



3.4 HMP 调度器

ARM 公司在推出 Cortex-A15 之后市场反馈功耗有点过大，于是提出了大小核的概念（注意不是大小老婆别看错了喔:-O）即 big.LITTLE 模型，该模型主要目的是为了省电。在 big.LITTLE 模型之前，处理器省电的主要技术是 DVFS（Dynamic Voltage and Frequency scaling）动态电压频率调整，根据应用程序计算能力的不同动态的调整运行频率和电压，从而达到省电的目的。目前大部分旗舰手机基本上都才有 big.LITTLE 模型，比较经典的配置是 Cortex-A72+Cortex-A53，Cortex-A72 是大核，Cortex-A53 是小核。用通俗的话来概况 big.LITTLE 模型的话就是用大核干重活，用小核来干轻活。big.LITTLE 模型在计算机术语上称为 HMP（Heterogeneous Multi-Processing）。目前的 Linux 内核实现的 CPU 负载均衡算法是基于 SMP 模型的，并没有考虑到 big.LITTLE 模型，因此 Linaro 组织对 big.LITTLE 模型开发了全新的负载均衡调度器，称为 HMP 调度器。

HMP 调度器并没有合并到 Linux 内核中，因此我们采用 Linaro 组织开发的 Linux 内核分支^①，它最新的代码是 Linux3.10^②，本文以该内核版本为蓝本。另外各大手机厂商根据 Linaro 开发的这个 HMP 调度器为蓝本在结合各自不同的需要做了不少的特别的优化^③。

1 初始化

HMP 的初始入口和 CFS 调度器一样。

[kernel/sched/fair.c]

```
__init void init_sched_fair_class(void)
{
#ifdef CONFIG_SMP
    open_softirq(SCHED_SOFTIRQ, run_rebalance_domains);
#endif CONFIG_SCHED_HMP
    hmp_cpu_mask_setup();
#endif
#endif /* SMP */
}
```

和内核中默认的负载均衡调度器一样需要注册一个软中断 softirq，回调函数是 run_rebalance_domains()。另外一个就是建立 HMP 的 CPU 拓扑关系。HMP 调度器重新定义了 domain 的实现，定义了 struct hmp_domain 数据结构，该结构比较简单，cpus 和 possible_cpus 两个 cpumask 变量以及一个链表节点。hmp_cpu_domain 是定义为 pre-CPU 变量，即每个 CPU 有一个 struct hmp_domain 数据结构，另外还定义了一个全局的链表 hmp_domains。

[include/linux/sched.h]

```
struct hmp_domain {
    struct cpumask cpus;
```

^① <https://git.linaro.org/arm/big.LITTLE/mp.git>

^② <https://releases.linaro.org/components/kernel/linux-linaro-stable/16.03/linux-linaro-stable-3.10.100-2016.03.tar.bz2>

^③ HTC 手机内核源代码下载:<http://www.htcdev.com/devcenter/downloads>；魅族手机内核源代码下载：<https://github.com/meizuosc>

```

    struct cpumask possible_cpus;
    struct list_head hmp_domains;
};

static LIST_HEAD(hmp_domains);
DECLARE_PER_CPU(struct hmp_domain *, hmp_cpu_domain);
#define hmp_cpu_domain(cpu)    (per_cpu(hmp_cpu_domain, (cpu)))

```

arch_get_hmp_domains()实现和体系结构相关，实现在 arch/arm/kernel/topology.c 文件中。

[init_sched_fair_class()->hmp_cpu_mask_setup()->arch_get_hmp_domains()]

```

0 struct cpumask hmp_slow_cpu_mask;
1
2 void __init arch_get_hmp_domains(struct list_head *hmp_domains_list)
3 {
4     struct cpumask hmp_fast_cpu_mask;
5     struct hmp_domain *domain;
6
7     arch_get_fast_and_slow_cpus(&hmp_fast_cpu_mask, &hmp_slow_cpu_mask);
8
9     /*
10    * Initialize hmp_domains
11    * Must be ordered with respect to compute capacity.
12    * Fastest domain at head of list.
13    */
14    if(!cpumask_empty(&hmp_slow_cpu_mask)) {
15        domain = (struct hmp_domain *)
16            kmalloc(sizeof(struct hmp_domain), GFP_KERNEL);
17        cpumask_copy(&domain->possible_cpus, &hmp_slow_cpu_mask);
18        cpumask_and(&domain->cpus, cpu_online_mask, &domain->possible_cpus);
19        list_add(&domain->hmp_domains, hmp_domains_list);
20    }
21    domain = (struct hmp_domain *)
22        kmalloc(sizeof(struct hmp_domain), GFP_KERNEL);
23    cpumask_copy(&domain->possible_cpus, &hmp_fast_cpu_mask);
24    cpumask_and(&domain->cpus, cpu_online_mask, &domain->possible_cpus);
25    list_add(&domain->hmp_domains, hmp_domains_list);
26}

