤醒此进程p的cpu作为new_cpu*/
    int want_affine = 0;
    /*wake_falgs=0,so sync=0*/
    int sync = wake_flags & WF_SYNC;

#ifdef CONFIG_64BIT_ONLY_CPU
    struct cpumask tmpmask;

    if (find_packing_cpu(p, &new_cpu))
        return new_cpu;

    cpumask_andnot(&tmpmask, cpu_present_mask, &b64_only_cpu_mask);
    if (cpumask_test_cpu(cpu, &tmpmask)) {
        if (weighted_cpuload_32bit(cpu) >
            sysctl_sched_32bit_load_threshold &&
            !test_tsk_thread_flag(p, TIF_32BIT))
            return min_load_64bit_only_cpu();
    }
#endif
    /*sd_flag = SD_BALANCE_WAKE,是成立的,want_affine是一个核心变量包括三个部分数值
    的&&,后面会详细分析*/
    if (sd_flag & SD_BALANCE_WAKE) {
        record_wakee(p);
        want_affine = !wake_wide(p, sibling_count_hint) &&
                  !wake_cap(p, cpu, prev_cpu) &&
                  cpumask_test_cpu(cpu, &p->cpus_allowed);
    }
    /*如果系统使用EAS来决策的,则走这个分支,这种重点,也是新加的调度方案,根据cpu的能效和
    capacity来挑选cpu*/
    if (energy_aware())
        return select_energy_cpu_brute(p, prev_cpu, sync);

    rcu_read_lock();
    for_each_domain(cpu, tmp) {
        if (!(tmp->flags & SD_LOAD_BALANCE))
            break;

        /*
         * If both cpu and prev_cpu are part of this domain,
         * cpu is a valid SD_WAKE_AFFINE target.
         */
        if (want_affine && (tmp->flags & SD_WAKE_AFFINE) &&
            cpumask_test_cpu(prev_cpu, sched_domain_span(tmp))) {
            affine_sd = tmp;
            break;
        }

        if (tmp->flags & sd_flag)
            sd = tmp;
        else if (!want_affine)
            break;
```

```
        }

        if (affine_sd) {
            sd = NULL; /* Prefer wake_affine over balance flags */
            if (cpu != prev_cpu && wake_affine(affine_sd, p, prev_cpu, sync))
                new_cpu = cpu;
        }

        if (sd && !(sd_flag & SD_BALANCE_FORK)) {
            /*
             * We're going to need the task's util for capacity_spare_wake
             * in find_idlest_group. Sync it up to prev_cpu's
             * last_update_time.
             */
            sync_entity_load_avg(&p->se);
        }

        if (!sd) {
            if (sd_flag & SD_BALANCE_WAKE) /* XXX always ? */
                new_cpu = select_idle_sibling(p, prev_cpu, new_cpu);

        } else {
            new_cpu = find_idlest_cpu(sd, p, cpu, prev_cpu, sd_flag);
        }
        rcu_read_unlock();

        return new_cpu;
    }
```

分别分几个部分来select_task_rq_fair函数:

1.1 want_affine变量怎么获取的呢?

```

        want_affine = !wake_wide(p, sibling_count_hint) &&
                      !wake_cap(p, cpu, prev_cpu) &&
                      cpumask_test_cpu(cpu, &p->cpus_allowed);
 ---->
/*
 * Detect M:N waker/wakee relationships via a
switching-frequency heuristic.
 * A waker of many should wake a different task than the one
last awakened
 * at a frequency roughly N times higher than one of its
wakees.  In order
 * to determine whether we should let the load spread vs
consolodating to
 * shared cache, we look for a minimum 'flip' frequency of
llc_size in one
 * partner, and a factor of lls_size higher frequency in the
other.  With
 * both conditions met, we can be relatively sure that the
relationship is
 * non-monogamous, with partner count exceeding socket size.
Waker/wakee
```

```c
 * being client/server, worker/dispatcher, interrupt source or whatever is
 * irrelevant, spread criteria is apparent partner count exceeds socket size.
 */
/*当前cpu的唤醒次数没有超标*/
static int wake_wide(struct task_struct *p, int sibling_count_hint)
{
    unsigned int master = current->wakee_flips;
    unsigned int slave = p->wakee_flips;
    int llc_size = this_cpu_read(sd_llc_size);

    if (sibling_count_hint >= llc_size)
        return 1;

    if (master < slave)
        swap(master, slave);
    if (slave < llc_size || master < slave * llc_size)
        return 0;
    return 1;
}

/*
 * Disable WAKE_AFFINE in the case where task @p doesn't fit in the
 * capacity of either the waking CPU @cpu or the previous CPU @prev_cpu.
 *
 * In that case WAKE_AFFINE doesn't make sense and we'll let
 * BALANCE_WAKE sort things out.
 */
static int wake_cap(struct task_struct *p, int cpu, int prev_cpu)
{
    long min_cap, max_cap;
    /*获取当前cpu的orig_of和唤醒进程p的cpu的orig_of capacity的最小值*/
    min_cap = min(capacity_orig_of(prev_cpu), capacity_orig_of(cpu));
    /*获取最大的capacity,为1024*/
    max_cap = cpu_rq(cpu)->rd->max_cpu_capacity.val;

    /* Minimum capacity is close to max, no need to abort wake_affine */
    if (max_cap - min_cap < max_cap >> 3)
        return 0;
    /*根据PELT算法更新进程p作为调度实体的负载*/
    /* Bring task utilization in sync with prev_cpu */
    sync_entity_load_avg(&p->se);
```

```
        /*根据条件判断min_cap的capacity能够能够满足进程p吗?*/
        /*min_cap * 1024 < task_util(p) * 1138,
          task_util(p)∈[0,1024]*/
        return min_cap * 1024 < task_util(p) * capacity_margin;
}

static inline unsigned long task_util(struct task_struct *p)
{   /*WALT启用*/
#ifdef CONFIG_SCHED_WALT
    if (!walt_disabled && sysctl_sched_use_walt_task_util) {
        unsigned long demand = p->ravg.demand;/*task的真实运行时间*/
        /*在一个窗口内是多少,注意是*了1024的,比如占用了50%的窗口时间,则这个
task_util =  0.5 * 1024=512.*/
        return (demand << 10) / walt_ravg_window;
    }
#endif
    return p->se.avg.util_avg;
}

/*
 * Synchronize entity load avg of dequeued entity without locking
 * the previous rq.
 */
void sync_entity_load_avg(struct sched_entity *se)
{
    struct cfs_rq *cfs_rq = cfs_rq_of(se);
    u64 last_update_time;

    last_update_time = cfs_rq_last_update_time(cfs_rq);
  /*PELT计算sched_entity调度实体的负载*/
__update_load_avg(last_update_time, cpu_of(rq_of(cfs_rq)), &se->avg, 0, 0,
NULL);
}

/**
 * cpumask_test_cpu - test for a cpu in a cpumask
 * @cpu: cpu number (< nr_cpu_ids)
 * @cpumask: the cpumask pointer
 *
 * Returns 1 if @cpu is set in @cpumask, else returns 0
 */
/*当前运行的cpu是否是task p cpu亲和数里面的一个*/
static inline int cpumask_test_cpu(int cpu, const struct
cpumask *cpumask)
{
    return test_bit(cpumask_check(cpu),
cpumask_bits((cpumask)));
}
```

只有满足下面三个条件:

- 当前cpu的唤醒次数没有超标
- 当前task p消耗的capacity * 1138小于min_cap * 1024
- 当前cpu在task p的cpu亲和数里面的一个

只有上面三个条件全部成立,则want_affine=1

1.2 使用EAS调度,怎么挑选合理的cpu呢?

如果使用EAS调度算法,则energy_aware()为true:

