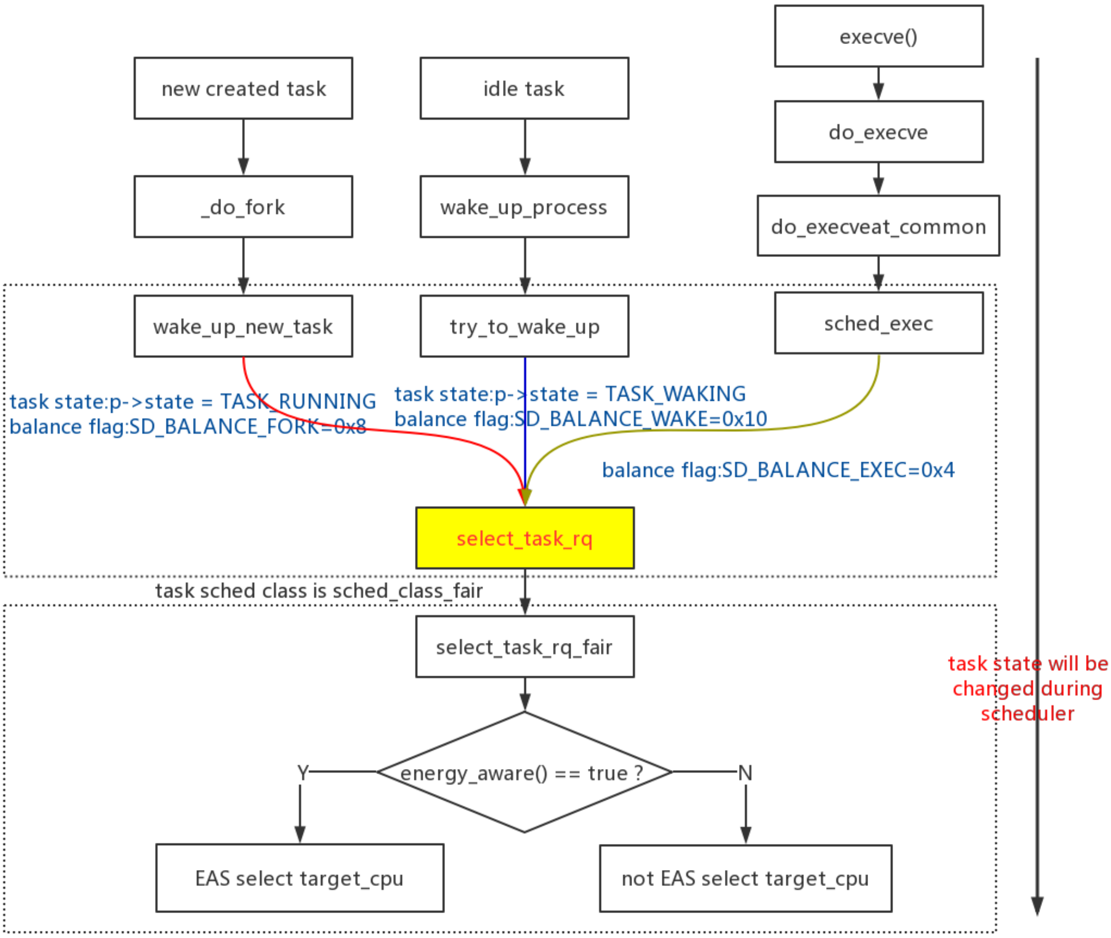
select_energy_cpu_brute函数分析

一、概述

之前在讲解新创建进程和idle进程被wakeup之后如何被调度器调度的原理，有两个点没有分析的很清楚，就是在这个调度的过程中，如何选择一个cpu来执行调度实体。现在单独拎出来详细分析：

- 如果EAS feature启用的话，执行函数流：select_energy_cpu_brute
- 如果EAS feature没有启用的话，执行传统的函数流：find_idlest_cpu

简单的流程图如下：



现在仅仅分析第一个部分，使用EAS如何来选择执行调度实体的cpu？

好，现在直接来分析select_energy_cpu_brute(p, prev_cpu, sync)函数的原理：

- p是需要调度的进程task
- prev_cpu是唤醒当前执行此进程的cpu
- sync = wake_flags & WF_SYNC = 0x0 & WF_SYNC = 0

1 energy_aware()含义

energy_aware()函数的定义如下：

```
static inline bool energy_aware(void)
{
    return sched_feat(ENERGY_AWARE);
}
```

经常在调度器的代码里面看到sched_feat，从字面意思看是调度特性，sched feature，即调度器根据不同的字段，执行不同的调度流程。

```
extern const_debug unsigned int sysctl_sched_features;

#define SCHED_FEAT(name, enabled)    \
    __SCHED_FEAT_##name ,

enum {
#include "features.h"
```

```c
    __SCHED_FEAT_NR,
};

#undef SCHED_FEAT

#if defined(CONFIG_SCHED_DEBUG) && defined(HAVE_JUMP_LABEL)
#define SCHED_FEAT(name, enabled)                    \
static __always_inline bool static_branch_##name(struct static_key *key) \
{                              \
    return static_key_##enabled(key);               \
}

#include "features.h"

#undef SCHED_FEAT
/* 下面是定义sched_feat */
extern struct static_key sched_feat_keys[__SCHED_FEAT_NR];
#define sched_feat(x)
(static_branch_##x(&sched_feat_keys[__SCHED_FEAT_##x]))
#else /* !(SCHED_DEBUG && HAVE_JUMP_LABEL) */
#define sched_feat(x) (sysctl_sched_features & (1UL << __SCHED_FEAT_##x))
#endif /* SCHED_DEBUG && HAVE_JUMP_LABEL */

extern struct static_key sched_feat_keys[__SCHED_FEAT_NR];
#define sched_feat(x)
(static_branch_##x(&sched_feat_keys[__SCHED_FEAT_##x]))
#else /* !(SCHED_DEBUG && HAVE_JUMP_LABEL) */
#define sched_feat(x) (sysctl_sched_features & (1UL << __SCHED_FEAT_##x))
#endif /* SCHED_DEBUG && HAVE_JUMP_LABEL */


/*
 * Energy aware scheduling. Use platform energy model to guide scheduling
 * decisions optimizing for energy efficiency.
 */  /* 如果使用EAS的话，则调度特性为true，否则为false */
#ifdef CONFIG_DEFAULT_USE_ENERGY_AWARE
SCHED_FEAT(ENERGY_AWARE, true)
#else
SCHED_FEAT(ENERGY_AWARE, false)
#endif
```