```

首先 arch_get_fast_and_slow_cpus()去获取系统中大小核 CPU 的 index。这里分别为大小核定义了 domain，把小核的 CPUs 放到小核的 domain 上，大核 CPUs 放到大核 domain 上，然后加入到刚才提到的全局链表 hmp_domains_list。

```

0 static const char * const little_cores[] = {
1     "arm,cortex-a7",
2     NULL,
3 };
4
5 static bool is_little_cpu(struct device_node *cn)
6 {
7     const char * const *lc;
8     for (lc = little_cores; *lc; lc++)
9         if (of_device_is_compatible(cn, *lc))
10             return true;
11     return false;

```

```

12}
13
14void __init arch_get_fast_and_slow_cpus(struct cpumask *fast,
15                                         struct cpumask *slow)
16{
17     struct device_node *cn = NULL;
18     int cpu;
19
20     cpumask_clear(fast);
21     cpumask_clear(slow);
22     ...
23     while ((cn = of_find_node_by_type(cn, "cpu"))) {
24
25         const u32 *mpidr;
26         int len;
27
28         mpidr = of_get_property(cn, "reg", &len);
29         if (!mpidr || len != 4) {
30             pr_err("%s missing reg property\n", cn->full_name);
31             continue;
32         }
33
34         cpu = get_logical_index(be32_to_cpup(mpidr));
35         if (cpu == -EINVAL) {
36             pr_err("couldn't get logical index for mpidr %x\n",
37                   be32_to_cpup(mpidr));
38             break;
39         }
40
41         if (is_little_cpu(cn))
42             cpumask_set_cpu(cpu, slow);
43         else
44             cpumask_set_cpu(cpu, fast);
45     }
46     ...
47}

```

查询那些 CPU 是大小核，HMP 调度器实现了两种方式，一个是在 CONFIG 中定义，另外一个是通过 DTS。DTS 的方式比较通用，我们直接看 DTS 的方式。第 28~34 行从 DTS 中读取 CPU，然后判断该 CPU 是否小核，如果是的话把该 CPU 加入 slow 的 cpumask 位图中。这里判断是否小核主要是查表 `little_cores[]`，ARM32 处理器中 cortex-A7 是小核，ARM64 处理器中 Cortex-A53 是小核。HMP 调度器中目前只有两个调度域，即大核调度域和小核调度域，比内核默认的负载均衡里的 CPU 拓扑关系要简单多了。

2. HMP 负载调度

HMP 调度器同样使用内核中 Per-entity 的负载计算方法，另外它还定义了额外的两个负载变量 `load_avg_ratio` 和 `usage_avg_sum`。`load_avg_ratio` 和内核中 `load_avg_contrib` 计算方法类似，但是它没有乘以调度实体的实际权重，而是用 nice 为 0 的权重，因此它是进程可运行时间的一个比率。

$$sa.load_avg_ratio = \frac{sa.runnable_sum * NICE_0_LOAD}{sa.runnable_period}$$

`usage_avg_sum` 是表示进程处于运行状态的负载。

```

0 static void run_rebalance_domains(struct softirq_action *h)
1 {
2     int this_cpu = smp_processor_id();
3     struct rq *this_rq = cpu_rq(this_cpu);
4     enum cpu_idle_type idle = this_rq->idle_balance ?
5         CPU_IDLE : CPU_NOT_IDLE;
6
7 #ifdef CONFIG_SCHED_HMP
8     /* shortcut for hmp idle pull wakeups */
9     if (unlikely(this_rq->wake_for_idle_pull)) {
10         this_rq->wake_for_idle_pull = 0;
11         if (hmp_idle_pull(this_cpu)) {
12             /* break out unless running nohz idle as well */
13             if (idle != CPU_IDLE)
14                 return;
15         }
16     }
17 #endif
18     hmp_force_up_migration(this_cpu);
19     rebalance_domains(this_cpu, idle);
20     ...
21 }

```

第 7~17 行这里判断当前的运行队列的 `wake_for_idle_pull` 这个变量，这段代码我们先暂时不看，稍后在回来看这个。需要注意的是 HMP 调度器定义了一个 `CONFIG_DISABLE_CPU_SCHED_DOMAIN_BALANCE` 宏，该宏的意思是不想执行内核默认的负载均衡调度器，因此如果定义了该宏，那么 `SD_LOAD_BALANCE` 标志位为 0，`rebalance_domains()` 函数其实不会执行的。

`hmp_force_up_migration()` 函数比较长，我们分段来阅读。

[`run_rebalance_domains()`->`hmp_force_up_migration()`]

```

0 /*
1  * hmp_force_up_migration checks runqueues for tasks that need to
2  * be actively migrated to a faster cpu.
3  */
4 static void hmp_force_up_migration(int this_cpu)
5 {
6     int cpu, target_cpu;
7     struct sched_entity *curr, *orig;
8     struct rq *target;
9     unsigned long flags;
10    unsigned int force, got_target;
11    struct task_struct *p;
12
13    if (!spin_trylock(&hmp_force_migration))
14        return;
15    for_each_online_cpu(cpu) {
16        force = 0;
17        got_target = 0;
18        target = cpu_rq(cpu);
19        raw_spin_lock_irqsave(&target->lock, flags);
20        curr = target->cfs.curr;
21        if (!curr || target->active_balance) {
22            raw_spin_unlock_irqrestore(&target->lock, flags);