```
    if (energy_aware())
        return select_energy_cpu_brute(p, prev_cpu, sync);

static inline bool energy_aware(void)
{   /*energy_aware调度类*/
    return sched_feat(ENERGY_AWARE);
}

static int select_energy_cpu_brute(struct task_struct *p, int prev_cpu, int sync)
{
    struct sched_domain *sd;
    int target_cpu = prev_cpu, tmp_target, tmp_backup;
    bool boosted, prefer_idle;
    /*调度统计信息*/
    schedstat_inc(p, se.statistics.nr_wakeups_secb_attempts);
    schedstat_inc(this_rq(), eas_stats.secb_attempts);
    /*条件不成立*/
    if (sysctl_sched_sync_hint_enable && sync) {
        int cpu = smp_processor_id();

        if (cpumask_test_cpu(cpu, tsk_cpus_allowed(p))) {
            schedstat_inc(p, se.statistics.nr_wakeups_secb_sync);
            schedstat_inc(this_rq(), eas_stats.secb_sync);
            return cpu;
        }
    }

    rcu_read_lock();
    /*下面的两个参数都可以在init.rc里面配置,一般boost都会配置,尤其是top-app*/
#ifdef CONFIG_CGROUP_SCHEDTUNE
    /*获取当前task是否有util boost增益.如果有则boosted=true.
     即如果原先的负载为util,那么boost之后的负载为util+boost/100*util*/
    boosted = schedtune_task_boost(p) > 0;
    /*获取在挑选cpu的时候,是否更倾向于idle cpu,默认为0,也就是说 prefer_idle=false*/
    prefer_idle = schedtune_prefer_idle(p) > 0;
#else
    boosted = get_sysctl_sched_cfs_boost() > 0;
    prefer_idle = 0;
#endif
    /*再次更新调度实体负载,使用PELT算法,比较奇怪的时候,在计算want_affine→
wake_cap函数里面已经update了调度实体的负载了,为何在这里还需要再次计算呢?*/
    sync_entity_load_avg(&p->se);
    /*DEFINE_PER_CPU(struct sched_domain *, sd_ea),在解析调度域调度组的创建和初始
化的时候,解析过,每个cpu在每个SDTL上面都有对应的调度域*/
    sd = rcu_dereference(per_cpu(sd_ea, prev_cpu));
    /* Find a cpu with sufficient capacity */
    /*核心函数*/
    tmp_target = find_best_target(p, &tmp_backup, boosted, prefer_idle);

```

- ……………………………………………….

1.2.1 下面讲解核心函数find_best_target,函数比较长:

```
static inline int find_best_target(struct task_struct *p, int *backup_cpu,
                        bool boosted, bool prefer_idle)
{
    unsigned long best_idle_min_cap_orig = ULONG_MAX;
    /*计算task p经过boost之后的util数值,即在task_util(p)的基础上+boost%*util*/
    unsigned long min_util = boosted_task_util(p);
    unsigned long target_capacity = ULONG_MAX;
    unsigned long min_wake_util = ULONG_MAX;
    unsigned long target_max_spare_cap = 0;
    int best_idle_cstate = INT_MAX;
    unsigned long target_cap = ULONG_MAX;
    unsigned long best_idle_cap_orig = ULONG_MAX;
    int best_idle = INT_MAX;
    int backup_idle_cpu = -1;
    struct sched_domain *sd;
    struct sched_group *sg;
    int best_active_cpu = -1;
    int best_idle_cpu = -1;
    int target_cpu = -1;
    int cpu, i;
    /*获取当前cpu的运行队列rq的root_domain*/
    struct root_domain *rd = cpu_rq(smp_processor_id())->rd;
    /*获取当前root_domain的最大capacity数值*/
    unsigned long max_cap = rd->max_cpu_capacity.val;

    *backup_cpu = -1;

    schedstat_inc(p, se.statistics.nr_wakeups_fbt_attempts);
    schedstat_inc(this_rq(), eas_stats.fbt_attempts);

    /* Find start CPU based on boost value */
    /*start_cpu找出rd->min_cap_orig_cpu,即min_cap_orig的第一个cpu id
     min的cpu为0*/
    cpu = start_cpu(boosted);
    if (cpu < 0) {
        schedstat_inc(p, se.statistics.nr_wakeups_fbt_no_cpu);
        schedstat_inc(this_rq(), eas_stats.fbt_no_cpu);
        return -1;
    }

    /* Find SD for the start CPU */
    /*找到启动cpu的调度域*/
    sd = rcu_dereference(per_cpu(sd_ea, cpu));
    if (!sd) {
        schedstat_inc(p, se.statistics.nr_wakeups_fbt_no_sd);
        schedstat_inc(this_rq(), eas_stats.fbt_no_sd);
        return -1;
    }

    /* Scan CPUs in all SDs */
    sg = sd->groups;
    do {
```

```
        for_each_cpu_and(i, tsk_cpus_allowed(p), sched_group_cpus(sg)) {
            unsigned long capacity_orig = capacity_orig_of(i);
            unsigned long wake_util, new_util;

            if (!cpu_online(i))
                continue;

            if (walt_cpu_high_irqload(i))
                continue;

            /*
             * p's blocked utilization is still accounted for on prev_cpu
             * so prev_cpu will receive a negative bias due to the double
             * accounting. However, the blocked utilization may be zero.
             */
            wake_util = cpu_util_wake(i, p);
            new_util = wake_util + task_util(p);

            /*
             * Ensure minimum capacity to grant the required boost.
             * The target CPU can be already at a capacity level higher
             * than the one required to boost the task.
             */
            new_util = max(min_util, new_util);
            if (new_util > capacity_orig) {
                if (idle_cpu(i)) {
                    int idle_idx;

                    idle_idx =
                        idle_get_state_idx(cpu_rq(i));

                    if (capacity_orig >
                        best_idle_cap_orig) {
                        best_idle_cap_orig =
                            capacity_orig;
                        best_idle = idle_idx;
                        backup_idle_cpu = i;
                        continue;
                    }

                    /*
                     * Skip CPUs in deeper idle state, but
                     * only if they are also less energy
                     * efficient.
                     * IOW, prefer a deep IDLE LITTLE CPU
                     * vs a shallow idle big CPU.
                     */
                    if (sysctl_sched_cstate_aware &&
                        best_idle <= idle_idx)
                        continue;

                    /* Keep track of best idle CPU */
                    best_idle_cap_orig = capacity_orig;
                    best_idle = idle_idx;
```

```
                backup_idle_cpu = i;
                continue;
            }

            if (capacity_orig > target_cap) {
                target_cap = capacity_orig;
                min_wake_util = wake_util;
                best_active_cpu = i;
                continue;
            }

            if (wake_util > min_wake_util)
                continue;

            min_wake_util = wake_util;
            best_active_cpu = i;
            continue;

        }
        /*
         * Enforce EAS mode
         *
         * For non latency sensitive tasks, skip CPUs that
         * will be overutilized by moving the task there.
         *
         * The goal here is to remain in EAS mode as long as
         * possible at least for !prefer_idle tasks.
         */
        if (capacity_orig == max_cap)
            if (idle_cpu(i))
                goto skip;

        if ((new_util * capacity_margin) >
             (capacity_orig * SCHED_CAPACITY_SCALE))
            continue;
skip:
        if (idle_cpu(i)) {
            int idle_idx;

            if (prefer_idle ||
                cpumask_test_cpu(i, &min_cap_cpu_mask)) {
                trace_sched_find_best_target(p,
                    prefer_idle, min_util, cpu,
                    best_idle_cpu, best_active_cpu,
                    i);
                return i;
            }
            idle_idx = idle_get_state_idx(cpu_rq(i));

            /* Select idle CPU with lower cap_orig */
            if (capacity_orig > best_idle_min_cap_orig)
                continue;

            /*
```

```
                 * Skip CPUs in deeper idle state, but only
                 * if they are also less energy efficient.
                 * IOW, prefer a deep IDLE LITTLE CPU vs a
                 * shallow idle big CPU.
                 */
                if (sysctl_sched_cstate_aware &&
                    best_idle_cstate <= idle_idx)
                    continue;