重点是分析下面的内容。

select_energy_cpu_brute函数源码如下：

```c
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;

    schedstat_inc(p, se.statistics.nr_wakeups_secb_attempts);
    schedstat_inc(this_rq(), eas_stats.secb_attempts);

    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();
#ifdef CONFIG_CGROUP_SCHEDTUNE
    boosted = schedtune_task_boost(p) > 0;
    prefer_idle = schedtune_prefer_idle(p) > 0;
#else
    boosted = get_sysctl_sched_cfs_boost() > 0;
    prefer_idle = 0;
#endif

    sync_entity_load_avg(&p->se);

    sd = rcu_dereference(per_cpu(sd_ea, prev_cpu));
    /* Find a cpu with sufficient capacity */
    tmp_target = find_best_target(p, &tmp_backup, boosted, prefer_idle);

    if (!sd)
        goto unlock;
    if (tmp_target >= 0) {
        target_cpu = tmp_target;
        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;
        }
    }

    if (target_cpu == prev_cpu && tmp_backup >= 0) {
        target_cpu = tmp_backup;
        tmp_backup = -1;
    }

    if (target_cpu != prev_cpu) {
        int delta = 0;
        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)
```

```
66.             delta = task_util(p);
67. #endif
68.         /* Not enough spare capacity on previous cpu */
69.         if (__cpu_overutilized(prev_cpu, delta, p)) {
70.             if (tmp_backup >= 0 &&
71.                 capacity_orig_of(tmp_backup) <
72.                     capacity_orig_of(target_cpu))
73.                 target_cpu = tmp_backup;
74.             schedstat_inc(p, se.statistics.nr_wakeups_secb_insuff_cap);
75.             schedstat_inc(this_rq(), eas_stats.secb_insuff_cap);
76.             goto unlock;
77.         }
78.
79.         if (energy_diff(&eenv) >= 0) {
80.             /* No energy saving for target_cpu, try backup */
81.             target_cpu = tmp_backup;
82.             eenv.dst_cpu = target_cpu;
83.             eenv.trg_cpu = target_cpu;
84.             if (tmp_backup < 0 ||
85.                 tmp_backup == prev_cpu ||
86.                 energy_diff(&eenv) >= 0) {
87.                 schedstat_inc(p, se.statistics.nr_wakeups_secb_no_nrg_sav);
88.                 schedstat_inc(this_rq(), eas_stats.secb_no_nrg_sav);
89.                 target_cpu = prev_cpu;
90.                 goto unlock;
91.             }
92.         }
93.
94.         schedstat_inc(p, se.statistics.nr_wakeups_secb_nrg_sav);
95.         schedstat_inc(this_rq(), eas_stats.secb_nrg_sav);
96.         goto unlock;
97.     }
98.
99.     schedstat_inc(p, se.statistics.nr_wakeups_secb_count);
100.     schedstat_inc(this_rq(), eas_stats.secb_count);
101.
102. unlock:
103.     rcu_read_unlock();
104.
105.     return target_cpu;
106. }
```

分如下两个部分来分析：

1. 3~33行代码，核心函数是find_best_target(p, &tmp_backup, boosted, prefer_idle)
2. 35~92行代码，核心结构体struct energy_env和核心函数energy_diff(struct energy_env)。

下面详细分析两个部分。

二、核心函数find_best_target的分析

列出需要分析的代码如下：

```
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;

    schedstat_inc(p, se.statistics.nr_wakeups_secb_attempts);
    schedstat_inc(this_rq(), eas_stats.secb_attempts);

    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();
#ifdef CONFIG_CGROUP_SCHEDTUNE
    boosted = schedtune_task_boost(p) > 0;
    prefer_idle = schedtune_prefer_idle(p) > 0;
#else
    boosted = get_sysctl_sched_cfs_boost() > 0;
    prefer_idle = 0;
#endif

    sync_entity_load_avg(&p->se);

    sd = rcu_dereference(per_cpu(sd_ea, prev_cpu));
    /* Find a cpu with sufficient capacity */
    tmp_target = find_best_target(p, &tmp_backup, boosted, prefer_idle);
```

2.1 非find_best_target部分

看看这部分代码：

```
    struct sched_domain *sd;
    int target_cpu = prev_cpu, tmp_target, tmp_backup;
    bool boosted, prefer_idle;

    schedstat_inc(p, se.statistics.nr_wakeups_secb_attempts);
    schedstat_inc(this_rq(), eas_stats.secb_attempts);
    /*我们知道sync=0，而且sys_ctrl_sched_sync_hint_enable对于ARM平台是没有设置的，
    默认为0*/
    if (sysctl_sched_sync_hint_enable && sync) {
        int cpu = smp_processor_id();
        /*比较唤醒进程P的cpu是否在进程cpus_allowed的cpumask里面*/
        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();

#ifdef CONFIG_CGROUP_SCHEDTUNE
    boosted = schedtune_task_boost(p) > 0;
    prefer_idle = schedtune_prefer_idle(p) > 0;
#else
    boosted = get_sysctl_sched_cfs_boost() > 0;
    prefer_idle = 0;
#endif

    sync_entity_load_avg(&p->se);

    sd = rcu_dereference(per_cpu(sd_ea, prev_cpu));
```
下面的代码含义：

boosted = schedtune_task_boost(p) > 0;

prefer_idle = schedtune_prefer_idle(p) > 0;

涉及到进程的可tunable属性,

1. boosted_value是在task_group里面对进程负载的补偿，即在当前进程的负载情况下，补偿boosted_value/100 *util的数值，boosted_value∈[-100,100]，会对频率有影响，具体怎么使用的可以参考：cpu util怎么计算的，我们知道可能每个task都存在一个可以tunable的boosted_value，而cpu_util取的boosted_value是所有task里面最大的boosted_value。一般对于top-app为了提升性能，会设置这个参数。看各个平台的设定
2. prefer_idle是一个进程在选择cpu运行时候可以tunable的参数，目的尽可能的选择idle cpu，一般不会设置。也是看各个平台自身的设定

下面这段代码的含义：

sync_entity_load_avg(&p->se);

OK，看其自身实现如下：
```
/*
 * 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;
        /*获取上次更新cfs_rq利用率的时间戳*/
    last_update_time = cfs_rq_last_update_time(cfs_rq);

        /*使用PELT算法，来衰减和累加各个周期的负载。*/
    __update_load_avg(last_update_time, cpu_of(rq_of(cfs_rq)), &se->avg, 0,
0, NULL);
}
```
分析如下：

1. last_update_time是上次更新调度实体负载的时间戳，保存在struct sched_entity-->struct sched_avg--->last_update_time中，这个数值都会在函数__update_load_avg中update
2. __update_load_avg函数是根据PELT算法(PELT算法实现原理)来衰减和累加此调度实体的负载，最终得到当前时刻的负载，并将load tracing的各个数值保存在

sched_entity-->struct sched_avg，这是用来追踪调度实体的负载，当前cfs_rq本身也有自己的负载追踪结构体，也是struct sched_avg。

下面分析下面的代码:

sd = rcu_dereference(per_cpu(sd_ea, prev_cpu));

这个比较有意思。需要知道sd_ea是什么？

- `DECLARE_PER_CPU(struct sched_domain *, sd_ea);`

OK，是一个调度域结构体指针，per_cpu就是获取唤醒此进程p的调度域，至于调度域/调度组怎么和cpu关联的可以参考下面的文章：调度域调度组的建立和初始化。下面开始分析第一部分的核心函数了。