```

```

23         continue;
24     }
25
26     orig = curr;
27     curr = hmp_get_heaviest_task(curr, -1);
28     if (!curr) {
29         raw_spin_unlock_irqrestore(&target->lock, flags);
30         continue;
31     }
32     p = task_of(curr);
33     if (hmp_up_migration(cpu, &target_cpu, curr)) {
34         cpu_rq(target_cpu)->wake_for_idle_pull = 1;
35         raw_spin_unlock_irqrestore(&target->lock, flags);
36         spin_unlock(&hmp_force_migration);
37         smp_send_reschedule(target_cpu);
38         return;
39     }

```

`hmp_force_migration` 是一个 HMP 定义的锁。`for_each_online_cpu()` 从头开始遍历 `cpu_online_mask` 上的所有的 CPU，首先检查该 CPU 上的当前运行的调度实体或者该 CPU 正在做负载均衡，那么跳过该 CPU。

[`run_rebalance_domains()`->`hmp_force_up_migration()`->`hmp_get_heaviest_task()`]

```

0 static struct sched_entity *hmp_get_heaviest_task(
1     struct sched_entity *se, int target_cpu)
2 {
3     int num_tasks = hmp_max_tasks;
4     struct sched_entity *max_se = se;
5     unsigned long int max_ratio = se->avg.load_avg_ratio;
6     const struct cpumask *hmp_target_mask = NULL;
7     struct hmp_domain *hmp;
8
9     if (hmp_cpu_is_fastest(cpu_of(se->cfs_rq->rq)))
10         return max_se;
11
12     hmp = hmp_faster_domain(cpu_of(se->cfs_rq->rq));
13     hmp_target_mask = &hmp->cpus;
14     if (target_cpu >= 0) {
15         /* idle_balance gets run on a CPU while
16          * it is in the middle of being hotplugged
17          * out. Bail early in that case.
18          */
19         if (!cpumask_test_cpu(target_cpu, hmp_target_mask))
20             return NULL;
21         hmp_target_mask = cpumask_of(target_cpu);
22     }
23     /* The currently running task is not on the runqueue */
24     se = __pick_first_entity(cfs_rq_of(se));
25
26     while (num_tasks && se) {
27         if (entity_is_task(se) &&
28             se->avg.load_avg_ratio > max_ratio &&
29             cpumask_intersects(hmp_target_mask,
30                 tsk_cpus_allowed(task_of(se)))) {
31             max_se = se;
32             max_ratio = se->avg.load_avg_ratio;
33         }
34         se = __pick_next_entity(se);
35         num_tasks--;

```

```

36 }
37 return max_se;
38}

```

`hmp_get_heaviest_task()` 查找并返回该 CPU 上最繁忙的进程，该函数参数 `se` 是该 CPU 的当前进程。`hmp_cpu_is_fastest()` 判断该 CPU 是否处在大核 CPU 的调度域中，如果是直接返回当前进程的调度实体。第 12~13 行 `hmp_target_mask` 指向大核调度域中 `cpumask` 位图。第 14~22 行判断 `target_cpu` 是否在大核调度域中。第 26~36 行这里从该 CPU 运行队列的红黑树中最左边开始比较 `hmp_max_tasks` 个进程并且取出进程中负载最大的一个（`se->avg.load_avg_ratio`），然后返回这个负载最大的调度实体 `curr`。

`hmp_force_up_migration()` 函数的第 33 行判断刚才取得的最大负载的调度实体 `curr` 是否需要迁移到大核 CPU 上。

```

[run_rebalance_domains()->hmp_force_up_migration()-
>hmp_up_migration()]

0 /* Check if task should migrate to a faster cpu */
1 static unsigned int hmp_up_migration(int cpu, int *target_cpu, struct sched_entity
  *se)
2 {
3     struct task_struct *p = task_of(se);
4     int temp_target_cpu;
5     u64 now;
6
7     if (hmp_cpu_is_fastest(cpu))
8         return 0;
9
10 #ifdef CONFIG_SCHED_HMP_PRIO_FILTER
11     /* Filter by task priority */
12     if (p->prio >= hmp_up_prio)
13         return 0;
14 #endif
15     if (!hmp_task_eligible_for_up_migration(se))
16         return 0;
17
18     /* Let the task load settle before doing another up migration */
19     /* hack - always use clock from first online CPU */
20     now = cpu_rq(cpumask_first(cpu_online_mask))->clock_task;
21     if (((now - se->avg.hmp_last_up_migration) >> 10)
22         < hmp_next_up_threshold)
23         return 0;
24
25     /* hmp_domain_min_load only returns 0 for an
26      * idle CPU or 1023 for any partly-busy one.
27      * Be explicit about requirement for an idle CPU.
28      */
29     if (hmp_domain_min_load(hmp_faster_domain(cpu), &temp_target_cpu,
30         tsk_cpus_allowed(p)) == 0 && temp_target_cpu != NR_CPUS) {
31         if(target_cpu)
32             *target_cpu = temp_target_cpu;
33         return 1;
34     }
35     return 0;
36}