                /* Keep track of best idle CPU */
                best_idle_min_cap_orig = capacity_orig;
                best_idle_cstate = idle_idx;
                best_idle_cpu = i;
                continue;
            }

            /* Favor CPUs with smaller capacity */
            if (capacity_orig > target_capacity)
                continue;

            /* Favor CPUs with maximum spare capacity */
            if ((capacity_orig - new_util) < target_max_spare_cap)
                continue;

            target_max_spare_cap = capacity_orig - new_util;
            target_capacity = capacity_orig;
            target_cpu = i;
        }

    } while (sg = sg->next, sg != sd->groups);

    /*
     * For non latency sensitive tasks, cases B and C in the previous loop,
     * we pick the best IDLE CPU only if we was not able to find a target
     * ACTIVE CPU.
     *
     * Policies priorities:
     *
     * - prefer_idle tasks:
     *
     *   a) IDLE CPU available, we return immediately
     *   b) ACTIVE CPU where task fits and has the bigger maximum spare
     *      capacity (i.e. target_cpu)
     *   c) ACTIVE CPU with less contention due to other tasks
     *      (i.e. best_active_cpu)
     *
     * - NON prefer_idle tasks:
     *
     *   a) ACTIVE CPU: target_cpu
     *   b) IDLE CPU: best_idle_cpu
     */
    if (target_cpu == -1) {
        if (best_idle_cpu != -1)
```

```
214.                 target_cpu = best_idle_cpu;
215.             else
216.                 target_cpu = (backup_idle_cpu != -1)
217.                 ? backup_idle_cpu
218.                 : best_active_cpu;
219.         } else
220.             *backup_cpu = best_idle_cpu;
221.
222.         trace_sched_find_best_target(p, prefer_idle, min_util, cpu,
223.                          best_idle_cpu, best_active_cpu,
224.                          target_cpu);
225.
226.         schedstat_inc(p, se.statistics.nr_wakeups_fbt_count);
227.         schedstat_inc(this_rq(), eas_stats.fbt_count);
228.
229.         return target_cpu;
230.     }
```

分下面如下几个部分来分析上面的do{}while()循环

- do{}while()循环是对sched domain里面的所有调度组进行遍历
- for_each_cpu_and(i, tsk_cpus_allowed(p), sched_group_cpus(sg)),这个for循环比较有意思,sched_group_cpus(sg),表示这个调度组所有的cpumask,其返回值就是sg->cpumask.for_each_cpu_and抽象循环的意思是i必须在tsk_cpus_allowed(p) && sched_group_cpus(sg)交集里面.

```
/**
 * for_each_cpu_and - iterate over every cpu in both masks
 * @cpu: the (optionally unsigned) integer iterator
 * @mask: the first cpumask pointer
 * @and: the second cpumask pointer
 *
 * This saves a temporary CPU mask in many places.  It is equivalent to:
 *  struct cpumask tmp;
 *  cpumask_and(&tmp, &mask, &and);
 *  for_each_cpu(cpu, &tmp)
 *      ...
 *
 * After the loop, cpu is >= nr_cpu_ids.
 */
#define for_each_cpu_and(cpu, mask, and)               \
    for ((cpu) = -1;                                \
        (cpu) = cpumask_next_and((cpu), (mask), (and)),      \
        (cpu) < nr_cpu_ids;)
```

- 如果cpu id为i的cpu offline,则遍历下一个cpu
- 如果这个cpu是一个irq high load的cpu,则遍历下一个cpu
- 计算wake_util,即为当前cpu id=i的cpu_util的数值,new_util为cpu i的cpu_util+进程p的task_util数值,最后new_util = max(min_util, new_util);min_util数值是task_util boost之后的数值,如果没有boost,则min_util=task_util.

上面条件判断之后,并且获取了当前遍历cpu的util和新唤醒的进程task_util叠加到cpu_util变成new_util之后,通过capacity/util的比较来获取target_cpu,下面分析代码77~187行代码:
在遍历cpu的时候
如果new_util > 遍历的cpu的capacity数值(dts获取),分两部分逻辑处理:

1. 如果cpu是idle状态.记录此cpu处在idle的level_idx,并修改下面三个参数数值之后遍历下一个符合条件的cpu:

- /*获取当前遍历cpu的capacity,并保存在best_idle_cap_orig变量中 */
- best_idle_cap_orig = capacity_orig;
- best_idle = idle_idx; //获取idle level index
- backup_idle_cpu = i; //idle cpu number,后面会使用到

2.如果不是idle cpu,则修正下面两个参数之后遍历下一个符合条件的cpu:

- min_wake_util = wake_util; //将遍历的cpu_util赋值给min_wake_uti
- best_active_cpu = i; //得到best_active_cpu id

如果new_util <= 遍历的cpu的capacity数值(dts获取),分如下几部分逻辑处理:

1.如果capacity_orig == max_cap并且遍历的cpu恰好是idle状态,直接调到去更新下面三个参数:

- /*将当前遍历的cpu的capacity保存到best_idle_min_cap_orig变量中*/
- best_idle_min_cap_orig = capacity_orig;
- best_idle_cstate = idle_idx; //保存idleindex
- best_idle_cpu = i; //保存最佳的idle cpu,后面会用到

2.如果(new_util * capacity_margin) > (capacity_orig * SCHED_CAPACITY_SCALE)成立,则直接遍历下一个cpu,说明此时遍历的cpu已经overutilization,没必须继续遍历了.
3.如果capacity_orig==max_cap不成立,也会执行第一条先判断遍历的cpu是否是idle,是的话,执行1一样的流程
4.如果遍历的cpu不是idle,则比较capacity_orig-new_util差值与target_max_spare_cap的比较目的是选择一个差值余量最大的cpu,防止新唤醒的task p在余量不足的cpu上运行导致后面的负载均衡,白白浪费系统资源.同时更新下面三个参数,并在每次遍历的时候更新:

- /*util余量,目的找出最大余量的cpu id*/
- target_max_spare_cap = capacity_orig - new_util;
- /*目标capacity*/
- target_capacity = capacity_orig;
- /*选择的目标cpu*/
- target_cpu = i;
-

下面分析212~220之间的代码:
从解释来看(对于非敏感延迟性进程)进程分两种,prefer_idle flag:

1. 一种是偏爱idle cpu运行的进程,那么如果有idle cpu,则优先选择idle cpu并立即返回,如代码145~152行的代码;之后task的util不太大,就选择有最大余量的cpu了;最后挑选有更少争抢的cpu,比如best_avtive_cpu
2. 不偏爱idle cpu运行的进程,优先选择余量最大的cpu,之后选择best_idle_cpu

明白了prefer_idle这个flag会影响对cpu类型的选择,那么分析212~220行之间的代码如下,即在没有机会执行寻找最大余量的cpu capacity的情况下:
如果target_cpu = -1

- 如果best_idle_cpu更新过,表明要么new_util很大,要么大部分cpu处于idle状态,这时候直接选择best_idle_cpu为target_cpu
- 否则,根据backup_idle_cpu是否update过来决定target_cpu选择是backup_idle_cpu还是best_active_cpu

如果target_cpu!=-1

- 候选cpu设置为best_idle_cpu,通过函数指针被使用

最后返回target_cpu的作为选择的cpu id.

1.2.2 select_energy_cpu_brute函数剩余部分分析:

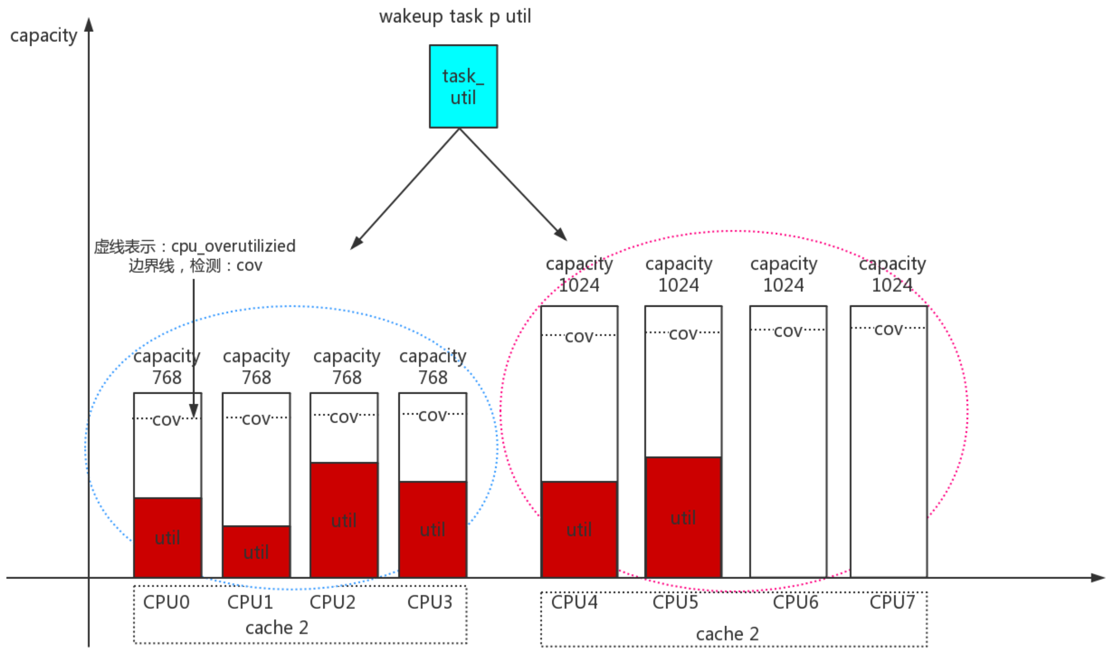
```
static int select_energy_cpu_brute(struct task_struct *p, int prev_cpu, int
sync)
{
.......................
    /* Find a cpu with sufficient capacity */
    /*下面这个函数已经解析完毕*/
    tmp_target = find_best_target(p, &tmp_backup, boosted, prefer_idle);