2.2 find_best_target部分

其源码如下：

```
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;
    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;
    struct root_domain *rd = cpu_rq(smp_processor_id())->rd;
    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 */
    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 */
    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,backup_idle_cpu);
                    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)
```

```
200.          *
201.          * - NON prefer_idle tasks:
202.          *
203.          *    a) ACTIVE CPU: target_cpu
204.          *    b) IDLE CPU: best_idle_cpu
205.          */
206.         if (target_cpu == -1) {
207.             if (best_idle_cpu != -1)
208.                 target_cpu = best_idle_cpu;
209.             else
210.                 target_cpu = (backup_idle_cpu != -1)
211.                 ? backup_idle_cpu
212.                 : best_active_cpu;
213.         } else
214.             *backup_cpu = best_idle_cpu;
215.
216.         trace_sched_find_best_target(p, prefer_idle, min_util, cpu,
217.                          best_idle_cpu, best_active_cpu,
218.                          target_cpu,backup_idle_cpu);
219.
220.         schedstat_inc(p, se.statistics.nr_wakeups_fbt_count);
221.         schedstat_inc(this_rq(), eas_stats.fbt_count);
222.
223.         return target_cpu;
224.     }
```

分如下几个部分来解析上面的源码：

2.2.1 20~40行之间的源码

```
/*获取当前cpu调度域的根节点，并获取它的最大capacity能力*/
       struct root_domain *rd = cpu_rq(smp_processor_id())->rd;
unsigned long max_cap = rd->max_cpu_capacity.val;
       /*关键参数变量1*/
*backup_cpu = -1;

schedstat_inc(p, se.statistics.nr_wakeups_fbt_attempts);
schedstat_inc(this_rq(), eas_stats.fbt_attempts);
       /*找到最小capacity的cpu，此时cpu=0*/
/* Find start CPU based on boost value */
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;
}
       /*根据per_cpu变量获取start_cpu的调度域*/
/* Find SD for the start CPU */
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;
}
```

接着往下分析

2.2.2 45~184行之间的源码，即do{}while()循环

一步一步来分析。

- do{}while()循环是对sched domain(start_cpu所在的调度域)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+新进程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.为何new_util=max(min_util,new_util),目的是新的util必须满足新唤醒进程p的boost请求。并且util的数值是1024刻度化来的。[cpu_util和task_util怎么计算的]。

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71~181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. 如果遍历的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,没必须继续执行，因为不可能选择此cpu作为target_cpu..
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;

```

从上面的的循环迭代来看，在整个的循环阶段，目的是更新下面四个cpu index参数：

- backup_idle_cpu
- best_active_cpu
- best_idle_cpu
- target_cpu

2.2.3 剩下的源码分析

如何根据2.2.2节的四个关键变量选择target_cpu和backup_cpu
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。

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
2. 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

上面的思路简单概况为首选new_util < capacity_orig && idle的cpu,其次才选择new_util > capacity_orig && idle的cpu，最后才选择new_util > capacity_orig && active的cpu
如果target_cpu!=-1

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.
最后概况选择cpu的优先级,如果存在这种cpu的话：

1. **new_util < capacity_orig && active的cpu，选择capacity余量最大的cpu作为target_cpu**
2. **new_util < capacity_orig && idle的cpu**
3. **new_util > capacity_orig && idle的cpu，基本不会存在这样的cpu**
4. **new_util > capacity_orig && active的cpu**

下面附件是我抓取的trace文件：trace文件 里面抓取了上面四类cpu的数值是如何改变的。添加的patch如下：

```
diff --git a/include/trace/events/sched.h b/include/trace/events/sched.h
index 31904e2..21271d8e 100644
--- a/include/trace/events/sched.h
+++ b/include/trace/events/sched.h
@@ -927,41 +927,43 @@ TRACE_EVENT(sched_boost_task,
  */
 TRACE_EVENT(sched_find_best_target,

-    TP_PROTO(struct task_struct *tsk, bool prefer_idle,
-        unsigned long min_util, int start_cpu,
-        int best_idle, int best_active, int target),
+    TP_PROTO(struct task_struct *tsk, unsigned long wake_util,
+        unsigned long new_util, int i,
+        int best_idle, int best_active, int target,int backup_idle),

-    TP_ARGS(tsk, prefer_idle, min_util, start_cpu,
-        best_idle, best_active, target),
+    TP_ARGS(tsk, wake_util, new_util, i,
+        best_idle, best_active, target, backup_idle),

     TP_STRUCT__entry(
         __array( char,   comm,    TASK_COMM_LEN    )
         __field( pid_t, pid            )
-        __field( unsigned long, min_util     )
-        __field( bool,   prefer_idle       )
-        __field( int,    start_cpu         )
+        __field( unsigned long, wake_util    )
+        __field( unsigned long, new_util         )
+        __field( int,            i         )
         __field( int,    best_idle         )
         __field( int,    best_active       )
         __field( int,    target            )
+        __field( int,    backup_idle           )
     ),

     TP_fast_assign(
         memcpy(__entry->comm, tsk->comm, TASK_COMM_LEN);
         __entry->pid      = tsk->pid;
-        __entry->min_util     = min_util;
-        __entry->prefer_idle = prefer_idle;
-        __entry->start_cpu    = start_cpu;
+        __entry->wake_util    = wake_util;
+        __entry->new_util     = new_util;
+        __entry->i    = i;
         __entry->best_idle    = best_idle;
```

```
        __entry->best_active = best_active;
        __entry->target      = target;
+       __entry->backup_idle     = backup_idle;
    ),

-   TP_printk("pid=%d comm=%s prefer_idle=%d start_cpu=%d "
-         "best_idle=%d best_active=%d target=%d",
+    TP_printk("pid=%d comm=%s wake_util=%lu new_util=%lu i=%d "
+            "best_idle=%d best_active=%d target=%d backup_idle=%d",
        __entry->pid, __entry->comm,
-       __entry->prefer_idle, __entry->start_cpu,
+       __entry->wake_util, __entry->new_util, __entry->i,
        __entry->best_idle, __entry->best_active,
-       __entry->target)
+       __entry->target,__entry->backup_idle)
 );

 /*
diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
index 6a02fd7..54ba7f7 100644
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -6629,6 +6629,9 @@ static inline int find_best_target(struct task_struct *p, int
*backup_cpu,
            wake_util = cpu_util_wake(i, p);
            new_util = wake_util + task_util(p);

+           trace_sched_find_best_target(p, wake_util, new_util, i,
+                   best_idle_cpu, best_active_cpu,
+                   target_cpu,backup_idle_cpu);
            /*
             * Ensure minimum capacity to grant the required boost.
             * The target CPU can be already at a capacity level higher
@@ -6706,10 +6709,10 @@ skip:

                if (prefer_idle ||
                    cpumask_test_cpu(i, &min_cap_cpu_mask)) {
-                   trace_sched_find_best_target(p,
+               /*  trace_sched_find_best_target(p,
                        prefer_idle, min_util, cpu,
                        best_idle_cpu, best_active_cpu,
-                       i);
+                       i);*/
                    return i;
                }
                idle_idx = idle_get_state_idx(cpu_rq(i));
@@ -6780,10 +6783,10 @@ skip:
    } else
        *backup_cpu = best_idle_cpu;

-   trace_sched_find_best_target(p, prefer_idle, min_util, cpu,
+/* trace_sched_find_best_target(p, prefer_idle, min_util, cpu,
                   best_idle_cpu, best_active_cpu,
                   target_cpu);
```

