几个结构体：

1、cfs_rq

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
………………………………..
#ifdef CONFIG_SCHED_WALT
    u64 cumulative_runnable_avg;
#endif

#ifdef CONFIG_CFS_BANDWIDTH
    int runtime_enabled;
    u64 runtime_expires;
    s64 runtime_remaining;

    u64 throttled_clock, throttled_clock_task;
    u64 throttled_clock_task_time;
    int throttled, throttle_count, throttle_uptodate;
    struct list_head throttled_list;
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

2、rq

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {  /*必须尽快明白这几个参数的含义*/
  ……………………………………………….
#ifdef CONFIG_SCHED_WALT
    u64 cumulative_runnable_avg;
    u64 window_start;
    u64 curr_runnable_sum;
    u64 prev_runnable_sum;
    u64 nt_curr_runnable_sum;
    u64 nt_prev_runnable_sum;
    u64 cur_irqload;
    u64 avg_irqload;
    u64 irqload_ts;
    u64 cum_window_demand;
#endif /* CONFIG_SCHED_WALT */
……………………………………………….
};
```

3、ravg

```
/* ravg represents frequency scaled cpu-demand of tasks */
struct ravg {
    /*
     * 'mark_start' marks the beginning of an event (task waking up, task
     * starting to execute, task being preempted) within a window
     *
```

```
        * 'sum' represents how runnable a task has been within current
        * window. It incorporates both running time and wait time and is
        * frequency scaled.
        *
        * 'sum_history' keeps track of history of 'sum' seen over previous
        * RAVG_HIST_SIZE windows. Windows where task was entirely sleeping are
        * ignored.
        *
        * 'demand' represents maximum sum seen over previous
        * sysctl_sched_ravg_hist_size windows. 'demand' could drive frequency
        * demand for tasks.
        *
        * 'curr_window' represents task's contribution to cpu busy time
        * statistics (rq->curr_runnable_sum) in current window
        *
        * 'prev_window' represents task's contribution to cpu busy time
        * statistics (rq->prev_runnable_sum) in previous window
        */
    u64 mark_start;
    /*sum在update_cpu_busy_time和update_task_demand里面更新
    * 而sum_history[]在update_task_demand调用update_history里面更新
    * sum_history[0]永远是最新的数值，是runtime时间，，数值是通过scale_exec_time转化
    * 的
    */
    u32 sum, demand;
    u32 sum_history[RAVG_HIST_SIZE_MAX];
    /*下面三个参数在update_cpu_busy_time函数里面更新的*/
    u32 curr_window, prev_window;
    u16 active_windows;
};
#endif
```

4、struct task_struct里面ravg结构体

```
struct task_struct {
......................
#ifdef CONFIG_SCHED_WALT
    struct ravg ravg;
    /*
     * 'init_load_pct' represents the initial task load assigned to children
     * of this task
     */
    u32 init_load_pct;
    u64 last_sleep_ts;
#endif
......................
}
```

弄明白上面的结构体元素怎么计算，怎么来的，就很好理解WALT算法的精髓了！！！！！

5、update_history怎么计算的：walt_ravg_hist_size=8

```
/*
 * Called when new window is starting for a task, to record cpu usage over
 * recently concluded window(s). Normally 'samples' should be 1. It can be >
1
```

```
 * when, say, a real-time task runs without preemption for several windows at a
 * stretch.
 */
static void update_history(struct rq *rq, struct task_struct *p,
             u32 runtime, int samples, int event)
{    /*指向sum_history数组指针的首地址*/
    u32 *hist = &p->ravg.sum_history[0];
    int ridx, widx;
    u32 max = 0, avg, demand;
    u64 sum = 0;

    /* Ignore windows where task had no activity */
    if (!runtime || is_idle_task(p) || exiting_task(p) || !samples)
            goto done;

    /* Push new 'runtime' value onto stack */
    widx = walt_ravg_hist_size - 1;
    ridx = widx - samples;
    /*sample=1的时候，将数据平移，最后空出hist[0]待填充，并计算hist数组的最大数值，相当于stack*/
    for (; ridx >= 0; --widx, --ridx) {
        hist[widx] = hist[ridx];
        sum += hist[widx];
        if (hist[widx] > max)
            max = hist[widx];
    }
    /*使用当前的runtime填充hist[0]*/
    for (widx = 0; widx < samples && widx < walt_ravg_hist_size; widx++) {
        hist[widx] = runtime;
        sum += hist[widx];
        if (hist[widx] > max)
            max = hist[widx];
    }

    p->ravg.sum = 0;
    /*根据policy的不同决策demand的数值*/
    if (walt_window_stats_policy == WINDOW_STATS_RECENT) {
        demand = runtime;
    } else if (walt_window_stats_policy == WINDOW_STATS_MAX) {
        demand = max;
    } else {
        avg = div64_u64(sum, walt_ravg_hist_size);
        if (walt_window_stats_policy == WINDOW_STATS_AVG)
            demand = avg;
        else
            demand = max(avg, runtime);
    }