    if (!sd)
        goto unlock;
    if (tmp_target >= 0) {
        target_cpu = tmp_target;
        /*如果boosted or prefer_idle && target_cpu为idlecpu,或者target_cpu为
        min_cap的cpu,则挑选的cpu就是target_cpu了并直接退出代码流程*/
        if (((boosted || prefer_idle) && idle_cpu(target_cpu)) ||
             cpumask_test_cpu(target_cpu, &min_cap_cpu_mask)) {
            schedstat_inc(p, se.statistics.nr_wakeups_secb_idle_bt);
            schedstat_inc(this_rq(), eas_stats.secb_idle_bt);
            goto unlock;
        }
    }
    /*如果target_cpu等于唤醒进程p的cpu并且best_idle_cpu>=0,则修改target_cpu为
    best_idle_cpu数值,目的不在唤醒进程P的cpu上运行.why???*/
    if (target_cpu == prev_cpu && tmp_backup >= 0) {
        target_cpu = tmp_backup;
        tmp_backup = -1;
    }

    if (target_cpu != prev_cpu) {
        int delta = 0;
        /*构造需要迁移的环境变量*/
        struct energy_env eenv = {
            .util_delta     = task_util(p),
            .src_cpu        = prev_cpu,
            .dst_cpu        = target_cpu,
            .task           = p,
            .trg_cpu    = target_cpu,
        };


#ifdef CONFIG_SCHED_WALT
        if (!walt_disabled && sysctl_sched_use_walt_cpu_util &&
            p->state == TASK_WAKING)
            /*获取进程P本身的util load*/
            delta = task_util(p);
#endif
        /* Not enough spare capacity on previous cpu */
        /*唤醒进程p的cpu负载过载,超过本身capacity的90%.*/
        if (__cpu_overutilized(prev_cpu, delta, p)) {
            /*有限选择Energy合理的cpu,即小的cluster的idle cpu*/
            if (tmp_backup >= 0 &&
                capacity_orig_of(tmp_backup) <
                    capacity_orig_of(target_cpu))
```

```
                target_cpu = tmp_backup;
            schedstat_inc(p, se.statistics.nr_wakeups_secb_insuff_cap);
            schedstat_inc(this_rq(), eas_stats.secb_insuff_cap);
            goto unlock;
        }
        /*计算pre_cpu与target_cpu的功耗差异,如果大于0,则执行下面的代码流程
        就是计算MC知道DIE的SDTL的功耗总和的差异*/
        if (energy_diff(&eenv) >= 0) {
            /* No energy saving for target_cpu, try backup */
            target_cpu = tmp_backup;
            eenv.dst_cpu = target_cpu;
            eenv.trg_cpu = target_cpu;
            if (tmp_backup < 0 ||
                tmp_backup == prev_cpu ||
                energy_diff(&eenv) >= 0) {
                schedstat_inc(p, se.statistics.nr_wakeups_secb_no_nrg_sav);
                schedstat_inc(this_rq(), eas_stats.secb_no_nrg_sav);
                target_cpu = prev_cpu;
                goto unlock;
            }
        }

        schedstat_inc(p, se.statistics.nr_wakeups_secb_nrg_sav);
        schedstat_inc(this_rq(), eas_stats.secb_nrg_sav);
        goto unlock;
    }

    schedstat_inc(p, se.statistics.nr_wakeups_secb_count);
    schedstat_inc(this_rq(), eas_stats.secb_count);

unlock:
    rcu_read_unlock();