```
-
+*/
    schedstat_inc(p, se.statistics.nr_wakeups_fbt_count);
    schedstat_inc(this_rq(), eas_stats.fbt_count);
```

三、根据能效选择目标cpu

对下面源码进行分析：

```
static int select_energy_cpu_brute(struct task_struct *p, int prev_cpu, int sync)
{
.....................
    if (!sd)
        goto unlock;
    if (tmp_target >= 0) {
        target_cpu = tmp_target;
        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;
        }
    }

    if (target_cpu == prev_cpu && tmp_backup >= 0) {
        target_cpu = tmp_backup;
        tmp_backup = -1;
    }

    if (target_cpu != prev_cpu) {
        int delta = 0;
        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)
            delta = task_util(p);
#endif
        /* Not enough spare capacity on previous cpu */
        if (__cpu_overutilized(prev_cpu, delta, p)) {
            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;
            }

            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;
}
```

分两部分来讲解：

3.1 非energy_diff函数分析

一步一步来分析：

```
if (!sd)
    goto unlock;
if (tmp_target >= 0) {
    target_cpu = tmp_target;
    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;
    }
}
```

如果通过函数find_best_target函数寻找的target_cpu！=-1，如果进程符合如下两个条件，则target_cpu就是我们需要的target_cpu，直接结束代码流程：

1. boosted || prefer_idle，prefer_idle一般为false，boosted根据tunable设定来决定，并且此target_cpu是idle cpu

2. target_cpu的capacity是最小的cpu

只有满足上面两点，则target_cpu就是实际需要的cpu，比如top-app起boosted就为true：

```
s9863a1h10:/dev/stune/top-app # cat schedtune.boost
10
```

接下来分析：

```
/*如果函数find_best_target的返回值为唤醒进程P的cpu，并且best_idle_cpu!=-1,则修改
   目标cpu为best_idle_cpu,但是并没有直接return，而是还是需要判决prev_cpu和
   target_cpu的能效*/
if (target_cpu == prev_cpu && tmp_backup >= 0) {
    target_cpu = tmp_backup;
    tmp_backup = -1;
}

if (target_cpu != prev_cpu) {
    int delta = 0;
    /*初始化energy_env结构体，重点关注src_cpu和dst_cpu分别是唤醒进程P的cpu和当前
   打算让进程P运行的cpu*/
    struct energy_env eenv = {
        .util_delta     = task_util(p),/*进程P的util*/
        .src_cpu        = prev_cpu,
        .dst_cpu        = target_cpu,
        .task           = p,  /*进程P*/
        .trg_cpu    = target_cpu,
    };
```

struct energy_env结构体如下，后面在energy_diff会使用到：

```
struct energy_env {
    struct sched_group  *sg_top;
    struct sched_group  *sg_cap;
    int         cap_idx;
    int         util_delta;
    int         src_cpu;
    int         dst_cpu;
    int         trg_cpu;
    int         energy;
    int         payoff;
    struct task_struct  *task;
    struct {
        int before;
        int after;
        int delta;
        int diff;
    } nrg;
    struct {
        int before;
        int after;
        int delta;
    } cap;
};
```

接着继续分析：

```
#ifdef CONFIG_SCHED_WALT
        if (!walt_disabled && sysctl_sched_use_walt_cpu_util &&
            p->state == TASK_WAKING)
            delta = task_util(p);
```

```
#endif
```

对于使用WALT算法来作为进程和cpu负载的计算方式，本平台上默认使用WALT，也就是说，walt_disable=false，sysctrl_sched_use_walt_cpu_util=1，如果当前进程P是从idle被wakeup的进程，则此时的进程P的状态为TASK_WAKING，则计算进程p的util并赋值给delta。

继续代码流程分析：

```
/* Not enough spare capacity on previous cpu */
if (__cpu_overutilized(prev_cpu, delta, p)) {
    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;
}
```

__cpu_overutilized函数是当前cpu的util+delta的总的负载大于此cpu capacity能力的90%以上，此cpu就是overutilized。这就会导致调度器优先选择capacity小的best_idle_cpu作为目标cpu并直接return。

3.2 核心函数energy_diff分析

energy_diff函数是函数select_energy_cpu_brute的第二个核心函数。

```
static inline int
energy_diff(struct energy_env *eenv)
{
    int boost = schedtune_task_boost(eenv->task);
    int nrg_delta;

    /* 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有下面三个核心函数如下：

1. __energy_diff
2. normalize_energy
3. schedtune_accept_deltas

下面分别来分析。

3.2.1 核心函数__energy_diff分析

看下这个函数实现如下：

```
/*
 * 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;
    /*构造迁移前的能效环境变量*/
    struct energy_env eenv_before = {
        /* enev结构体已经在3.1节初始化了，这里直接赋值 */
        .util_delta = task_util(eenv->task),
        .src_cpu    = eenv->src_cpu,
        .dst_cpu    = eenv->dst_cpu,
        /* 这个参数并不是初始化的数值，初始化的数值为eenv->dst_cpu,这里修改的目的是
         ????? */
        .trg_cpu    = eenv->src_cpu,
        .nrg        = { 0, 0, 0, 0},
        .cap        = { 0, 0, 0 },
        .task       = eenv->task,
    };
/* 在初始化eenv结构体的时候已经做了判决，为何还需要再次做判断呢???? */
    if (eenv->src_cpu == eenv->dst_cpu)
```

```
        return 0;

    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;

    do {
        if (cpu_in_sg(sg, eenv->src_cpu) || cpu_in_sg(sg, eenv->dst_cpu)) {
            /* 设置迁移前的sg_top为当前遍历的调度组作为调度组的top层（MC层级） */
            eenv_before.sg_top = eenv->sg_top = sg;

            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;

            if (sched_group_energy(eenv))
                return 0; /* Invalid result abort */
            energy_after += eenv->energy;
        }
    } while (sg = sg->next, sg != sd->groups);

    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;
}
```

3.2.1.1 如何计算调度组的能效

理解清楚核心函数sched_group_energy的逻辑，先理解下面三个重要函数：

1. find_new_capacity
2. group_idle_state
3. group_norm_util

下面单独分析上面的函数

1. find_new_capacity:

必须清楚的是，结构体sched_group_energy的成员变量在energy.c文件从dts里面就已经初始化的常量数据：能效的获取

```
static int find_new_capacity(struct energy_env *eenv,
    const struct sched_group_energy * const sge)
{   /* max_idx是最大消耗power的idx */
    int idx, max_idx = sge->nr_cap_states - 1;
    /* 得到eenv->sg_cap里面cpu最大的util */
    unsigned long util = group_max_util(eenv);

    /* default is max_cap if we don't find a match */
    eenv->cap_idx = max_idx;
    /* 获取idx数值，即在数组cap_states里面选择能力数值大于util的第一个下标
    这个能力数值表示不同频率不同能力，具体可以看能效的获取是怎么回事*/
    for (idx = 0; idx < sge->nr_cap_states; idx++) {
        if (sge->cap_states[idx].cap >= util) {
            eenv->cap_idx = idx;
            break;
        }
    }
    /* 返回符合要求的最低能效idx */
    return eenv->cap_idx;
}

 /* 获取此rq在一个walt window的util数值并刻度化为1024，即*1024*/
static inline u64 walt_cpu_util_cum(int cpu)
{
    return div64_u64(cpu_rq(cpu)->cum_window_demand,
             walt_ravg_window >> SCHED_CAPACITY_SHIFT);
}
/* 判断此进程P是否在rq的累加运行时间里面 */
static inline bool
walt_task_in_cum_window_demand(struct rq *rq, struct task_struct *p)
{
    return cpu_of(rq) == task_cpu(p) &&
           (p->on_rq || p->last_sleep_ts >= rq->window_start);
}
/* 其实就是获取调度组内各个cpu的util，并返回最大的util */
static unsigned long group_max_util(struct energy_env *eenv)
{
    unsigned long max_util = 0;
    unsigned long util;
    int cpu;

    for_each_cpu(cpu, sched_group_cpus(eenv->sg_cap)) {
```

```
        /* 根据进程P状态和是否启用WALT，来计算util数值，是否是本cpu自身的util，还是
         需要减去进程P的util */
        util = cpu_util_energy_wake(cpu, eenv->task);

        /*
         * If we are looking at the target CPU specified by the eenv,
         * then we should add the (estimated) utilization of the task
         * assuming we will wake it up on that CPU.
         */
        if (unlikely(cpu == eenv->trg_cpu))
            util += eenv->util_delta;
        /* 得到eenv->sg_cap cpumask 里面util最大的cpu_util */
        max_util = max(max_util, util);
    }

    return max_util;
}

static unsigned long cpu_util_energy_wake(int cpu, struct task_struct *p)
{
    unsigned long util, capacity;
    /* 如果enable WALT计算util，则执行 */
    if (!walt_disabled && sysctl_sched_use_walt_cpu_util) {
        util = walt_cpu_util_cum(cpu); /* 按照WALT计算此cpu所在的rq util */
        /* 如果进程P运行在当前cpu的rq里面，则util需要减去进程P的util */
        if (walt_task_in_cum_window_demand(cpu_rq(cpu), p))
            util -= task_util(p);
        /*
         * walt_cpu_util_cum(cpu) is always equal or greater than
         * task_util(p) when p is in cumulative window demand.
         */  /* 获得此cpu的capacity */
        capacity = capacity_orig_of(cpu);
        /* 保证util∈[0,1024]里面 */
        return (util >= capacity) ? capacity : util;
    }

    /* cpu_util_wake() already set ceiling to capacity_orig_of() */
    return cpu_util_wake(cpu, p);
}


/*
 * cpu_util_wake: Compute cpu utilization with any contributions from
 * the waking task p removed.  check_for_migration() looks for a better CPU
of
 * rq->curr. For that case we should return cpu util with contributions from
 * currently running task p removed.
 */
static int cpu_util_wake(int cpu, struct task_struct *p)
{
    unsigned long util, capacity;

#ifdef CONFIG_SCHED_WALT
    /*
```

```
         * WALT does not decay idle tasks in the same manner
         * as PELT, so it makes little sense to subtract task
         * utilization from cpu utilization. Instead just use
         * cpu_util for this case.
         */ /* enable WALT并且进程是从idle wakeup起来的进程，则返回当前遍历的cpu的
        cpu_util */
        if (!walt_disabled && sysctl_sched_use_walt_cpu_util &&
            p->state == TASK_WAKING)
            return cpu_util(cpu);
    #endif
        /* Task has no contribution or is new */
        /* 如果disable WALT，并且进程P所在的cpu不是当前遍历的cpu上或者进程p的load
        tracing从来没有更新过，即是一个全新的进程（fork/clone?） */
        if (cpu != task_cpu(p) || !p->se.avg.last_update_time)
            return cpu_util(cpu);
        /* 最后就算一般情况下的util，即正常的迁移，当前进程P已经在当前遍历的cpu上运行了，所以
        需要减掉这部分的util */
        capacity = capacity_orig_of(cpu);
        util = max_t(long, cpu_util(cpu) - task_util(p), 0);

        return (util >= capacity) ? capacity : util;
    }
```

下面画一张图来说明上面是如何查找到在调度组内哪一个cpu的能效刚刚好大于cpu的util的：find_new_capacity就是查找下面的索引：比如CPU_COST_0的busy-cost-data数组大小为5, cap_states[idx].cap = 501/576/652…..之类，而cap_states[idx].power=80/101/125….之类的。

```
    energy-costs {
        CPU_COST_0: core-cost0 {
            busy-cost-data = <
                501 80  /* 768MHz */
                576 101 /* 884MHz */
                652 125 /* 1000MHz */
                717 151 /* 1100MHz */
                782 181 /* 1200MHz */
            >;
            idle-cost-data = <
                25      /* ACTIVE-IDLE */
                25      /* WFI */
                0       /* CORE_PD */
            >;
        };
        CPU_COST_1: core-cost1 {
            busy-cost-data = <
                501 110 /* 768MHz */
                685 160 /* 1050MHz */
                799 206 /* 1225MHz */
                913 258 /* 1400MHz */
                978 305 /* 1500MHz */
                1024    352 /* 1570MHz */
            >;
            idle-cost-data = <
                44      /* ACTIVE-IDLE */
                44      /* WFI */
                0       /* CORE_PD */
```

```
        >;
    };
```

2. group_idle_state

```
static int group_idle_state(struct energy_env *eenv, struct sched_group *sg)
{
    int i, state = INT_MAX;

    /* Find the shallowest idle state in the sched group. */
    for_each_cpu(i, sched_group_cpus(sg))
        state = min(state, idle_get_state_idx(cpu_rq(i)));

    /* Take non-cpuidle idling into account (active idle/arch_cpu_idle()) */
    state++;

    return state;
}
```

需要注意的一点就是state++为何要加1，就是需要考虑idle的空转。

3. group_norm_util

```
/*
 * group_norm_util() returns the approximated group util relative to it's
 * current capacity (busy ratio), in the range [0..SCHED_LOAD_SCALE], for use
 * in energy calculations.
 *
 * Since task executions may or may not overlap in time in the group the true
 * normalized util is between MAX(cpu_norm_util(i)) and SUM(cpu_norm_util(i))
 * when iterating over all CPUs in the group.
 * The latter estimate is used as it leads to a more pessimistic energy
 * estimate (more busy).
 */
static unsigned
long group_norm_util(struct energy_env *eenv, struct sched_group *sg)
{
    unsigned long capacity = sg->sge->cap_states[eenv->cap_idx].cap;
    unsigned long util, util_sum = 0;
    int cpu;

    for_each_cpu(cpu, sched_group_cpus(sg)) {
        /*获取此调度组内所有cpu的cpu_util数值（根据进程相应状态减去进程的util数值）*/
        util = cpu_util_energy_wake(cpu, eenv->task);

        /*
         * If we are looking at the target CPU specified by the eenv,
         * then we should add the (estimated) utilization of the task
         * assuming we will wake it up on that CPU.
         */
        if (unlikely(cpu == eenv->trg_cpu))
            util += eenv->util_delta;
```

```
            /* __cpu_norm_util获取相对负载，即在调度组内的各个cpu的util相对于最大capacity的比例，之和<< 10刻度化1024。*/
            util_sum += __cpu_norm_util(util, capacity);
        }
    /* 返回不超过1024的数值 */
     return min_t(unsigned long, util_sum, SCHED_CAPACITY_SCALE);
}

/*
 * __cpu_norm_util() returns the cpu util relative to a specific capacity,
 * i.e. it's busy ratio, in the range [0..SCHED_LOAD_SCALE], which is useful for
 energy calculations.
 *
 * Since util is a scale-invariant utilization defined as:
 *
 *   util ~ (curr_freq/max_freq)*1024 * capacity_orig/1024 * running_time/time
 *
 * the normalized util can be found using the specific capacity.
 *
 *   capacity = capacity_orig * curr_freq/max_freq
 *
 *   norm_util = running_time/time ~ util/capacity
 */
static unsigned long __cpu_norm_util(unsigned long util, unsigned long capacity)
{
    if (util >= capacity)
        return SCHED_CAPACITY_SCALE;

    return (util << SCHED_CAPACITY_SHIFT)/capacity;
}
```