    /*
     * A throttled deadline sched class task gets dequeued without
     * changing p->on_rq. Since the dequeue decrements hmp stats
     * avoid decrementing it here again.
     *
```

```
         * When window is rolled over, the cumulative window demand
         * is reset to the cumulative runnable average (contribution from
         * the tasks on the runqueue). If the current task is dequeued
         * already, it's demand is not included in the cumulative runnable
         * average. So add the task demand separately to cumulative window
         * demand.
         */
        /* 上面这段话的目的是校正cpu负载, 分两种情况。第一种, task在rq queue里面, 上次task
           的demand为x, 本次计算为y,     * 则cpu负载:cumulative_runnable_avg += y-x。
    第
           二种情况task不在rq queue里面, 并且当前task是本次计算demand的task,则直接计算
          window load, cum_window_demand += y;感觉还是没有讲清楚, 在仔细check下
        */
        if (!task_has_dl_policy(p) || !p->dl.dl_throttled) {
            if (task_on_rq_queued(p))
                fixup_cumulative_runnable_avg(rq, p, demand);
            else if (rq->curr == p)
                fixup_cum_window_demand(rq, demand);
        }

        p->ravg.demand = demand;

    done:
        trace_walt_update_history(rq, p, runtime, samples, event);
        return;
    }
```

6、搞清楚, rq里面和ravg里面几个元素的含义是什么, 并加以理解和怎么获取, 怎么更新的

6.1 rq

```
    #ifdef CONFIG_SCHED_WALT
        u64 cumulative_runnable_avg;
        u64 window_start;
        u64 curr_runnable_sum;
        u64 prev_runnable_sum;
        u64 nt_curr_runnable_sum;
        u64 nt_prev_runnable_sum;
        u64 cur_irqload;
        u64 avg_irqload;
        u64 irqload_ts;
        u64 cum_window_demand;
    #endif /* CONFIG_SCHED_WALT */
```

6.1.1 cumulative_runnable_avg

这个数值的计算依赖task元素ravg里面的demand, 即task->ravg.demand, 而且结构体简介了demand的意义 : /*

```
         * 'demand' represents maximum sum seen over previous
         * sysctl_sched_ravg_hist_size windows. 'demand' could drive frequency
         * demand for tasks.
    */
```

大致意思是 :

demand是基于之前窗口获取的max sum，对于task，demand能够驱动频率需求。这个demand应该是一个非常重要，对于频率的改变，从下面两个调用可以理解：

```
/*rt.c ， 两个sched_class*/
static inline unsigned long task_walt_util(struct task_struct *p)
{
#ifdef CONFIG_SCHED_WALT
    if (!walt_disabled && sysctl_sched_use_walt_task_util) {
        unsigned long demand = p->ravg.demand;

        return (demand << 10) / walt_ravg_window;
    }
#endif
    return 0;
}
-------------------------
/*fair.c*/
static inline unsigned long task_util(struct task_struct *p)
{
#ifdef CONFIG_SCHED_WALT  /*使用WALT*/
    if (!walt_disabled && sysctl_sched_use_walt_task_util) {
        unsigned long demand = p->ravg.demand;
        return (demand << 10) / walt_ravg_window;
    }
#endif
    /*使用PELT*/
    return p->se.avg.util_avg;
}
```

在walt.c初始化ravg.demand数值：

```
static unsigned int task_load(struct task_struct *p)
{
    return p->ravg.demand;
}
void walt_init_new_task_load(struct task_struct *p)
{
    int i;
    /*init_load_windows = 1800 000,应该是1800us*/
    u32 init_load_windows =
            div64_u64((u64)sysctl_sched_walt_init_task_load_pct *
                          (u64)walt_ravg_window, 100);
    u32 init_load_pct = current->init_load_pct;

    p->init_load_pct = 0;
    memset(&p->ravg, 0, sizeof(struct ravg));

