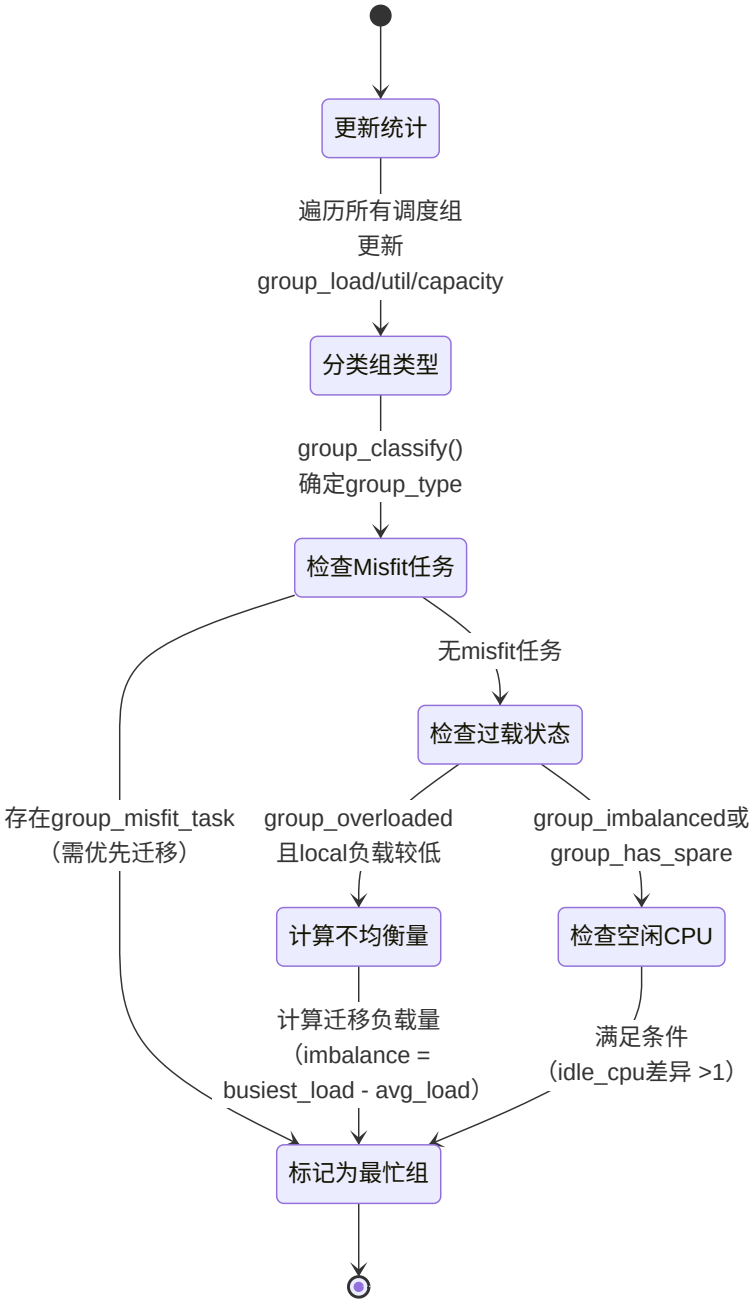
如何查找出一个调度域里最繁忙的调度组?

状态图



状态图说明

1. 更新统计（update_sd_lb_stats）

- **动作**：遍历调度域内所有调度组（sched group），更新：

```
group_load = Σ(cpu_load)       // 组总负载
group_util = Σ(cpu_util)       // 组总利用率
group_capacity = Σ(cpu_capacity) // 组总算力
```

- **输出**：生成 `struct sd_lb_stats`，包含各组的负载和算力信息。

2. 分类组类型（group_classify）

- **条件判断**：根据负载状态将组分为5类（优先级降序）：

```
enum group_type {
    group_overloaded,    // 过载（负载>算力）
    group_imbalanced,    // 因affinity不均衡
    group_misfit_task,   // 有misfit任务
    group_fully_busy,    // 满负荷但未过载
    group_has_spare      // 有空闲算力
};
```

3. 检查Misfit任务

- **优先级最高**：若组内有 `misfit_task` （任务所需算力 > 当前CPU能力）：

```
if (busiest->group_type == group_misfit_task) {
    env->migration_type = migrate_misfit;
    return busiest_group;  // 立即标记为最忙组
}
```