至此上面三个函数解析完毕

1. find_new_capacity：获取调度组内所有cpu的util数值，并查表返回第一个大于max_util的索引，索引cap_idx对应的数值为capacity=sg->sge->cap_states[eenv->cap_idx].cap
2. group_idle_state：获取调度组内idle最浅的idle索引
3. group_norm_util：获取调度组内各个cpu的util相对于capacity的归一化累加数值

上面三个函数比较好理解。

好接下来继续分析sched_group_energy函数源码：

```
/*
 * 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);
    /* 将调度组包括的cpu mask赋值给visit_cpus变量 */
    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.
     */
    /* 计算visit_cpus包括多少个cpu. */
    cpu_count = cpumask_weight(&visit_cpus);
    /* visit_cpus cpumask不能为空，后面会每遍历一次都会将遍历过的cpu从visit_cpus
    中clear掉 */
    while (!cpumask_empty(&visit_cpus)) {
        struct sched_group *sg_shared_cap = NULL;
        /* 选择visit_cpus cpumask第一bitmap */
        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_scs per_cpu变量在函数update_top_cache_domain实现的，关键点
         是调度域里面的调度组是否有调度组的能效信息(struct sched_group_energy) */
        sd = rcu_dereference(per_cpu(sd_scs, cpu));
        /* 调度域存在并且此调度域不是top调度域，则将调度域的parent的调度组赋值给
         shared_cap变量
        if (sd && sd->parent)
            sg_shared_cap = sd->parent->groups;
        /* 从SDTL底层level开始逐个遍历调度域
        for_each_domain(cpu, sd) {
            struct sched_group *sg = sd->groups;
            /* 如果是顶层调度域，那么只有一个调度组，只会调用一次 */
            /* Has this sched_domain already been visited? */
            if (sd->child && group_first_cpu(sg) != cpu)
                break;
            /* 逐个遍历该层次的sg链表 */
            do {
                unsigned long group_util;
                int sg_busy_energy, sg_idle_energy;
                int cap_idx, idle_idx;
```

```
            /* group_weight是调度组包含的cpu数量 */
            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) */
                /* 由于在__eenv_diff函数会调用两次sched_group_energy函数
                 ，就在与计算before和after两个eenv结构体（before和after的eenv
                 结构体是不相同的），后面详细分析 */
                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;
                }
            }
            /* 查找出sg内所有cpu最浅的idle状态索引 */
            idle_idx = group_idle_state(eenv, sg);
            /* 累加sg内所有cpu的相对于
             sg->sge->cap_states[eenv->cap_idx].cap负载之和 */
            group_util = group_norm_util(eenv, sg);
            /* 计算这个调度组的busy和idle energy，比较有意思 */
            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;
            /* 调度域不是顶层SDTL，那么其child必然为NULL */
            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，则将sg里面的cpu从visit_cpus中去掉 */
                    cpumask_xor(&visit_cpus, &visit_cpus,
                                   sched_group_cpus(sg));
                    cpu_count--;
                }
                /* 如果遍历了cpu的底层到顶层的sd，那么将此cpu从visit_cpus中clear掉
                          */
                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:
        cpumask_clear_cpu(cpu, &visit_cpus);
        continue;
    }
    /* 将最后的energy的数值赋值给eenv->energy成员变量 */
    eenv->energy = total_energy >> SCHED_CAPACITY_SHIFT;
    return 0;
}
```

上面分析完毕sched_group_energy即计算一个调度组内cpu的energy数值，包括idle和active的energy之和。

3.2.1.2 如何计算after和before的能效差值

通过sched_group_energy获取进程在src_cpu和在dst_cpu的能效数值，下面继续分__energy_diff函数：

```
/*
 * 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;

    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_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;
    /* 遍历整个调度组，计算前后能效之和。遍历完毕之和计算差值 */
    do {
        if (cpu_in_sg(sg, eenv->src_cpu) || cpu_in_sg(sg, eenv->dst_cpu)) {
            eenv_before.sg_top = eenv->sg_top = sg;

            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;

            if (sched_group_energy(eenv))
                return 0; /* Invalid result abort */
            energy_after += eenv->energy;
        }
    } while (sg = sg->next, sg != sd->groups);
    /* 对能效结构体成员变量赋值 */
    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.
         */
        /* 这个补偿的目的是防止进程的频繁抖动，即可能进程P迁移到dst_cpu上，它的能效消耗马上
        比src_cpu高，很有可能造成更多此的进程迁移操作，得不偿失！！！ */
        margin = eenv->nrg.before >> 6; /* ~1.56% */

        diff = eenv->nrg.after - eenv->nrg.before;
        /* 两个cpu的能效差值太小，不值得操作。代价太高 */
        eenv->nrg.diff = (abs(diff) < margin) ? 0 : eenv->nrg.diff;

        return eenv->nrg.diff;
    }
```
上面在每次循环的时候都调用两次sched_group_energy的目的是计算前后的能效。
第一次调用sched_group_energy(eenv_before)

```
    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,
    };

```
第二次调用sched_group_energy(eenv)，eenv赋值在select_energy_cpu_brute函数中：

```
    if (target_cpu != prev_cpu) {
        int delta = 0;
        struct energy_env eenv = {
            .util_delta     = task_util(p),
            .src_cpu        = prev_cpu,
            .dst_cpu        = target_cpu,
            .task           = p,
            .trg_cpu    = target_cpu,
        };
```
其实eenv_before结构体是通过eenv赋值得到的。唯一的不同就是trg_cpu这个成员变量，其他都一致。即比较如果进程P在src_cpu上运行还是在dst_cpu上运行能效哪个好？

3.2.2 核心函数normalize_energy分析

上面已经分析了__enev_diff函数，返回的是before和after的能效差值：eenv->nrg.diff。

```
    static inline int
    energy_diff(struct energy_env *eenv)
    {
        int boost = schedtune_task_boost(eenv->task);
        int nrg_delta;

        /* Conpute "absolute" energy diff */
        __energy_diff(eenv);  /* 计算绝对能效差值 */

        /* Return energy diff when boost margin is 0 */
        /* 对于非boost的进程，直接返回能效差值 1|| boost == 0永远为true啊。。。*/
        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 计算归一化的能效差值,下面代码不会执行*/
    nrg_delta = normalize_energy(eenv->nrg.diff);
    eenv->nrg.delta = nrg_delta;
……………………….
}
```