    if (init_load_pct) {
        init_load_windows = div64_u64((u64)init_load_pct *
               (u64)walt_ravg_window, 100);
    }
    /*最终code验证了下，init的ravg.demand数值一直都是1800us*/
    p->ravg.demand = init_load_windows;
    for (i = 0; i < RAVG_HIST_SIZE_MAX; ++i)
        p->ravg.sum_history[i] = init_load_windows;
}
```

对于这个函数的解释如下

- current->init_load_psc这个元素，是task_struct里面的元素，意思表示，这个task分配给children的初始化task load，但是没有看到在哪里赋值，所以为0，而且本地验证这个数值一直也是为0的；
- 所以可以得到arvg.demand数值为init_load_windows了，
- 同时初始化ravg.sum_history[]数组元素。
- walt_init_task_load在wakeup一个新创建的task和创建一个新task是被调用，如下：

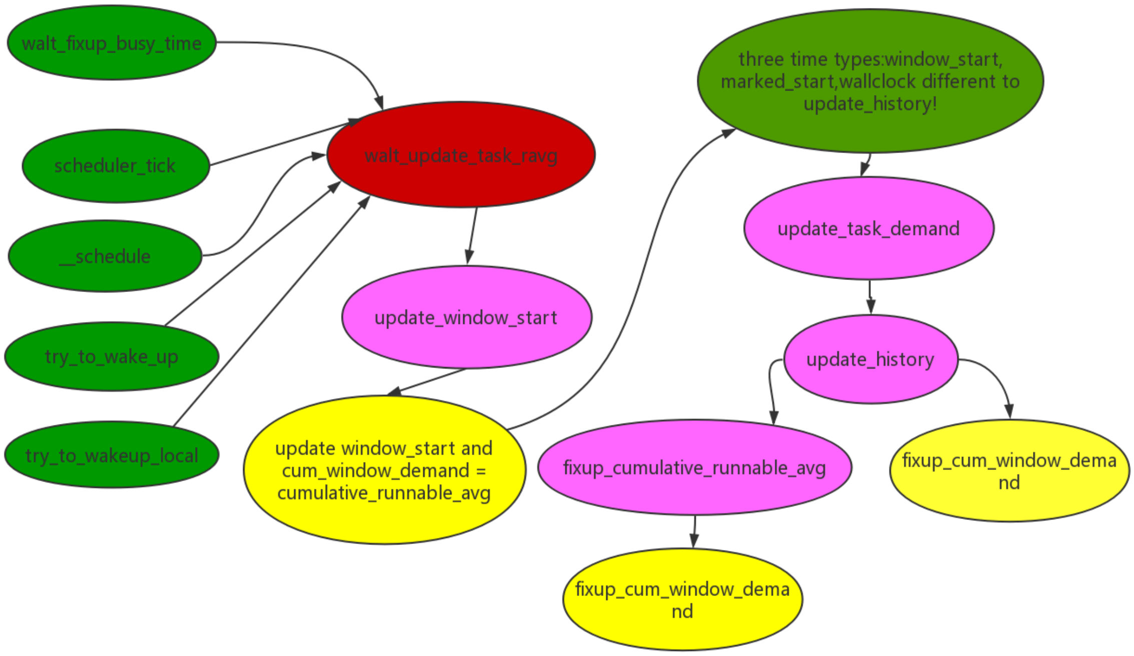
```
/*
 * Perform scheduler related setup for a newly forked process p.
 * p is forked by current.
 *
 * __sched_fork() is basic setup used by init_idle() too:
 */
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
..........
     walt_init_new_task_load(p);
............
}

/*
 * wake_up_new_task - wake up a newly created task for the first time.
 *
 * This function will do some initial scheduler statistics housekeeping
 * that must be done for every newly created context, then puts the task
 * on the runqueue and wakes it.
 */
void wake_up_new_task(struct task_struct *p)
{
..........
      walt_init_new_task_load(p);
..........
}
```

上面得到了初始化的ravg.demad数值，真实的获取在如下函数中：

```
/*
 * Called when new window is starting for a task, to record cpu usage over
 * recently concluded window(s). Normally 'samples' should be 1. It can be >
1
 * when, say, a real-time task runs without preemption for several windows
at a
 * stretch.
 */
static void update_history(struct rq *rq, struct task_struct *p,
             u32 runtime, int samples, int event)
{
    u32 *hist = &p->ravg.sum_history[0];
    int ridx, widx;
    u32 max = 0, avg, demand;
    u64 sum = 0;