```

首先判断该 CPU 是否在大核调度域中，如果已经在大核调度域中，那就没有必要迁移繁忙的进程到大核 CPU 中。`hmp_up_prio` 是用来过滤优先级大于该值的进程，如果该进程优先

级大于 hmp_up_prio 也没必要在迁移到大核 CPU 上，这个要打开 CONFIG_SCHED_HMP_PRIO_FILTER 这个功能才有的，注意优先级是数值越低优先级越高。第 15 行的 hmp_task_eligible_for_up_migration() 函数其实是判断该进程的平均负载是否 hmp_up_threshold 这个阈值，hmp_up_threshold 也是一个过滤作用。这里有两个过滤，一个优先级，另外一个平均负载 (load_avg_ratio)。第 20~23 行做一个时间上过滤，该进程上一次迁移离现在的时间间隔小于 hmp_next_up_threshold 阈值也不需要迁移，避免进程被迁移来迁移去。第 29~34 行是查找大核调度域中是否有空闲 CPU 即 idle cpu。

[run_rebalance_domains()->hmp_force_up_migration()->hmp_up_migration()->hmp_domain_min_load()]

```

0 static inline unsigned int hmp_domain_min_load(struct hmp_domain *hmpd,
1         int *min_cpu, struct cpumask *affinity)
2 {
3     int cpu;
4     int min_cpu_runnable_temp = NR_CPUS;
5     u64 min_target_last_migration = ULLONG_MAX;
6     u64 curr_last_migration;
7     unsigned long min_runnable_load = INT_MAX;
8     unsigned long contrib;
9     struct sched_avg *avg;
10    struct cpumask temp_cpumask;
11    /*
12     * only look at CPUs allowed if specified,
13     * otherwise look at all online CPUs in the
14     * right HMP domain
15     */
16    cpumask_and(&temp_cpumask, &hmpd->cpus, affinity ? affinity :
17    cpu_online_mask);
18    for_each_cpu_mask(cpu, temp_cpumask) {
19        avg = &cpu_rq(cpu)->avg;
20        /* used for both up and down migration */
21        curr_last_migration = avg->hmp_last_up_migration ?
22        avg->hmp_last_up_migration : avg->hmp_last_down_migration;
23
24        contrib = avg->load_avg_ratio;
25        /*
26         * Consider a runqueue completely busy if there is any load
27         * on it. Definitely not the best for overall fairness, but
28         * does well in typical Android use cases.
29         */
30        if (contrib)
31            contrib = 1023;
32
33        if ((contrib < min_runnable_load) ||
34            (contrib == min_runnable_load &&
35             curr_last_migration < min_target_last_migration)) {
36            /*
37             * if the load is the same target the CPU with
38             * the longest time since a migration.
39             * This is to spread migration load between
40             * members of a domain more evenly when the
41             * domain is fully loaded
42             */
43            min_runnable_load = contrib;
44            min_cpu_runnable_temp = cpu;
45            min_target_last_migration = curr_last_migration;
46        }
47    }

```



```

48
49 if (min_cpu)
50     *min_cpu = min_cpu_runnable_temp;
51
52 return min_runnable_load;
53}

```

`hmp_domain_min_load()`函数有 3 个参数，`hmpd` 是传进来的 HMP 调度域，在我们上下文中是大核调度域，`min_cpu` 是一个指针变量用来传递结果给调用者，`affinity` 是另外一个 `cpumask` 位图在我们上下文中是刚才讨论的进程可以运行的 CPU 位图，进程通常都允许在所有 CPU 上运行。注意该函数如果返回 0 表示找到空闲 CPU，如果返回 1023 表示该调度域没有空闲 CPU 也就是都在繁忙中。第 16 行是 `hmpd` 调度域上 `cpumaks` 与上 `affinity` 位图。第 18 行就是遍历这个 `cpumask` 位图上的 CPU，如果该 CPU 上有负载（`load_avg_ratio`）那么 `contrib` 统统设置为 1023，为何这样呢？刚才说过该函数的目的是找一个空闲 CPU，进入当前 CPU 的有负载，说明不悠闲，因此这里统一设置 1023，仅仅是为了表示该 CPU 不是空闲而已。如果有多个 CPU 的 `contrib` 值相同，那么选择该调度域中最近一个发生过迁移的 CPU（`least-recently-disturbed`）。

回到 `hmp_force_up_migration()`函数中，`hmp_up_migration()`返回 1 表示在大核调度域中找到一个空闲的 CPU 即 `target_cpu`，然后设置 `target_cpu` 运行队列上 `wake_for_idle_pull` 标志位。回想在 `run_rebalance_domains()`函数最开头首先判断当前 CPU 运行队列的 `wake_for_idle_pull` 标志位，现在我们明白了该标志位是说有一个小核调度域上的比较繁忙的进程并且大核调度域上也有一个空闲 CPU，那么这样正好可以适合把该进程迁移到大核的空闲 CPU 上，注意不是现在迁移喔，是要等到该进程对应的 CPU 运行到 `run_rebalance_domains()`函数的时候才会去做迁移。`smp_send_reschedule()`函数发送一个 `IPI_RESCHEDULE` 的 IPI 中断给 `target_cpu`。