下面分析核心函数normalize_energy是怎么计算nrg_delta：

```
/*
 * System energy normalization
 * Returns the normalized value, in the range [0..SCHED_CAPACITY_SCALE],
 * corresponding to the specified energy variation.
 */
static inline int
normalize_energy(int energy_diff)
{
    u32 normalized_nrg;

#ifdef CONFIG_CGROUP_SCHEDTUNE
    /* during early setup, we don't know the extents */
    if (unlikely(!schedtune_initialized))
        return energy_diff < 0 ? -1 : 1 ;
#endif /* CONFIG_CGROUP_SCHEDTUNE */

#ifdef CONFIG_SCHED_DEBUG
    {
    int max_delta;

    /* Check for boundaries */
    max_delta  = schedtune_target_nrg.max_power;
    max_delta -= schedtune_target_nrg.min_power;
    WARN_ON(abs(energy_diff) >= max_delta);
    }
#endif
    /* 将差值转化为正数 */
    /* Do scaling using positive numbers to increase the range */
    normalized_nrg = (energy_diff < 0) ? -energy_diff : energy_diff;

    /* Scale by energy magnitude */
    normalized_nrg <<= SCHED_CAPACITY_SHIFT;

    /* Normalize on max energy for target platform */
    /* 实际的计算方式是：normalized_nrg=
     abs(energy_diff) / (schedtune_target_nrg.max_power -
schedtune_target_nrg.min_power) */
    normalized_nrg = reciprocal_divide(
            normalized_nrg, schedtune_target_nrg.rdiv);

    return (energy_diff < 0) ? -normalized_nrg : normalized_nrg;
```

```
}
```

通过上面的函数，我们知道全局结构体变量schedtune_target_nrg在下面函数中赋值了。

```
/*
 * Initialize the constants required to compute normalized energy.
 * The values of these constants depends on the EM data for the specific
 * target system and topology.
 * Thus, this function is expected to be called by the code
 * that bind the EM to the topology information.
 */
static int
schedtune_init(void)
{   /* 下面为全局结构体变量schedtune_target_nrg赋值 */
    struct target_nrg *ste = &schedtune_target_nrg;
    unsigned long delta_pwr = 0;
    struct sched_domain *sd;
    struct sched_group *sg;

    pr_info("schedtune: init normalization constants...\n");
    ste->max_power = 0;
    ste->min_power = 0;

    rcu_read_lock();

    /*
     * When EAS is in use, we always have a pointer to the highest SD
     * which provides EM data.
     */
    sd = rcu_dereference(per_cpu(sd_ea, cpumask_first(cpu_online_mask)));
    if (!sd) {
        pr_info("schedtune: no energy model data\n");
        goto nodata;
    }
    /* 核心函数，schedtune_add_cluster_nrg，会获取每个cpu和cluster的最小和最大的
    power数值，并存储在ste结构体中，即全局结构体schedtune_target_nrg中 */
    sg = sd->groups;
    do {
        schedtune_add_cluster_nrg(sd, sg, ste);
    } while (sg = sg->next, sg != sd->groups);

    rcu_read_unlock();

    pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
        "SYSTEM", ste->min_power, ste->max_power);

    /* Compute normalization constants */
    delta_pwr = ste->max_power - ste->min_power;
    /* 将最大power和最小power的差值经过除法优化，转化为struct reciprocal_value rdiv
     结构体成员变量 */
    ste->rdiv = reciprocal_value(delta_pwr);
    pr_info("schedtune: using normalization constants mul: %u sh1: %u sh2:
%u\n",
        ste->rdiv.m, ste->rdiv.sh1, ste->rdiv.sh2);

    schedtune_test_nrg(delta_pwr);
```

```
#ifdef CONFIG_CGROUP_SCHEDTUNE
    schedtune_init_cgroups();
#else
    pr_info("schedtune: configured to support global boosting only\n");
#endif
    /* 将除数100,即经过除法优化, 转化为struct reciprocal_value结构体变量 */
    schedtune_spc_rdiv = reciprocal_value(100);

    return 0;

nodata:
    pr_warning("schedtune: disabled!\n");
    rcu_read_unlock();
    return -EINVAL;
}
postcore_initcall(schedtune_init);
```

上面的核心函数是schedtune_add_cluster_nrg(sd, sg, ste), 下面分析这个函数：

```
/*
 * Compute the min/max power consumption of a cluster and all its CPUs
 */
static void
schedtune_add_cluster_nrg(
        struct sched_domain *sd,
        struct sched_group *sg,
        struct target_nrg *ste)
{
    struct sched_domain *sd2;
    struct sched_group *sg2;

    struct cpumask *cluster_cpus;
    char str[32];

    unsigned long min_pwr;
    unsigned long max_pwr;
    int cpu;

    /* Get Cluster energy using EM data for the first CPU */
    /* 获取调度组的cpumask */
    cluster_cpus = sched_group_cpus(sg);
    snprintf(str, 32, "CLUSTER[%*pbl]",
         cpumask_pr_args(cluster_cpus));

    min_pwr = sg->sge->idle_states[sg->sge->nr_idle_states - 1].power;
    max_pwr = sg->sge->cap_states[sg->sge->nr_cap_states - 1].power;
    pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
        str, min_pwr, max_pwr);

    /*
     * Keep track of this cluster's energy in the computation of the
     * overall system energy
     */
    ste->min_power += min_pwr;
    ste->max_power += max_pwr;
```

```
	/* Get CPU energy using EM data for each CPU in the group */
	for_each_cpu(cpu, cluster_cpus) {
		/* Get a SD view for the specific CPU */
		for_each_domain(cpu, sd2) {
			/* Get the CPU group */
			sg2 = sd2->groups;
			min_pwr = sg2->sge->idle_states[sg2->sge->nr_idle_states - 1].power;
			max_pwr = sg2->sge->cap_states[sg2->sge->nr_cap_states - 1].power;

			ste->min_power += min_pwr;
			ste->max_power += max_pwr;

			snprintf(str, 32, "CPU[%d]", cpu);
			pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
				str, min_pwr, max_pwr);

			/*
			 * Assume we have EM data only at the CPU and
			 * the upper CLUSTER level
			 */
			BUG_ON(!cpumask_equal(
				sched_group_cpus(sg),
				sched_group_cpus(sd2->parent->groups)
				));
			break;
		}
	}
}
```
在核心函数里面使用到了energy的capacity，上面的函数使用到了energy的power数值。结构体struct sched_group_energy数值的填充在energy.c文件中：energy的获取 调度组与energy如何关联的

从函数schedtune_add_cluster_nrg中得到全局结构体schedtune_target_nrg的max_pwr和min_pwr这两个数值是常量数值，所以使用了除法优化方案struct `reciprocal_value`，具体的这个优化怎么计算的，可以看：Newton-Raphson除法优化

所以我们能够得到函数normalize_energy返回值计算方式如下：

normalized_nrg

=(abs(energy_diff) / (schedtune_target_nrg.max_pwr - schedtune_target_nrg.min_pwr))