    /* Ignore windows where task had no activity */
    if (!runtime || is_idle_task(p) || exiting_task(p) || !samples)
            goto done;
```

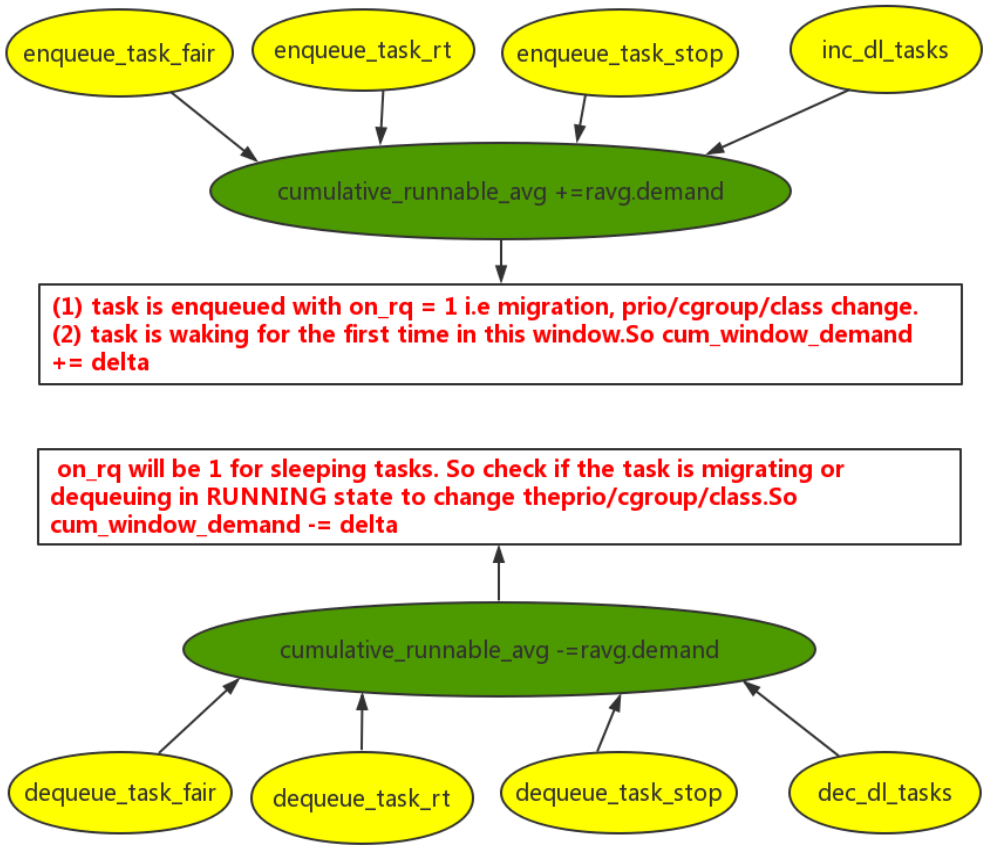
```
        /* Push new 'runtime' value onto stack */
        widx = walt_ravg_hist_size - 1;
        ridx = widx - samples;
        for (; ridx >= 0; --widx, --ridx) {
            hist[widx] = hist[ridx];
            sum += hist[widx];
            if (hist[widx] > max)
                max = hist[widx];
        }

        for (widx = 0; widx < samples && widx < walt_ravg_hist_size; widx++) {
            hist[widx] = runtime;
            sum += hist[widx];
            if (hist[widx] > max)
                max = hist[widx];
        }

        p->ravg.sum = 0;

        if (walt_window_stats_policy == WINDOW_STATS_RECENT) {
            demand = runtime;
        } else if (walt_window_stats_policy == WINDOW_STATS_MAX) {
            demand = max;
        } else {
            avg = div64_u64(sum, walt_ravg_hist_size);
            if (walt_window_stats_policy == WINDOW_STATS_AVG)
                demand = avg;
            else
                demand = max(avg, runtime);
        }
        /*上面的代码分如下几部分：
         1.首先获取sum_history数值指针放在hist指针中，并将前widx-samples个元素往后移动，
          比如widx-samples为3，widx为最大数组索引，则将前三个元素往后移动，移动从最大索引
          开始覆盖，并计算这个三个元素之后和最大值
         2. 将samples个元素插入到第一步没有覆盖的数组中，再次累加，和计算最大数值
         3. 根据policy，如何得到一个task的WALT，是最近的值 or 是最大值，还是平均值亦或是
          最大值与最近值的最大者
        */
        /*
         * A throttled deadline sched class task gets dequeued without
         * changing p->on_rq. Since the dequeue decrements hmp stats
         * avoid decrementing it here again.
         *
         * When window is rolled over, the cumulative window demand
         * is reset to the cumulative runnable average (contribution from
         * the tasks on the runqueue). If the current task is dequeued
         * already, it's demand is not included in the cumulative runnable
         * average. So add the task demand separately to cumulative window
         * demand.
         */
        /* 上面这段话的目的是校正参数，分两种情况。第一种，task在rq queue里面，上次task
           的demand为x，本次计算为y，则cpu负载:cumulative_runnable_avg += (y-x)。
           第二种情况task不在rq queue里面，并且当前task是本次计算demand的task,则直接计算
```

```
        window load, cum_window_demand += y;感觉还是没有讲清楚，在仔细check下
    */
    if (!task_has_dl_policy(p) || !p->dl.dl_throttled) {
        if (task_on_rq_queued(p))
            fixup_cumulative_runnable_avg(rq, p, demand);
        else if (rq->curr == p)
            fixup_cum_window_demand(rq, demand);
    }