    return target_cpu;
}
```

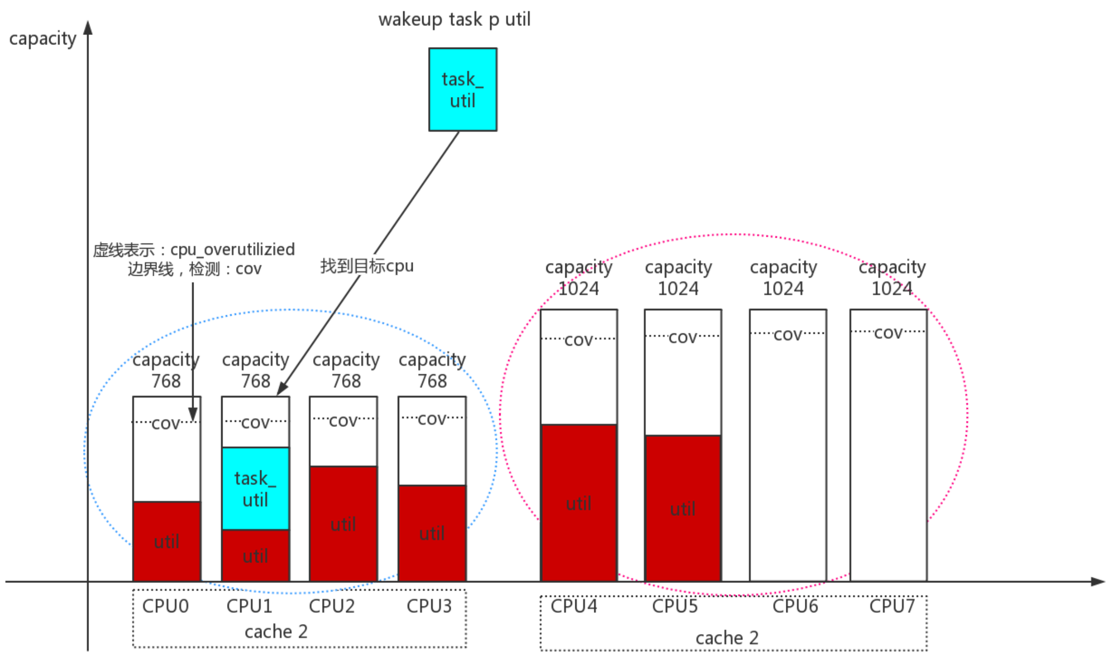
总结下就是如下四点:

1. EAS通过对sd = rcu_dereference(per_cpu(sd_ea, cpu)), 是DIE顶层domain, 这个调度域的调度组和符合进程亲和数调度组内cpu进行遍历, 查找出符合要求

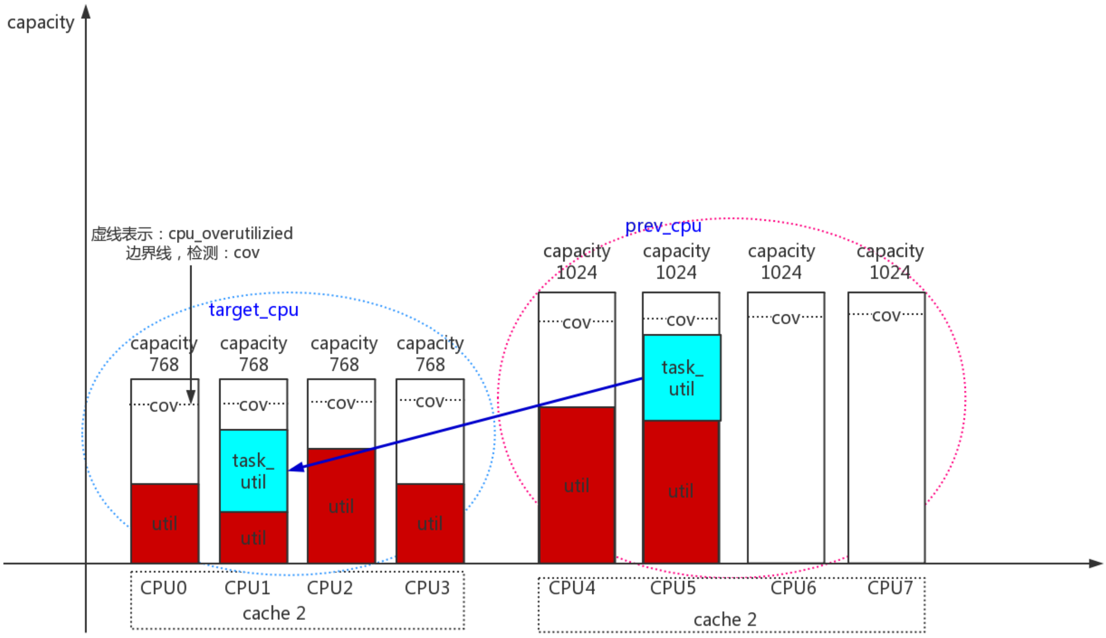
的目标cpu。



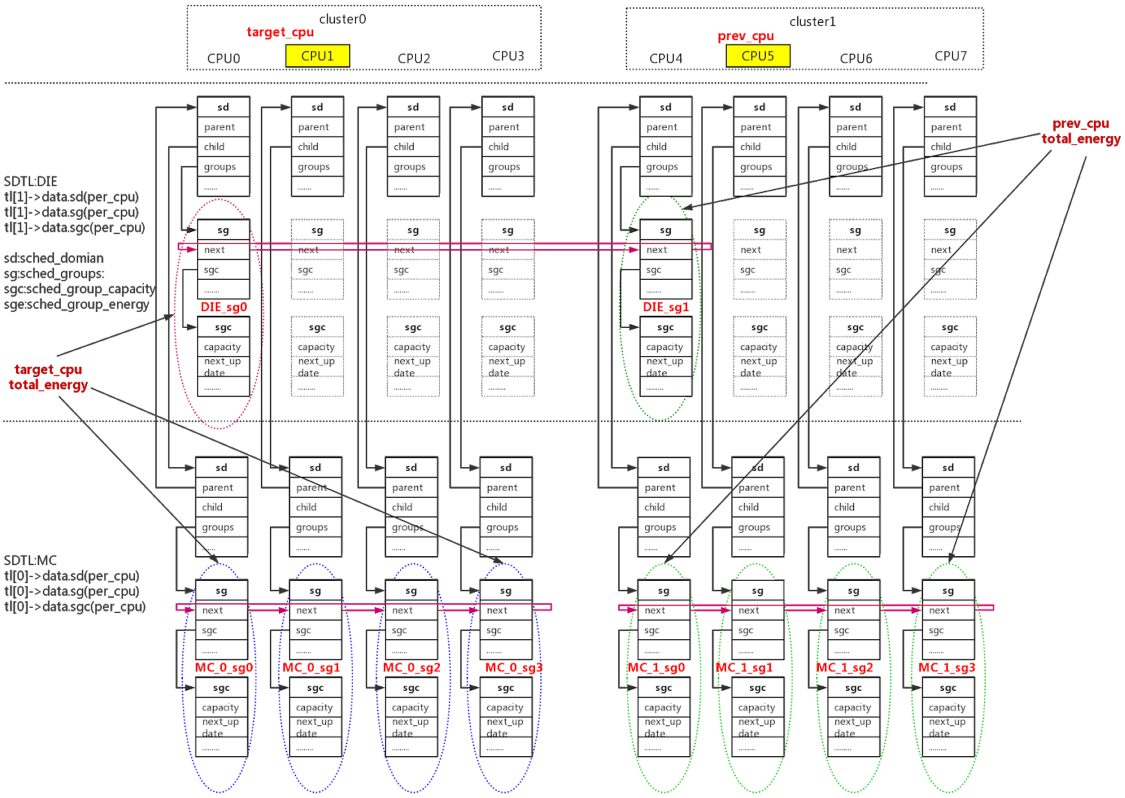
2. 当target_cpu!=-1,将task_util放在余量最大的cpu上。



3. prev_cpu是进程p上一次运行的cpu作为src_cpu，上面选择的target_cpu作为dst_cpu，就是尝试计算进程p从prev_cpu迁移到target_cpu系统的功耗差异，如果进程在target_cpu上的power < prev_cpu上的power，则直接选择target_cpu



4. 计算负载变化前后，target_cpu和prev_cpu带来的power变化。如果power的变化超过一定的门限比重，则直接选择target_cpu，如果有power增加超过一定门限比重，根据情况返回best_idle_cpu或者prev_cpu。计算负载变化的函数energy_diff()循环很多比较复杂，仔细分析下来就是计算target_cpu/prev_cpu在“MC层次cpu所在sg链表”+“DIE层级cpu所在sg”，这两种范围在负载变化中的功耗差异。示意图如下。



上面四个过程很清晰明了的pre_cpu和target_cpu的转换关系

接下来分析energy_diff(&eenv)怎么来计算pre_cpu与target_cpu power之间的关系的。

```
static inline int
energy_diff(struct energy_env *eenv)
{
    int boost = schedtune_task_boost(eenv->task);
    int nrg_delta;
    /*计算绝对功耗差值*/
    /* Conpute "absolute" energy diff */
    __energy_diff(eenv);

    /* Return energy diff when boost margin is 0 */
    if (1 || boost == 0) {
        trace_sched_energy_diff(eenv->task,
                eenv->src_cpu, eenv->dst_cpu, eenv->util_delta,
                eenv->nrg.before, eenv->nrg.after, eenv->nrg.diff,
                eenv->cap.before, eenv->cap.after, eenv->cap.delta,
                0, -eenv->nrg.diff);
        return eenv->nrg.diff;
    }

    /* Compute normalized energy diff */
    nrg_delta = normalize_energy(eenv->nrg.diff);
    eenv->nrg.delta = nrg_delta;

    eenv->payoff = schedtune_accept_deltas(
            eenv->nrg.delta,
            eenv->cap.delta,
            eenv->task);

```