刚才那个 CPU 真是小幸运，正好它是小核上的 CPU 并且有合适迁移到大核上的进程，最重要的是大核调度域上有空闲的 CPU，这叫作无巧不成书。我们下面看看没那么好运气的其他 CPU 的情况。

[`hmp_force_up_migration()`]

```

41     if (!got_target) {
42         /*
43          * For now we just check the currently running task.
44          * Selecting the lightest task for offloading will
45          * require extensive book keeping.
46          */
47         curr = hmp_get_lightest_task(orig, 1);
48         p = task_of(curr);
49         target->push_cpu = hmp_offload_down(cpu, curr);
50         if (target->push_cpu < NR_CPUS) {
51             get_task_struct(p);
52             target->migrate_task = p;
53             got_target = 1;
54             trace_sched_hmp_migrate(p, target->push_cpu,
HMP_MIGRATE_OFFLOAD);
55             hmp_next_down_delay(&p->se, target->push_cpu);
56         }
57     }
58     /*

```



```

59      * We have a target with no active_balance.  If the task
60      * is not currently running move it, otherwise let the
61      * CPU stopper take care of it.
62      */
63      if (got_target) {
64          if (!task_running(target, p)) {
65              trace_sched_hmp_migrate_force_running(p, 0);
66              hmp_migrate_runnable_task(target);
67          } else {
68              target->active_balance = 1;
69              force = 1;
70          }
71      }
72
73      raw_spin_unlock_irqrestore(&target->lock, flags);
74
75      if (force)
76          stop_one_cpu_nowait(cpu_of(target),
77                              hmp_active_task_migration_cpu_stop,
78                              target, &target->active_balance_work);
79  }
80  spin_unlock(&hmp_force_migration);
81}

```

第 47 行的 `hmp_get_lightest_task()` 函数是查找当前 cpu 运行队列上负载比较轻的调度实体。注意 `orig` 是 for 循环里的 CPU 的当前运行进程。

[run_rebalance_domains()->hmp_force_up_migration()->hmp_get_lightest_task()]

```

0 static struct sched_entity *hmp_get_lightest_task(
1     struct sched_entity *se, int migrate_down)
2 {
3     int num_tasks = hmp_max_tasks;
4     struct sched_entity *min_se = se;
5     unsigned long int min_ratio = se->avg.load_avg_ratio;
6     const struct cpumask *hmp_target_mask = NULL;
7
8     if (migrate_down) {
9         struct hmp_domain *hmp;
10        if (hmp_cpu_is_slowest(cpu_of(se->cfs_rq->rq)))
11            return min_se;
12        hmp = hmp_slower_domain(cpu_of(se->cfs_rq->rq));
13        hmp_target_mask = &hmp->cpus;
14    }
15    /* The currently running task is not on the runqueue */
16    se = __pick_first_entity(cfs_rq_of(se));
17
18    while (num_tasks && se) {
19        if (entity_is_task(se) &&
20            (se->avg.load_avg_ratio < min_ratio &&
21             hmp_target_mask &&
22             cpumask_intersects(hmp_target_mask,
23                               tsk_cpus_allowed(task_of(se))))) {
24            min_se = se;
25            min_ratio = se->avg.load_avg_ratio;
26        }
27        se = __pick_next_entity(se);
28        num_tasks--;
29    }

```

```

30 return min_se;
31}

```

`hmp_get_lightest_task()`函数和 `hmp_get_heaviest_task()`函数类似，返回 `se` 调度实体对应的运行队列中任务最轻的调度实体 `min_se`。

回到 `hmp_force_up_migration()`函数中第 49 行的 `hmp_offload_down()`函数是查询刚才找到的最轻负载的进程能迁移到那里去，返回迁移目标 CPU – `target_cpu`。如果返回值是 `NR_CPUS`，表示没有找到合适的迁移目标 CPU。

[run_rebalance_domains()->hmp_force_up_migration()->hmp_offload_down()]

```

0 static inline unsigned int hmp_offload_down(int cpu, struct sched_entity *se)
1 {
2     int min_usage;
3     int dest_cpu = NR_CPUS;
4
5     if (hmp_cpu_is_slowest(cpu))
6         return NR_CPUS;
7
8     /* Is there an idle CPU in the current domain */
9     min_usage = hmp_domain_min_load(hmp_cpu_domain(cpu), NULL, NULL);
10    if (min_usage == 0) {
11        trace_sched_hmp_offload_abort(cpu, min_usage, "load");
12        return NR_CPUS;
13    }
14
15    /* Is the task alone on the cpu? */
16    if (cpu_rq(cpu)->cfs.h_nr_running < 2) {
17        trace_sched_hmp_offload_abort(cpu,
18            cpu_rq(cpu)->cfs.h_nr_running, "nr_running");
19        return NR_CPUS;
20    }
21
22    /* Is the task actually starving? */
23    /* >=25% ratio running/runnable = starving */
24    if (hmp_task_starvation(se) > 768) {
25        trace_sched_hmp_offload_abort(cpu, hmp_task_starvation(se),
26            "starvation");
27        return NR_CPUS;
28    }
29
30    /* Does the slower domain have any idle CPUs? */
31    min_usage = hmp_domain_min_load(hmp_slower_domain(cpu), &dest_cpu,
32        tsk_cpus_allowed(task_of(se)));
33
34    if (min_usage == 0) {
35        trace_sched_hmp_offload_succeed(cpu, dest_cpu);
36        return dest_cpu;
37    } else
38        trace_sched_hmp_offload_abort(cpu, min_usage, "slowdomain");
39    return NR_CPUS;
40}