    p->ravg.demand = demand;  //更新ravg.demand

done:
    trace_walt_update_history(rq, p, runtime, samples, event);
    return;
}
```
ravg.demand update，cumulative_runnable_avg和cum_window_demand补偿如下：



有下面三个途径会修rq的cumulative_runnable_avg:
- walt_inc_cumulative_runnable_avg，task enqueue会修改
- walt_dec_cumulative_runnable_avg，task dequeue会修改
- fixup_cumulative_runnable_avg，update_history会修改

修改cummulative_runnable_avg的拓扑图如下：



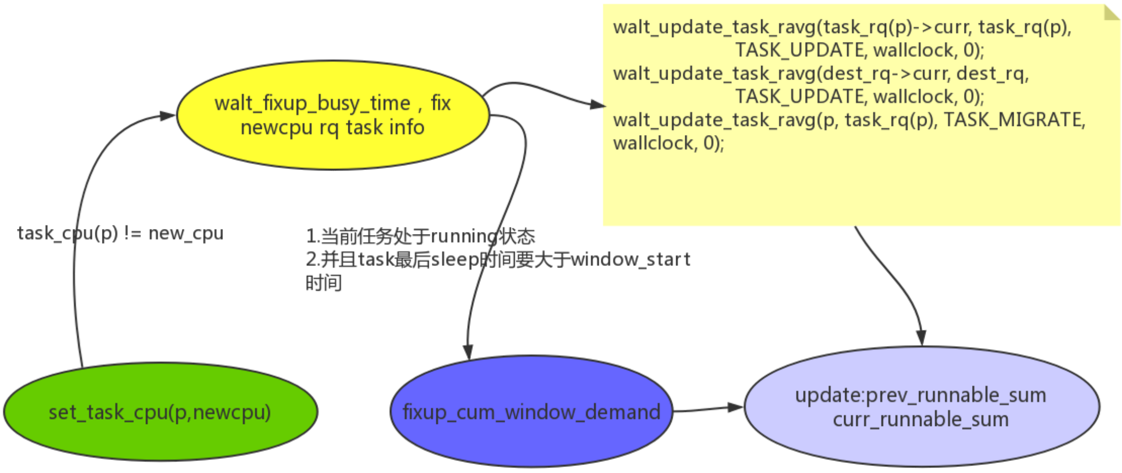
从上面可以知道cumulative_runnable_avg的数值来源是按照这样来的：

- 获取初始值ravg.demand
- 在tick scheduler和task wakeup schedule过程中，update，walt_update_task_ravg，之后会调用到update_history函数，来更新ravg.demand，唯一的更新路径。
- 同时在各个sched_class enqueue和dequque过程中调用函数增减函数来更新cumulative_runnable_avg数值，其实就是叠加或者减少ravg.demand数值，也说明了demand这个元素的意义了，它的大小直接影响utilization。

6.1.2 cum_window_demand/prev_runnable_sum/curr_runnable_sum

从6.1.1节可以看到cum_window_demand数值与cumulative_runnable_avg有很大关系，下面来讲解这个，下面是更新cum_window_demand的调用路径：

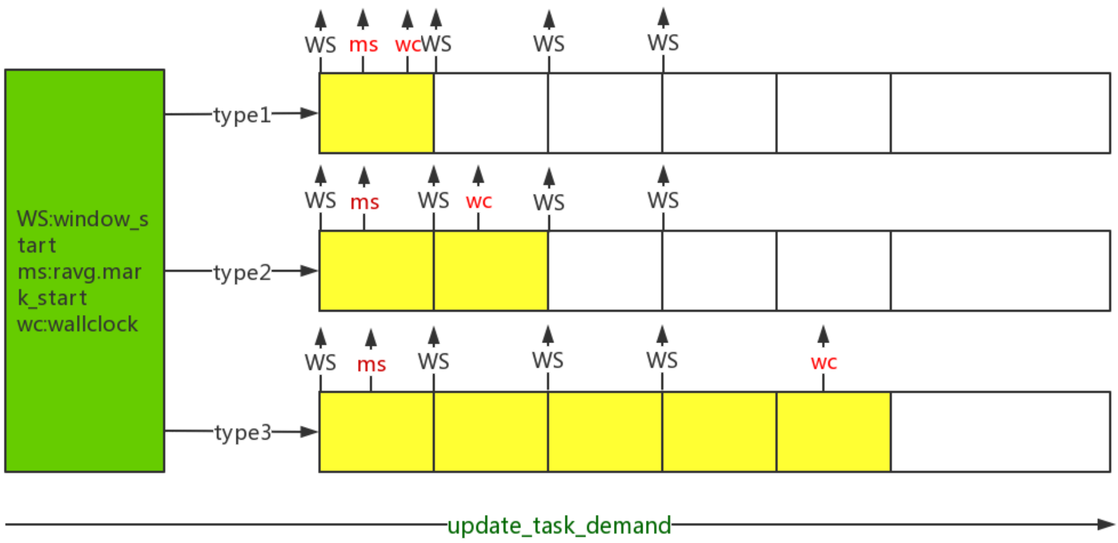
- 每次更新(task enqueue/dequeue)cumulative_runnable_avg就必然会update cum_window_demand（根据当前进程的状态来觉得是否要update，可以从6.1.1节方框图可以看到）
- task状态变化,如迁移到另一个cpu上



上面是cum_window_demand的更新，下面讲解prev_runnable_sum和curr_runnable_sum:
需要先看下ravg.curr_window和prev_window怎么计算的。
这里面涉及到的内容还是蛮多的，我们知道curr_runnable_sum是当前task在此rq里面runnable的时间累加，在ravg.curr_window是当前，涉及到的计算如下：
1. 如果task的状态发生变化，上面的四个参数都会跟随变化，正常路径，walt_update_task_ravg是从这里更新，操作如下：