```
	trace_sched_energy_diff(eenv->task,
			eenv->src_cpu, eenv->dst_cpu, eenv->util_delta,
			eenv->nrg.before, eenv->nrg.after, eenv->nrg.diff,
			eenv->cap.before, eenv->cap.after, eenv->cap.delta,
			eenv->nrg.delta, eenv->payoff);

	/*
	 * When SchedTune is enabled, the energy_diff() function will return
	 * the computed energy payoff value. Since the energy_diff() return
	 * value is expected to be negative by its callers, this evaluation
	 * function return a negative value each time the evaluation return a
	 * positive payoff, which is the condition for the acceptance of
	 * a scheduling decision
	 */
	return -eenv->payoff;
}

/*
 * energy_diff(): Estimate the energy impact of changing the utilization
 * distribution. eenv specifies the change: utilisation amount, source, and
 * destination cpu. Source or destination cpu may be -1 in which case the
 * utilization is removed from or added to the system (e.g. task wake-up).
If
 * both are specified, the utilization is migrated.
 */
static inline int __energy_diff(struct energy_env *eenv)
{
	struct sched_domain *sd;
	struct sched_group *sg;
	int sd_cpu = -1, energy_before = 0, energy_after = 0;
	int diff, margin;
	/*在brute函数里面设置好的eenv参数,构造迁移前的环境变量*/
	struct energy_env eenv_before = {
		.util_delta = task_util(eenv->task),
		.src_cpu    = eenv->src_cpu,
		.dst_cpu    = eenv->dst_cpu,
		.trg_cpu    = eenv->src_cpu,
		.nrg        = { 0, 0, 0, 0},
		.cap        = { 0, 0, 0 },
		.task       = eenv->task,
	};

	if (eenv->src_cpu == eenv->dst_cpu)
		return 0;
	/*sd来至于cache sd_ea，是cpu对应的顶层sd(tl DIE层)*/
	sd_cpu = (eenv->src_cpu != -1) ? eenv->src_cpu : eenv->dst_cpu;
	sd = rcu_dereference(per_cpu(sd_ea, sd_cpu));

	if (!sd)
		return 0; /* Error */

	sg = sd->groups;
	/*遍历sg所在sg链表，找到符合条件的sg， 累加计算eenv_before、eenv
	相关sg的功耗*/
	do {  /*如果当前sg包含src_cpu或者dst_cpu，则进行计算*/
		if (cpu_in_sg(sg, eenv->src_cpu) || cpu_in_sg(sg, eenv->dst_cpu)) {
			/*当前顶层sg为eenv的sg_top*/
			eenv_before.sg_top = eenv->sg_top = sg;
			/*计算eenv_before负载下sg的power*/
			if (sched_group_energy(&eenv_before))
				return 0; /* Invalid result abort */
			energy_before += eenv_before.energy;

			/* Keep track of SRC cpu (before) capacity */
```

```
            eenv->cap.before = eenv_before.cap.before;
            eenv->cap.delta = eenv_before.cap.delta;
            /*计算eenv负载下sg的power*/
            if (sched_group_energy(eenv))
                return 0; /* Invalid result abort */
            energy_after += eenv->energy;
        }
    } while (sg = sg->next, sg != sd->groups);
    /*计算energy_after - energy_before*/
    eenv->nrg.before = energy_before;
    eenv->nrg.after = energy_after;
    eenv->nrg.diff = eenv->nrg.after - eenv->nrg.before;
    eenv->payoff = 0;
#ifndef CONFIG_SCHED_TUNE
    trace_sched_energy_diff(eenv->task,
            eenv->src_cpu, eenv->dst_cpu, eenv->util_delta,
            eenv->nrg.before, eenv->nrg.after, eenv->nrg.diff,
            eenv->cap.before, eenv->cap.after, eenv->cap.delta,
            eenv->nrg.delta, eenv->payoff);
#endif
    /*
     * Dead-zone margin preventing too many migrations.
     */

    margin = eenv->nrg.before >> 6; /* ~1.56% */

    diff = eenv->nrg.after - eenv->nrg.before;

    eenv->nrg.diff = (abs(diff) < margin) ? 0 : eenv->nrg.diff;

    return eenv->nrg.diff;
}
/*接下来看sched_group_energy函数的实现过程*/
/*
 * sched_group_energy(): Computes the absolute energy consumption of cpus
 * belonging to the sched_group including shared resources shared only by
 * members of the group. Iterates over all cpus in the hierarchy below the
 * sched_group starting from the bottom working it's way up before going to
 * the next cpu until all cpus are covered at all levels. The current
 * implementation is likely to gather the same util statistics multiple
times.
 * This can probably be done in a faster but more complex way.
 * Note: sched_group_energy() may fail when racing with sched_domain
updates.
 */
static int sched_group_energy(struct energy_env *eenv)
{
    struct cpumask visit_cpus;
    u64 total_energy = 0;
    int cpu_count;

    WARN_ON(!eenv->sg_top->sge);

    cpumask_copy(&visit_cpus, sched_group_cpus(eenv->sg_top));
    /* If a cpu is hotplugged in while we are in this function,
     * it does not appear in the existing visit_cpus mask
     * which came from the sched_group pointer of the
     * sched_domain pointed at by sd_ea for either the prev
     * or next cpu and was dereferenced in __energy_diff.
```

```
        * Since we will dereference sd_scs later as we iterate
        * through the CPUs we expect to visit, new CPUs can
        * be present which are not in the visit_cpus mask.
        * Guard this with cpu_count.
        */
    cpu_count = cpumask_weight(&visit_cpus);
    /*根据sg_top顶层sd，找到需要计算的cpu集合visit_cpus，逐个遍历其中每一个
   cpu,这一套复杂的循环算法计算下来，其实就计算了几个power，以cpu0-cpu3为例
  :4个底层sg的power + 1个顶层sg的power*/
    while (!cpumask_empty(&visit_cpus)) {
        struct sched_group *sg_shared_cap = NULL;
        /*选取visit_cpus中的第一个cpu*/
        int cpu = cpumask_first(&visit_cpus);
        struct sched_domain *sd;

        /*
         * Is the group utilization affected by cpus outside this
         * sched_group?
         * This sd may have groups with cpus which were not present
         * when we took visit_cpus.
         */
        sd = rcu_dereference(per_cpu(sd_scs, cpu));

        if (sd && sd->parent)
            sg_shared_cap = sd->parent->groups;
         /*从底层到顶层逐个遍历cpu所在的sd*/
        for_each_domain(cpu, sd) {
            struct sched_group *sg = sd->groups;
             /*如果是顶层sd，只会计算一个sg*/
            /* Has this sched_domain already been visited? */
            if (sd->child && group_first_cpu(sg) != cpu)
                break;
            /*逐个遍历该层次sg链表所在sg*/
            do {
                unsigned long group_util;
                int sg_busy_energy, sg_idle_energy;
                int cap_idx, idle_idx;

                if (sg_shared_cap && sg_shared_cap->group_weight >=
sg->group_weight)
                    eenv->sg_cap = sg_shared_cap;
                else
                    eenv->sg_cap = sg;
                /*根据eenv指示的负载变化，找出满足该sg中最大负载cpu的
                   capacity_index*/
                cap_idx = find_new_capacity(eenv, sg->sge);

                if (sg->group_weight == 1) {
                    /* Remove capacity of src CPU (before task move) */
                    if (eenv->trg_cpu == eenv->src_cpu &&
                        cpumask_test_cpu(eenv->src_cpu,
sched_group_cpus(sg))) {
```