```

参数 `cpu` 是 `for` 循环遍历到的 CPU，`se` 是刚才找到该 CPU 上负载比较轻的进程。如果该 CPU 已经在小核调度域中，那么不用迁移。第 9 行，既然已经判断该 CPU 不在小核调度域中，那必然是在大核调度域中，因为目前 HMP 只支持 2 个 HMP 调度域。

`hmp_domain_min_load()`函数刚才我们分析过，它是查找调度域中是否有空闲 CPU，返回 0 表

示有空闲 CPU。如果该 CPU 所在的大核调度域里有空闲 CPU，那么也不做迁移。第 16 行该 CPU 的运行队列中正在运行的进程只有一个或者没有，那么也不需要迁移。

hmp_task_starvation()判断当前进程是否饥饿，判断条件公式如下。

$$starving = \frac{running_avg_sum}{runnable_avg_sum}$$

当 starving > 75% 的时候说明该进程一直渴望获得更多的 CPU 时间，这样的进程也不适合迁移。

第 31~38 行查找小核调度域中是否有空闲 CPU，如果有的话该函数返回该空闲 CPU，如果返回 NR_CPUS 说明没找到合适的 CPU 用做迁移目的地。

回到 hmp_force_up_migration() 函数，第 50~56 行刚才找到负载最轻的进程当做迁移进程 target->migrate_task，hmp_next_down_delay() 更新迁移 CPU 和迁移目的地 CPU 的相关信息，调度实体中 hmp_last_down_migration 和 hmp_last_up_migration 记录现在时刻的时间。

如果要迁移进程 p 没有正在运行即 p->on_cpu=0，那么就迁移吧。

```
[run_rebalance_domains()->hmp_force_up_migration()->hmp_migrate_runnable_task()]
0 static void hmp_migrate_runnable_task(struct rq *rq)
1 {
2     struct sched_domain *sd;
3     int src_cpu = cpu_of(rq);
4     struct rq *src_rq = rq;
5     int dst_cpu = rq->push_cpu;
6     struct rq *dst_rq = cpu_rq(dst_cpu);
7     struct task_struct *p = rq->migrate_task;
8     /*
9      * One last check to make sure nobody else is playing
10     * with the source rq.
11     */
12     if (src_rq->active_balance)
13         goto out;
14
15     if (src_rq->nr_running <= 1)
16         goto out;
17
18     if (task_rq(p) != src_rq)
19         goto out;
20     /*
21     * Not sure if this applies here but one can never
22     * be too cautious
23     */
24     BUG_ON(src_rq == dst_rq);
25
26     double_lock_balance(src_rq, dst_rq);
27
28     rcu_read_lock();
29     for_each_domain(dst_cpu, sd) {
30         if (cpumask_test_cpu(src_cpu, sched_domain_span(sd)))
31             break;
32     }
33
34     if (likely(sd)) {
35         struct lb_env env = {
36             .sd = sd,
37             .dst_cpu = dst_cpu,
38             .dst_rq = dst_rq,
```

```

39         .src_cpu      = src_cpu,
40         .src_rq        = src_rq,
41         .idle          = CPU_IDLE,
42     };
43
44     schedstat_inc(sd, alb_count);
45
46     if (move_specific_task(&env, p))
47         schedstat_inc(sd, alb_pushed);
48     else
49         schedstat_inc(sd, alb_failed);
50 }
51
52 rcu_read_unlock();
53 double_unlock_balance(src_rq, dst_rq);
54out:
55 put_task_struct(p);
56}

```

- 迁移进程是之前找到的那个负载比较轻的进程 `migrate_task`。
- 迁移源 CPU 是 `for` 循环遍历到的 CPU，
- 迁移目的地 CPU 是在小核调度域中找到的空闲 CPU - `rq->push_cpu`。

这里和内核默认的负载均衡调度器的 `load_balance()` 函数一样使用 `struct lb_env` 结构体来描述刚才那些信息。迁移的动作是 `move_specific_task()` 函数。`move_specific_task()` 函数的实现和 `load_balance()` 函数里实现的类似。

回到 `hmp_force_up_migration()` 函数中第 67~70 行如果该迁移进程正在运行，那么就会调用 `stop_one_cpu_nowait()` 函数来暂停迁移源 CPU 后强行迁移了。

我们回到 HMP 调度器最开始的函数 `run_rebalance_domains()` 第 7~17 行，我们现在知道了 `wake_for_idle_pull` 标志位的含义了，它是小核调度域上有一个合适迁移到大核上的进程并且大核调度域上有空闲的 CPU。