```
if (p->ravg.curr_window) {
    src_rq->curr_runnable_sum -= p->ravg.curr_window;
    dest_rq->curr_runnable_sum += p->ravg.curr_window;
}

if (p->ravg.prev_window) {
    src_rq->prev_runnable_sum -= p->ravg.prev_window;
    dest_rq->prev_runnable_sum += p->ravg.prev_window;
}
```

2. 如果task发生任务迁移，即task从一个cpu迁移到newcpu，则需要fix cpu的busy time，也就是上面的逻辑图所示的。被调用set_task_cpu即将当前的task迁移到newcpu上。
3. 按照walt_update_task_ravg的路径来讲解上面四个参数是怎么计算的。
分三种类型来计算：



type1: task event在一个window内，只需要将wc-ms的执行时间累加到ravg.sum上即可
type2：task跨两个window：更新ravg.sum和ravg.sum_history[]最后一个数值。
type3：task跨超过两个window：更新ravg.sum和ravg.sum_history[]更新n个窗口的数值。
目的都是通过add_to_task_demand更新一个窗口内的ravg.sum数值。并且根据相应的条件调用update_history来更新ravg.sum_history数值。计算方式很巧妙。至此计算完毕了一个task的demand并存储在ravg.sum中，这个数值是一个window窗口task占用的数值。

通过update_cpu_busy_time计算rq curr_runnable_sum/prev_runnable_sum数值分的更加精细，也是通过上面的三种类型来计算prev_runnable_sum和curr_runnable_sum数值，同时更新ravg.active_window，curr_window，prev_window。计算一个task的runnable时间通过函数scal_exec_time，频率和capacity，时间归一化函数来计算runnable时间。例如：new_window=ravg.mark_start - window_start，如果它为0，则占用窗口的时间为delta=wallclock-mark_start时间，转化为runnable时间为curr_runnable_sum+=scale_exec_time(delta,rq),如果task不是idle/退出的thread，则ravg.curr_window+=delta。很有意思的计算方式。
设定new_window=ravg.mark_start-window_start，先初始化ravg.curr_window, prev_window,active_windows数值。update_cpu_busy_time执行流程如下：

- 如果task的状态是idle/exit等，如果new_window==0,则直接返回，否则将curr_runnable_sum清零，prev_runnable_sum根据窗口大小是否使用之前数值还是清零。
- new_window==0，即mark_start与window_start在一个窗口内，计算得到curr_runnable_sum+=scale_exec_time(wc-mark_start,rq);
- new_window==1,分如下三种情况来处理，窗口是否需要rollover
    1. 计算的task不是current thread，即标记p_is_curr_task为空，窗口不需要roll over。
    2. !irqtime || !is_idle_task(p) || cpu_is_waiting_on_io(rq)，即irqtime为0，或者p不是idle进程或者当前rq正在waiting io，cpu_is_waiting_on_io被设定为一直返回0。窗口需要roll over
    3. irqtime != 0，窗口也需要roll over，但是计算方式与2是不一样的。

主要的计算思想如下：

- 首先计算窗口数量，nr_window=(window_start-mark_start)/window_size(常量)。

- 如果nr_window==0，则计算之前窗口运行的时间delta=scale_exec_time(window-mark_start,rq)，如果task不是exiting进程，则ravg.prev_window+=delta
- 如果nr_window!=0，则delta=scale_exec_window(window_size,rq)，如果task不是exiting进程，则ravg.prev_window=delta。
- 如果需要roll over则，prev_runnable_sum+=delta；否则prev_runnable_sum=delta
- 计算当前task的runnable：curr_runnable_sum=scale_exec_time(wallclock-window_start,rq)
- 如果task不是idle进程，则ravg.curr_window=delta

从上面能够清晰的看到，curr_runnable_sum/ravg.curr_window数值，都是wallclock-window_start归一化时间，对于irqtime为1的情况可以看code稍有不同，但是原理都是类似的。流程图如下：

6.1.3 window_start

在walt.c文件设定函数是怎样的？

```
void walt_set_window_start(struct rq *rq)
{
    int cpu = cpu_of(rq);
    struct rq *sync_rq = cpu_rq(sync_cpu);

    if (likely(rq->window_start))
        return;
    /*感觉下面的code仅仅会被执行一次，check一下*/
    if (cpu == sync_cpu) {
        rq->window_start = 1;
    } else {
        raw_spin_unlock(&rq->lock);
        double_rq_lock(rq, sync_rq);
        rq->window_start = cpu_rq(sync_cpu)->window_start;
        rq->curr_runnable_sum = rq->prev_runnable_sum = 0;
        raw_spin_unlock(&sync_rq->lock);
    }

    rq->curr->ravg.mark_start = rq->window_start;
}
```

关键点是这个函数在哪里调用的，如下。有三个调用路径，一个是migration_call（接受CPU_UP_PREPARE notification event），另一个是scheduler_tick，周期性获取，最后一个update_window_start,会在walt_update_ravg_task中被调用

- 第一次调用wal_set_window_start，会走到cpu==sync_cpu，初始化window_start数值为1ms。之后会在函数update_window_start(rq,wallclock)里面更新真实的window_start时间：