```
                        eenv->cap.before = sg->sge->cap_states[cap_idx].cap;
                        eenv->cap.delta -= eenv->cap.before;
                    }
                    /* Add capacity of dst CPU  (after task move) */
                    if (eenv->trg_cpu == eenv->dst_cpu &&
                        cpumask_test_cpu(eenv->dst_cpu,
sched_group_cpus(sg))) {
                        eenv->cap.after = sg->sge->cap_states[cap_idx].cap;
                        eenv->cap.delta += eenv->cap.after;
                    }
                }
                /*找出sg所有cpu中最小的idle index*/
                idle_idx = group_idle_state(eenv, sg);
 /*累加sg中所有cpu的相对负载，
             最大负载为sg->sge->cap_states[eenv->cap_idx].cap*/
                group_util = group_norm_util(eenv, sg);
      /*计算power = busy_power + idle_power*/
                sg_busy_energy = (group_util *
sg->sge->cap_states[cap_idx].power);
                sg_idle_energy = ((SCHED_LOAD_SCALE-group_util)
                                * sg->sge->idle_states[idle_idx].power);

                total_energy += sg_busy_energy + sg_idle_energy;

                if (!sd->child) {
                    /*
                     * cpu_count here is the number of
                     * cpus we expect to visit in this
                     * calculation. If we race against
                     * hotplug, we can have extra cpus
                     * added to the groups we are
                     * iterating which do not appear in
                     * the visit_cpus mask. In that case
                     * we are not able to calculate energy
                     * without restarting so we will bail
                     * out and use prev_cpu this time.
                     */
                    if (!cpu_count)
                        return -EINVAL;
                    /*如果遍历了底层sd，从visit_cpus中去掉对应的sg cpu*/
                    cpumask_xor(&visit_cpus, &visit_cpus,
sched_group_cpus(sg));
                    cpu_count--;
                }

                if (cpumask_equal(sched_group_cpus(sg),
sched_group_cpus(eenv->sg_top)))
                    goto next_cpu;

            } while (sg = sg->next, sg != sd->groups);
        }

```

```
        /*
         * If we raced with hotplug and got an sd NULL-pointer;
         * returning a wrong energy estimation is better than
         * entering an infinite loop.
         * Specifically: If a cpu is unplugged after we took
         * the visit_cpus mask, it no longer has an sd_scs
         * pointer, so when we dereference it, we get NULL.
         */
        if (cpumask_test_cpu(cpu, &visit_cpus))
            return -EINVAL;
next_cpu:  /*如果遍历了cpu的底层到顶层sd, 从visit_cpus中去掉对应的cpu*/
        cpumask_clear_cpu(cpu, &visit_cpus);
        continue;
    }

    eenv->energy = total_energy >> SCHED_CAPACITY_SHIFT;
    return 0;
}

```

计算思想简单，但是代码计算比较烧脑，痛苦。。。。。。。。。。。。。。

1.3 如果不使用EAS,那又怎么挑选合理的cpu呢?

接着分析select_task_rq_fair函数剩下的部分:

```
static int
select_task_rq_fair(struct task_struct *p, int prev_cpu, int sd_flag, int wake_flags,
            int sibling_count_hint)
{
............
    if (energy_aware())
        return select_energy_cpu_brute(p, prev_cpu, sync);
    /*如果没有启用EAS,则使用传统的方式来挑选合适的cpu*/
    rcu_read_lock();
    for_each_domain(cpu, tmp) {
        if (!(tmp->flags & SD_LOAD_BALANCE))
            break;

        /*
         * If both cpu and prev_cpu are part of this domain,
         * cpu is a valid SD_WAKE_AFFINE target.
         */
        if (want_affine && (tmp->flags & SD_WAKE_AFFINE) &&
            cpumask_test_cpu(prev_cpu, sched_domain_span(tmp))) {
            affine_sd = tmp;
            break;
        }

        if (tmp->flags & sd_flag)
            sd = tmp;
        else if (!want_affine)
            break;
    }

    if (affine_sd) {
```

```
            sd = NULL; /* Prefer wake_affine over balance flags */
            if (cpu != prev_cpu && wake_affine(affine_sd, p, prev_cpu, sync))
                new_cpu = cpu;
        }

        if (sd && !(sd_flag & SD_BALANCE_FORK)) {
            /*
             * We're going to need the task's util for capacity_spare_wake
             * in find_idlest_group. Sync it up to prev_cpu's
             * last_update_time.
             */
            sync_entity_load_avg(&p->se);
        }

        if (!sd) {
            if (sd_flag & SD_BALANCE_WAKE) /* XXX always ? */
                new_cpu = select_idle_sibling(p, prev_cpu, new_cpu);

        } else {
            new_cpu = find_idlest_cpu(sd, p, cpu, prev_cpu, sd_flag);
        }
        rcu_read_unlock();

        return new_cpu;
    }
```

2 当挑选的cpu与当前唤醒进程所在的cpu不同时,怎么处理?

```
        cpu = select_task_rq(p, p->wake_cpu, SD_BALANCE_WAKE, wake_flags,
                     sibling_count_hint);
        if (task_cpu(p) != cpu) {
            wake_flags |= WF_MIGRATED;
            set_task_cpu(p, cpu);
        }
      ……………..

    void set_task_cpu(struct task_struct *p, unsigned int new_cpu)
    {
    #ifdef CONFIG_SCHED_DEBUG
        /*
         * We should never call set_task_cpu() on a blocked task,
         * ttwu() will sort out the placement.
         */
        WARN_ON_ONCE(p->state != TASK_RUNNING && p->state != TASK_WAKING &&
                !p->on_rq);

    #ifdef CONFIG_LOCKDEP
        /*
         * The caller should hold either p->pi_lock or rq->lock, when changing
         * a task's CPU. ->pi_lock for waking tasks, rq->lock for runnable
    tasks.
         *
         * sched_move_task() holds both and thus holding either pins the cgroup,
```

```
        * see task_group().
        *
        * Furthermore, all task_rq users should acquire both locks, see
        * task_rq_lock().
        */
       WARN_ON_ONCE(debug_locks && !(lockdep_is_held(&p->pi_lock) ||
                                     lockdep_is_held(&task_rq(p)->lock)));
#endif
#endif

       trace_sched_migrate_task(p, new_cpu);

       if (task_cpu(p) != new_cpu) {
           if (p->sched_class->migrate_task_rq)
               p->sched_class->migrate_task_rq(p);
           p->se.nr_migrations++;
           perf_event_task_migrate(p);

           walt_fixup_busy_time(p, new_cpu);
       }

       __set_task_cpu(p, new_cpu);
}

/*
 * Called immediately before a task is migrated to a new cpu; task_cpu(p)
and
 * cfs_rq_of(p) references at time of call are still valid and identify the
 * previous cpu.  However, the caller only guarantees p->pi_lock is held; no
 * other assumptions, including the state of rq->lock, should be made.
 */
static void migrate_task_rq_fair(struct task_struct *p)
{
       /*
        * We are supposed to update the task to "current" time, then its up to
date
        * and ready to go to new CPU/cfs_rq. But we have difficulty in getting
        * what current time is, so simply throw away the out-of-date time. This
        * will result in the wakee task is less decayed, but giving the wakee
more
        * load sounds not bad.
        */
       /*重新计算在新cpu上的调度实体的负载*/
       remove_entity_load_avg(&p->se);
       /*重置新的调度实体的负载的最后更新时间和调度实体的执行时间*/
       /* Tell new CPU we are migrated */
       p->se.avg.last_update_time = 0;

       /* We have migrated, no longer consider this task hot */
       p->se.exec_start = 0;
}
/*
 * Synchronize entity load avg of dequeued entity without locking
 * the previous rq.
```

```
 */
void sync_entity_load_avg(struct sched_entity *se)
{
    struct cfs_rq *cfs_rq = cfs_rq_of(se);
    u64 last_update_time;

    last_update_time = cfs_rq_last_update_time(cfs_rq);
    __update_load_avg(last_update_time, cpu_of(rq_of(cfs_rq)), &se->avg, 0,
0, NULL);
}