[`run_rebalance_domains()->hmp_idle_pull()`]

```

0 /*
1  * hmp_idle_pull looks at little domain runqueues to see
2  * if a task should be pulled.
3  *
4  * Reuses hmp_force_migration spinlock.
5  *
6  */
7 static unsigned int hmp_idle_pull(int this_cpu)
8 {
9     int cpu;
10     struct sched_entity *curr, *orig;
11     struct hmp_domain *hmp_domain = NULL;
12     struct rq *target = NULL, *rq;
13     unsigned long flags, ratio = 0;
14     unsigned int force = 0;
15     struct task_struct *p = NULL;
16
17     if (!hmp_cpu_is_slowest(this_cpu))
18         hmp_domain = hmp_slower_domain(this_cpu);
19     if (!hmp_domain)
20         return 0;
21

```

```

22 if (!spin_trylock(&hmp_force_migration))
23     return 0;
24
25 /* first select a task */
26 for_each_cpu(cpu, &hmp_domain->cpus) {
27     rq = cpu_rq(cpu);
28     raw_spin_lock_irqsave(&rq->lock, flags);
29     curr = rq->cfs.curr;
30     if (!curr) {
31         raw_spin_unlock_irqrestore(&rq->lock, flags);
32         continue;
33     }
34     orig = curr;
35     curr = hmp_get_heaviest_task(curr, this_cpu);
36     /* check if heaviest eligible task on this
37      * CPU is heavier than previous task
38      */
39     if (curr && hmp_task_eligible_for_up_migration(curr) &&
40         curr->avg.load_avg_ratio > ratio &&
41         cpumask_test_cpu(this_cpu,
42             tsk_cpus_allowed(task_of(curr)))) {
43         p = task_of(curr);
44         target = rq;
45         ratio = curr->avg.load_avg_ratio;
46     }
47     raw_spin_unlock_irqrestore(&rq->lock, flags);
48 }

```

参数 `this_cpu` 应该是大核调度域上的 CPU。第 26~48 行 `for` 循环遍历小核调度域上所有的 CPU，然后找出该 CPU 的运行队列中负载最重的进程 `curr`，并且首先判断这个负载重的进程是否合适迁移到大核 CPU 上见 `hmp_task_eligible_for_up_migration()` 函数。这里比较小核调度域上所有 CPU 的最重负载进程找出最重的一个。进程间负载轻重的比较是通过 `load_avg_ratio` 这个变量。

[hmp_idle_pull()]

```

50 /* now we have a candidate */
51 raw_spin_lock_irqsave(&target->lock, flags);
52 if (!target->active_balance && task_rq(p) == target) {
53     get_task_struct(p);
54     target->push_cpu = this_cpu;
55     target->migrate_task = p;
56     trace_sched_hmp_migrate(p, target->push_cpu, HMP_MIGRATE_IDLE_PULL);
57     hmp_next_up_delay(&p->se, target->push_cpu);
58     /*
59      * if the task isn't running move it right away.
60      * Otherwise setup the active_balance mechanic and let
61      * the CPU stopper do its job.
62      */
63     if (!task_running(target, p)) {
64         trace_sched_hmp_migrate_idle_running(p, 0);
65         hmp_migrate_runnable_task(target);
66     } else {
67         target->active_balance = 1;
68         force = 1;
69     }
70 }
71 raw_spin_unlock_irqrestore(&target->lock, flags);
72
73 if (force) {

```

```

74     /* start timer to keep us awake */
75     hmp_cpu_keepalive_trigger();
76     stop_one_cpu_nowait(cpu_of(target),
77         hmp_active_task_migration_cpu_stop,
78         target, &target->active_balance_work);
79 }
80done:
81 spin_unlock(&hmp_force_migration);
82 return force;
83}

```

我们找到一个合适的迁移进程之后就可以开始迁移了。

- 迁移进程 `migrate_task` 是刚才找到的 `curr` 进程
- 迁移源 CPU: 迁移进程对应的 CPU
- 迁移目的地 CPU: 当前 CPU, 当前 CPU 是大核调度域中的一个

如果迁移进程正在运行, 那么和之前一样, 调用 `stop_one_cpu_nowait()` 函数强行迁移。

3 新创建的进程

在 HMP 调度器中对待新创建的进程会有特殊的处理。新创建的进程创建完成之后需要把进程添加到合适的运行队列中, 这个过程中会调用 `select_task_rq()` 函数来选择一个最合适新进程运行的 CPU。

```

[wake_up_new_task()->select_task_rq()->select_task_rq_fair()]
0 static int
1 select_task_rq_fair(struct task_struct *p, int sd_flag, int wake_flags)
2 {
3     struct sched_domain *tmp, *affine_sd = NULL, *sd = NULL;
4     int cpu = smp_processor_id();
5     int prev_cpu = task_cpu(p);
6     int new_cpu = cpu;
7     int want_affine = 0;
8     int sync = wake_flags & WF_SYNC;
9
10    if (p->nr_cpus_allowed == 1)
11        return prev_cpu;
12
13#ifdef CONFIG_SCHED_HMP
14    /* always put non-kernel forking tasks on a big domain */
15    if (unlikely(sd_flag & SD_BALANCE_FORK) && hmp_task_should_forkboost(p)) {
16        new_cpu = hmp_select_faster_cpu(p, prev_cpu);
17        if (new_cpu != NR_CPUS) {
18            hmp_next_up_delay(&p->se, new_cpu);
19            return new_cpu;
20        }
21        /* failed to perform HMP fork balance, use normal balance */
22        new_cpu = cpu;
23    }
24#endif
25
26    . . .
27}