```
static void
update_window_start(struct rq *rq, u64 wallclock)
{
    s64 delta;
    int nr_windows;

```

```
    delta = wallclock - rq->window_start;
    /* If the MPM global timer is cleared, set delta as 0 to avoid kernel
BUG happening */
    if (delta < 0) {
        delta = 0;
        WARN_ONCE(1, "WALT wallclock appears to have gone backwards or
reset\n");
    }

    if (delta < walt_ravg_window)
        return;

    nr_windows = div64_u64(delta, walt_ravg_window);
    rq->window_start += (u64)nr_windows * (u64)walt_ravg_window;

    rq->cum_window_demand = rq->cumulative_runnable_avg;
}
```

update_window_start函数被调用在如下函数中：

```
/* Reflect task activity on its demand and cpu's busy time statistics */
void walt_update_task_ravg(struct task_struct *p, struct rq *rq,
         int event, u64 wallclock, u64 irqtime)
{
    if (walt_disabled || !rq->window_start)
        return;

    lockdep_assert_held(&rq->lock);

    update_window_start(rq, wallclock);

    if (!p->ravg.mark_start)
        goto done;

    update_task_demand(p, rq, event, wallclock);
    update_cpu_busy_time(p, rq, event, wallclock, irqtime);

done:
    trace_walt_update_task_ravg(p, rq, event, wallclock, irqtime);

    p->ravg.mark_start = wallclock;
}
```

这个函数被调用的路径很多，因为涉及到task，wakeup，idle，fork，migration，irq等等task event的不同处理，在后面的章节中有相关的调用路径讲解。

从上面可以知道，window_start数值的获取，是通过wallclock减去之前设定的window_start数值，并看这个diff之间存在几个已经设定好的window规格(默认是20ms，五个窗口，我们手机上现在修改为12ms，8个窗口)。计算方式如下：

new_window_start += ((wallclock-window_start)/window_size) *window_size

其中wallclock是系统时间，从开机到现在的实际单位为ns，如果系统suspend了，则wallclock为上次suspend的时间。，代码实现如下：

```
u64 walt_ktime_clock(void)
{
    if (unlikely(walt_ktime_suspended))
        return ktime_to_ns(ktime_last);
```

```
    return ktime_get_ns();
}

static void walt_resume(void)
{
    walt_ktime_suspended = false;
}

static int walt_suspend(void)
{
    ktime_last = ktime_get();
    walt_ktime_suspended = true;
    return 0;
}

static struct syscore_ops walt_syscore_ops = {
    .resume = walt_resume,
    .suspend = walt_suspend
};

static int __init walt_init_ops(void)
{
    register_syscore_ops(&walt_syscore_ops);
    return 0;
}
late_initcall(walt_init_ops);
```

6.1.4 nt_curr_runnable_sum/nt_prev_runnable_sum

这两个元素没有被使用。

6.1.5 cur_irqload/avg_irqloa/irqload_ts

上面三个涉及到irq统计的，一起讲解

```
/*有如下两条调用路径，分为硬件中断，软件中断两个路径*/
__irq_enter(__do_softirq)--> account_irq_enter_time --> irqtime_account_irq
--> walt_account_irqtime

/*
 * Called before incrementing preempt_count on {soft,}irq_enter
 * and before decrementing preempt_count on {soft,}irq_exit.
 */
void irqtime_account_irq(struct task_struct *curr)
{
    ..........
#ifdef CONFIG_SCHED_WALT
    u64 wallclock;
    bool account = true;
#endif
.......
#ifdef CONFIG_SCHED_WALT
    wallclock = sched_clock_cpu(cpu);
#endif
    delta = sched_clock_cpu(cpu) - __this_cpu_read(irq_start_time);
    __this_cpu_add(irq_start_time, delta);
```

```
    irq_time_write_begin();
    /*
     * We do not account for softirq time from ksoftirqd here.
     * We want to continue accounting softirq time to ksoftirqd thread
     * in that case, so as not to confuse scheduler with a special task
     * that do not consume any time, but still wants to run.
     */
    /*我们没有考虑ksoftirqd这里的softirq时间。在这种情况下，我们希望继续将softirq时间计
     *算到ksoftirqd线程，以免将调度程序与不消耗任何时间但仍想运行的特殊任务混淆。
    */
    if (hardirq_count())
        __this_cpu_add(cpu_hardirq_time, delta);
    else if (in_serving_softirq() && curr != this_cpu_ksoftirqd())
        __this_cpu_add(cpu_softirq_time, delta);
#ifdef CONFIG_SCHED_WALT
    else
        account = false;  //注意这点，出现这种情况应该很少
#endif

    irq_time_write_end();
#ifdef CONFIG_SCHED_WALT
     if (account)
         walt_account_irqtime(cpu, curr, delta, wallclock);
#endif
    local_irq_restore(flags);
}
EXPORT_SYMBOL_GPL(irqtime_account_irq);
```

从上面信息能够看到：

- delta是irq运行时间，unit：ns
- wallclock是当前函数运行在cpu上的系统时间，unit：ns

好，我们现在可以来看上面三个元素如何计算的了：这三个元素初始化在kernel/sched/core.c文件，如下：