/*
 * Task first catches up with cfs_rq, and then subtract
 * itself from the cfs_rq (task must be off the queue now).
 */
void remove_entity_load_avg(struct sched_entity *se)
{
    struct cfs_rq *cfs_rq = cfs_rq_of(se);

    /*
     * tasks cannot exit without having gone through wake_up_new_task() ->
     * post_init_entity_util_avg() which will have added things to the
     * cfs_rq, so we can remove unconditionally.
     *
     * Similarly for groups, they will have passed through
     * post_init_entity_util_avg() before unregister_sched_fair_group()
     * calls this.
     */

    sync_entity_load_avg(se);
    atomic_long_add(se->avg.load_avg, &cfs_rq->removed_load_avg);
    atomic_long_add(se->avg.util_avg, &cfs_rq->removed_util_avg);
}

```

3 How task p enqueue?

入队操作:ttwu_queue(p, cpu);

```
static void ttwu_queue(struct task_struct *p, int cpu)
{
    struct rq *rq = cpu_rq(cpu);

#if defined(CONFIG_SMP)
    if (sched_feat(TTWU_QUEUE) && !cpus_share_cache(smp_processor_id(),
cpu)) {
        sched_clock_cpu(cpu); /* sync clocks x-cpu */
        ttwu_queue_remote(p, cpu);
        return;
    }
#endif

    raw_spin_lock(&rq->lock);
    lockdep_pin_lock(&rq->lock);
    ttwu_do_activate(rq, p, 0);
    lockdep_unpin_lock(&rq->lock);
```

```
    raw_spin_unlock(&rq->lock);
}

static void
ttwu_do_activate(struct rq *rq, struct task_struct *p, int wake_flags)
{
    lockdep_assert_held(&rq->lock);

#ifdef CONFIG_SMP
    if (p->sched_contributes_to_load)
        rq->nr_uninterruptible--;
#endif

    ttwu_activate(rq, p, ENQUEUE_WAKEUP | ENQUEUE_WAKING);
    ttwu_do_wakeup(rq, p, wake_flags);
}

static inline void ttwu_activate(struct rq *rq, struct task_struct *p, int
en_flags)
{    /*实际入队的核心函数,同时更新task的vruntime并插入rb tree*/
    activate_task(rq, p, en_flags);
    p->on_rq = TASK_ON_RQ_QUEUED;

    /* if a worker is waking up, notify workqueue */
    if (p->flags & PF_WQ_WORKER)
        wq_worker_waking_up(p, cpu_of(rq));
}

/*
 * Mark the task runnable and perform wakeup-preemption.
 */
static void
ttwu_do_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
    check_preempt_curr(rq, p, wake_flags);
    /*修改task的状态为running状态,即task正在cpu上运行*/
    p->state = TASK_RUNNING;
    trace_sched_wakeup(p);

#ifdef CONFIG_SMP
    /*cfs没有定义*/
    if (p->sched_class->task_woken) {
        /*
         * Our task @p is fully woken up and running; so its safe to
         * drop the rq->lock, hereafter rq is only used for statistics.
         */
        lockdep_unpin_lock(&rq->lock);
        p->sched_class->task_woken(rq, p);
        lockdep_pin_lock(&rq->lock);
    }
    /*如果之前rq处于idle状态,即cpu处于idle状态,则修正rq的idle时间戳和判决idle时间*/
    if (rq->idle_stamp) {
        u64 delta = rq_clock(rq) - rq->idle_stamp;
        u64 max = 2*rq->max_idle_balance_cost;//max=1ms
```

```
        update_avg(&rq->avg_idle, delta);

        if (rq->avg_idle > max)
            rq->avg_idle = max;

        rq->idle_stamp = 0;
    }
#endif
}
```

3.1 核心函数activate_task分析

```
void activate_task(struct rq *rq, struct task_struct *p, int flags)
{   /*如果进程被置为TASK_UNINTERRUPTIBLE状态的话,减少处于uninterruptible状态的进程
   数量,因为当前是进程处于唤醒运行阶段*/
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;
    /*入队操作*/
    enqueue_task(rq, p, flags);
}

#define task_contributes_to_load(task)  \
                ((task->state & TASK_UNINTERRUPTIBLE) != 0 && \
                 (task->flags & PF_FROZEN) == 0 && \
                 (task->state & TASK_NOLOAD) == 0)

static inline void enqueue_task(struct rq *rq, struct task_struct *p, int
flags)
{
    update_rq_clock(rq);
    /*flag=0x3 & 0x10 = 0*/
    if (!(flags & ENQUEUE_RESTORE))
        /*更新task p入队的时间戳*/
        sched_info_queued(rq, p);
#ifdef CONFIG_INTEL_DWS //没有定义
    if (sched_feat(INTEL_DWS))
        update_rq_runnable_task_avg(rq);
#endif
    /*callback enqueue_task_fair函数*/
    p->sched_class->enqueue_task(rq, p, flags);
}
```

核心函数enqueue_task_fair解析如下:

```
/*
 * The enqueue_task method is called before nr_running is
 * increased. Here we update the fair scheduling stats and
 * then put the task into the rbtree:
 */
static void
enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
#ifdef CONFIG_SMP
    /*task_new = 0x3 & 0x20 = 0*/
```

```
    int task_new = flags & ENQUEUE_WAKEUP_NEW;
#endif
    /*进程p开始运行,并将进程p的运行时间累加到整个rq的cumulative_runnable_avg时间上,作为rq的负载值*/
    walt_inc_cumulative_runnable_avg(rq, p);

    /*
     * Update SchedTune accounting.
     *
     * We do it before updating the CPU capacity to ensure the
     * boost value of the current task is accounted for in the
     * selection of the OPP.
     *
     * We do it also in the case where we enqueue a throttled task;
     * we could argue that a throttled task should not boost a CPU,
     * however:
     * a) properly implementing CPU boosting considering throttled
     *    tasks will increase a lot the complexity of the solution
     * b) it's not easy to quantify the benefits introduced by
     *    such a more complex solution.
     * Thus, for the time being we go for the simple solution and boost
     * also for throttled RQs.
     */
    /*主要根据这个task的属性是否需要update task group的boost参数*/
    schedtune_enqueue_task(p, cpu_of(rq));

    /*
     * If in_iowait is set, the code below may not trigger any cpufreq
     * utilization updates, so do it here explicitly with the IOWAIT flag
     * passed.
     */
    /*如果进程p是一个iowait的进程,则进行cpu频率调整*/
    if (p->in_iowait)
        cpufreq_update_util(rq, SCHED_CPUFREQ_IOWAIT);

    for_each_sched_entity(se) {
        if (se->on_rq)
            break;
        cfs_rq = cfs_rq_of(se);
        walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
        enqueue_entity(cfs_rq, se, flags);

        /*
         * end evaluation on encountering a throttled cfs_rq
         *
         * note: in the case of encountering a throttled cfs_rq we will
         * post the final h_nr_running increment below.
         */
        if (cfs_rq_throttled(cfs_rq))
            break;
        cfs_rq->h_nr_running++;

        flags = ENQUEUE_WAKEUP;
    }
```

```
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        cfs_rq->h_nr_running++;
        walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);

        if (cfs_rq_throttled(cfs_rq))
            break;

        update_load_avg(se, UPDATE_TG);
        update_cfs_shares(se);
    }

    if (!se)
        add_nr_running(rq, 1);

#ifdef CONFIG_SMP
    if (!se) {
        struct sched_domain *sd;

        rcu_read_lock();
        sd = rcu_dereference(rq->sd);
        if (!task_new && sd) {
            if (cpu_overutilized(rq->cpu))
                set_sd_overutilized(sd);
            if (rq->misfit_task && sd->parent)
                set_sd_overutilized(sd->parent);
        }
        rcu_read_unlock();
    }

#endif /* CONFIG_SMP */
    hrtick_update(rq);
}
```
后面的流程与新创建进程的流程一致了。