```

第 13~14 行对于新创建的进程并且该进程是用户进程, 那么就调用 `hmp_select_faster_cpu()` 函数来选择一个最合适的大核调度域上的 CPU。也就是说新创建的用户进程首先会在大核 CPU 上运行了。

4 总结

HMP 调度器的实现可以简单概况为：

- 把小核调度域上“大活”迁移到大核调度域的空闲 CPU 上
- 把每个大核 CPU 上“小活”迁移到小核调度域的空闲 CPU 上

大活就是负载比较重的进程，小活就是负载比较轻的进程。如何判断进程是大活还是小活呢？HMP 采用 `load_avg_ratio` 来比较，`load_avg_ratio` 的计算公式之前已经描述过，它并没有像内核中采用的 `load_avg_contrib` 一样考虑进程的可运行时间比重

（`runnable_sum/unable_period`）和实际权重，在 HMP 调度器眼里只考虑进程的可运行时间比重。那么 CPU 密集型的进程以及长时间运行的进程容易理解为大活，那些间隙性运行的进程就变成小活了，即便它优先级很高。比如一个优先级很高的进程，它只是间歇性的运行，那么它是没机会在大核上遛弯的。

另外 HMP 调度器还定义了 `hmp_up_threshold`（700）和 `hmp_down_threshold`（512），那么也就是说可运行时间比重（`runnable_sum/unable_period`）小于 50% 就认为是小活，大于 68.3% 就认为是大活。

HMP 调度器的实现比内核中自带的 CPU 负载均衡要简单的多，首先 HMP 调度器只定义了两个调度域，没有调度组和调度能力的概念，而且调度域没有层次。内核自带的负载均衡调度器可以根据 CPU 的物理属性来定义调度域的层次关系。

另外 HMP 调度器没有考虑调度域内以及调度域之间的负载均衡。HMP 调度器寄托在调度域中有空闲 CPU。假设小核上有进程突然持续的在使用 CPU，那么 `load_avg_ratio` 变大表示这个是大活，可是大核上暂时没有空闲 CPU 啊，那怎么办？

假设大小核调度域都没有空闲 CPU，那么谁来保证他们之间的负载均衡呢？有人说用系统默认的 CPU 负载均衡调度器啊。Linaro 上实现的 HMP 调度器是默认关闭了系统自带的 SMP 负载均衡的，即关闭 `CONFIG_DISABLE_CPU_SCHED_DOMAIN_BALANCE` 这个宏。如果开启的话会出现什么情况，那么 SMP 调度器就会考虑大小核调度域之间的负载均衡了，他们要负载均衡大致相等喔（假设不考虑大小核之间的能力系数 `capacity`），那相当于小核调度域也要和大核调度域干一样的活，那这样的话 big.LITTLE 模型就失去意义。另外两套调度器一起运行是否会冲突，即 HMP 迁移了进程，又被 SMP 调度器给迁移回来。

总之 HMP 调度器算不上完美，期待读者去优化了。

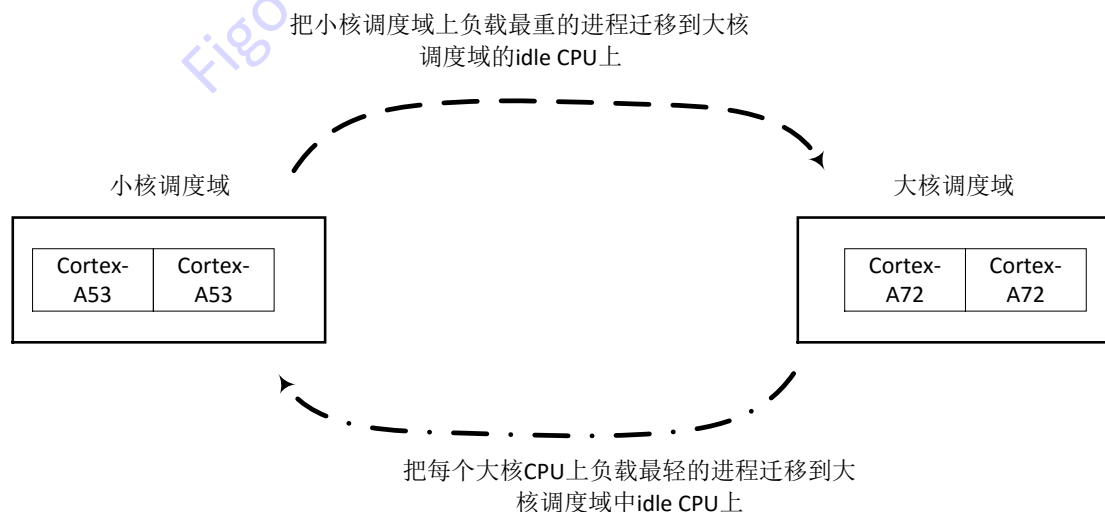


图 3.x HMP 调度器