=reciprocal_divide(abs(energy_diff) , schedtune_target_nrg.rdiv)

schedtune_target_nrg.max_pwr - schedtune_target_nrg.min_pwr是一个常量，在schedtune_init函数里面获取的。根据dts的设定的数值可以推算出来：

```
energy-costs {
	CPU_COST_0: core-cost0 {
		busy-cost-data = <
			501 80  /* 768MHz */
			576 101 /* 884MHz */
			652 125 /* 1000MHz */
			717 151 /* 1100MHz */
			782 181 /* 1200MHz */
		>;
```

```
            idle-cost-data = <
                25      /* ACTIVE-IDLE */
                25      /* WFI */
                0       /* CORE_PD */
            >;
        };
        CPU_COST_1: core-cost1 {
            busy-cost-data = <
                501 110 /* 768MHz */
                685 160 /* 1050MHz */
                799 206 /* 1225MHz */
                913 258 /* 1400MHz */
                978 305 /* 1500MHz */
                1024    352 /* 1570MHz */
            >;
            idle-cost-data = <
                44      /* ACTIVE-IDLE */
                44      /* WFI */
                0       /* CORE_PD */
            >;
        };
        CLUSTER_COST_0: cluster-cost0 {
            busy-cost-data = <
                501 0   /* 768MHz */
                576 0   /* 884MHz */
                652 0   /* 1000MHz */
                717 0   /* 1100MHz */
                782 0   /* 1200MHz */
            >;
            idle-cost-data = <
                0       /* ACTIVE-IDLE */
                0       /* WFI */
                0       /* CORE_PD */
            >;
        };
        CLUSTER_COST_1: cluster-cost1 {
            busy-cost-data = <
                501 68  /* 768MHz */
                685 85  /* 1050MHz */
                799 106 /* 1225MHz */
                913 130 /* 1400MHz */
                978 153 /* 1500MHz */
                1024 179 /* 1570MHz */
            >;
            idle-cost-data = <
                42      /* ACTIVE-IDLE */
                42      /* WFI */
                42      /* CORE_PD */
            >;
        };
    };
```

我们知道min_pwr是取的idle最大索引的power数值，从上面的dts看，min_pwr各个cpu的累加为42；而max_pwr去的是最大active索引的power数值，累加和=181*4 + 352 * 4 + 179 = 2311。所以差值为：2269。

3.2.3 核心函数schedtune_accept_deltas分析

具体实现如下：

```
int
schedtune_accept_deltas(int nrg_delta, int cap_delta,
            struct task_struct *task)
{
    struct schedtune *ct;
    int perf_boost_idx;
    int perf_constrain_idx;
    /* 条件判断 */
    /* Optimal (O) region */
    if (nrg_delta < 0 && cap_delta > 0) {
        trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, 1, 0);
        return INT_MAX;
    }

    /* Suboptimal (S) region */
    if (nrg_delta > 0 && cap_delta < 0) {
        trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, -1, 5);
        return -INT_MAX;
    }

    /* Get task specific perf Boost/Constraints indexes */
    rcu_read_lock();
    /* 通过进程获取schedtune数据结构体 */
    ct = task_schedtune(task);
    perf_boost_idx = ct->perf_boost_idx;
    perf_constrain_idx = ct->perf_constrain_idx;
    rcu_read_unlock();

    return __schedtune_accept_deltas(nrg_delta, cap_delta,
            perf_boost_idx, perf_constrain_idx);
}

/**
 * Performance-Energy (P-E) Space thresholds constants
 */
struct threshold_params {
    int nrg_gain;
    int cap_gain;
};

/*
 * System specific P-E space thresholds constants
 */
static struct threshold_params
threshold_gains[] = {
    { 0, 5 }, /*   < 10% */
    { 1, 5 }, /*   < 20% */
    { 2, 5 }, /*   < 30% */
    { 3, 5 }, /*   < 40% */
    { 4, 5 }, /*   < 50% */
```

```
    { 5, 4 }, /*   < 60% */
    { 5, 3 }, /*   < 70% */
    { 5, 2 }, /*   < 80% */
    { 5, 1 }, /*   < 90% */
    { 5, 0 }  /* <= 100% */
};

static int
__schedtune_accept_deltas(int nrg_delta, int cap_delta,
              int perf_boost_idx, int perf_constrain_idx)
{
    int payoff = -INT_MAX;
    int gain_idx = -1;

    /* Performance Boost (B) region */
    if (nrg_delta >= 0 && cap_delta > 0)
        gain_idx = perf_boost_idx;
    /* Performance Constraint (C) region */
    else if (nrg_delta < 0 && cap_delta <= 0)
        gain_idx = perf_constrain_idx;

    /* Default: reject schedule candidate */
    if (gain_idx == -1)
        return payoff;

    /*
     * Evaluate "Performance Boost" vs "Energy Increase"
     *
     * - Performance Boost (B) region
     *
     *   Condition: nrg_delta > 0 && cap_delta > 0
     *   Payoff criteria:
     *     cap_gain / nrg_gain  < cap_delta / nrg_delta =
     *     cap_gain * nrg_delta < cap_delta * nrg_gain
     *   Note that since both nrg_gain and nrg_delta are positive, the
     *   inequality does not change. Thus:
     *
     *     payoff = (cap_delta * nrg_gain) - (cap_gain * nrg_delta)
     *
     * - Performance Constraint (C) region
     *
     *   Condition: nrg_delta < 0 && cap_delta < 0
     *   payoff criteria:
     *     cap_gain / nrg_gain  > cap_delta / nrg_delta =
     *     cap_gain * nrg_delta < cap_delta * nrg_gain
     *   Note that since nrg_gain > x while nrg_delta < 0, the
     *   inequality change. Thus:
     *
     *     payoff = (cap_delta * nrg_gain) - (cap_gain * nrg_delta)
     *
     * This means that, in case of same positive defined {cap,nrg}_gain
     * for both the B and C regions, we can use the same payoff formula
     * where a positive value represents the accept condition.
     *///*根据上面描述，计算payoff的数值*/
```

```
        payoff  = cap_delta * threshold_gains[gain_idx].nrg_gain;
        payoff -= nrg_delta * threshold_gains[gain_idx].cap_gain;

        return payoff;
    }
```

OK，至此，函数energy_diff返回值要么为eenv->nrg.diff的数值，要么为-eenv->payoff数值，实际情况是，只有一种数值会返回：eenv->nrg.diff

3.3 根据能效结果选择目标cpu

```
static int select_energy_cpu_brute(struct task_struct *p, int prev_cpu, int sync)
{
.............
         /* energy_diff返回值>=0，说明进程P进入到dst_cpu上去，没有能效的节省 */
        if (energy_diff(&eenv) >= 0) {
            /* No energy saving for target_cpu, try backup，选择best_idle_cpu
            作为目标cpu */
            target_cpu = tmp_backup;
            /* 更新eenv环境变量 */
            eenv.dst_cpu = target_cpu;
            eenv.trg_cpu = target_cpu;
            /* 满足下面三个条件之一，就将目标cpu设置为唤醒进程P的cpu(prev_cpu),也是
              在没有能效节省的情况下，迁移进程P到其他cpu上，这中间的开销不得而知 */
            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;
}
```

至此EAS根据能效为候选进程P挑选可运行cpu的解析完毕。通读一下上面的分析，还是比较简单的。