```
void __init sched_init(void)
{
.......................
#ifdef CONFIG_SCHED_WALT
        rq->cur_irqload = 0;
        rq->avg_irqload = 0;
        rq->irqload_ts = 0;
#endif
.......................
}
```

我们在看看walt_account_irqtime如何对上面三个参数update了：

```
void walt_account_irqtime(int cpu, struct task_struct *curr,
                 u64 delta, u64 wallclock)
{
    struct rq *rq = cpu_rq(cpu);
    unsigned long flags, nr_windows;
    u64 cur_jiffies_ts;

    raw_spin_lock_irqsave(&rq->lock, flags);
```

```
    /*
     * cputime (wallclock) uses sched_clock so use the same here for
     * consistency.
     */
    delta += sched_clock() - wallclock;
    cur_jiffies_ts = get_jiffies_64();

    if (is_idle_task(curr))
        walt_update_task_ravg(curr, rq, IRQ_UPDATE, walt_ktime_clock(),
                 delta);

    nr_windows = cur_jiffies_ts - rq->irqload_ts;

    if (nr_windows) {
        if (nr_windows < 10) {
            /* Decay CPU's irqload by 3/4 for each window. */
            rq->avg_irqload *= (3 * nr_windows);
            rq->avg_irqload = div64_u64(rq->avg_irqload,
                        4 * nr_windows);
        } else {
            rq->avg_irqload = 0;
        }
        rq->avg_irqload += rq->cur_irqload;
        rq->cur_irqload = 0;
    }

    rq->cur_irqload += delta;
    rq->irqload_ts = cur_jiffies_ts;
    raw_spin_unlock_irqrestore(&rq->lock, flags);
}
```

说明如下：

- delta原先数值是irq开始时间到执行函数irqtime_account_irq的时间差值，现在执行到walt_account_irqtime函数，由于中间经过了很多代码指令的执行，再次校正delta数值：delta += sched_clock -wallclock(上次系统时间)
- cur_jiffies_ts获取当前jiffies节拍数。
- nr_windows是计算本地统计irqtime，cur_jiffies_ts与上次统计irqtime，irqload_ts的差值
- 如果nr_windows数值在10节拍内，则每个window衰减 avg_irqload为原先的0.75，否则avg_irqload为0。这里面的含义是说，如果一个rq上的cpu irq中断时间间隔比较长，那么它的avg_irqload就可以忽略不计。同时将当前cur_irqload累加到avg_irqload，一般nr_window不为0。avg_irqload其实是一个累加值。
- 最后更新cur_irqload为累加时间，irqload_ts为当前统计irqtime的jiffies时间。

最后看下看下walt_cpu_high_irqload这个函数是做什么的：

```
#define WALT_HIGH_IRQ_TIMEOUT 3

u64 walt_irqload(int cpu) {
    struct rq *rq = cpu_rq(cpu);
    s64 delta;
    delta = get_jiffies_64() - rq->irqload_ts;

        /*
```

```
         * Current context can be preempted by irq and rq->irqload_ts can be
         * updated by irq context so that delta can be negative.
         * But this is okay and we can safely return as this means there
         * was recent irq occurrence.
         */
    /*当前上下文可以被irq抢占，并且rq-> irqload_ts可以通过irq上下文更新，以便delta可以是
      负数。但这没关系，我们可以安全返回，因为这意味着最近发生了irq。怎么会为负数呢？
    */
      if (delta < WALT_HIGH_IRQ_TIMEOUT)
          return rq->avg_irqload;
      else
          return 0;
  }

  int walt_cpu_high_irqload(int cpu) {
      return walt_irqload(cpu) >= sysctl_sched_walt_cpu_high_irqload;
  }
```

是判断当前cpu irqload是否过高的。其中：

```
  __read_mostly unsigned int sysctl_sched_walt_cpu_high_irqload =
      (10 * NSEC_PER_MSEC);
```

为10ms，如果当前cpu上avg_irqload超过10ms，则过高。WALT_HIGH_IRQ_TIMEOUT这个宏表示，如果当前时间减去计算irqtime的时间diff小于这个宏，则认为irqload还影响着系统，就需要返回avg_irqload，也就是累加的irqload数值，为何是3呢？不太明白？

find_best_target，如果walt_cpu_high_irqload返回值为1，怎此cpu不合适，继续遍历其他cpu函数，并且每次都会判决，这个函数是做负载均衡的，目的是在sched domain里面找到最佳的cpu，之后将task迁移过去。负载均衡是很大的内容，以后在细看。