- **目的**：优先迁移这类任务以提升性能。

4. 检查过载状态

- **条件**：组类型为 `group_overloaded` 且本地组（local group）负载较低：

```
if (busiest->avg_load > local->avg_load * imbalance_pct/100) {
    calculate_imbalance(); // 计算需迁移的负载量
}
```

- **动作**：计算需迁移的负载量 `imbalance` ，确保迁移后不过载。

5. 检查空闲CPU

- **条件**：组类型为 `group_has_spare` 或 `group_imbalanced` ：

```
if (busiest->idle_cpus < local->idle_cpus) {
    // 迁移任务以平衡空闲CPU
    env->imbalance = (local->idle_cpus - busiest->idle_cpus) / 2;
}
```

- **目的**：通过任务迁移使各组的空闲CPU数量均衡。

6. 标记为最忙组

- **最终决策**：综合上述条件后，返回最符合条件的调度组：

```
return busiest_group;  // 可能为过载组、misfit组或空闲差异组
```

关键设计思想

1. **优先级分层**：Misfit任务 > 过载组 > 空闲CPU均衡
2. **动态阈值**：通过 `imbalance_pct` （默认117%）避免微小负载波动触发迁移
3. **能效导向**：在异构CPU架构中优先迁移到高算力集群（如大核）

如果一个调度域负载不均衡，请问如何计算需要迁移的负载量呢?

基于 `calculate_imbalance()` 函数的负载迁移量计算详解

以下是针对Linux内核 `calculate_imbalance()` 函数的逻辑分析，详细说明如何计算需要迁移的负载量：

1. 特殊场景处理（优先级最高）

a. Misfit任务（ `group_misfit_task` ）

- **场景**：存在算力不足的任务（如大核任务在小核运行）。
- **处理**：直接迁移一个任务。

```
if (busiest->group_type == group_misfit_task) {
    env->migration_type = migrate_misfit;
    env->imbalance = 1; // 强制迁移1个Misfit任务
    return;
}
```

b. 不对称打包（ `group_asym_packing` ）

- **场景**：异构CPU架构（如大小核），需将任务迁移到高算力CPU。
- **处理**：迁移所有可运行任务。

```
if (busiest->group_type == group_asym_packing) {
    env->migration_type = migrate_task;
    env->imbalance = busiest->sum_h_nr_running; // 迁移所有CFS任务
    return;
}
```

c. 亲和性不均衡（ `group_imbalanced` ）

- **场景**：因任务亲和性（ `cpus_ptr` ）导致无法均衡。
- **处理**：迁移一个任务以打破僵局。

```
if (busiest->group_type == group_imbalanced) {
    env->migration_type = migrate_task;
    env->imbalance = 1; // 迁移1个任务
    return;
}
```

2. 本地组有空闲算力（ `group_has_spare` ）

当本地组（ `local` ）有空闲资源时，尝试拉取负载：

a. 最忙组过载（ `group_overloaded` ）

- **目标**：利用本地空闲算力，避免本地过载。
- **计算逻辑**：

$$\text{imbalance} = \max(\text{local\_capacity}, \text{local\_util}) - \text{local\_util}$$

  - **代码**：

```
if (local->group_type == group_has_spare && busiest->group_type > group_fully_busy) {
    env->migration_type = migrate_util;
    env->imbalance = max(local->group_capacity, local->group_util) - local->group_util;
}
```

b. 最忙组未过载

- **目标**：平衡空闲CPU或任务数。
  - **单CPU调度组（如MC域）**：

```
if (busiest->group_weight == 1) { // Base domain（单CPU组）
    unsigned int nr_diff = busiest->sum_nr_running - local->sum_nr_running;
    env->imbalance = nr_diff >> 1; // 迁移任务数差的一半
}
```

- **多CPU调度组（如DIE域）**：

```
env->imbalance = max_t(long, 0, (local->idle_cpus - busiest->idle_cpus) >> 1);
```

3. 本地组过载或即将过载

当本地组即将过载时，需谨慎迁移：

a. 本地组负载过高

- **条件**：本地组平均负载 ≥ 最忙组或调度域平均负载。
- **处理**：放弃迁移。

```
if (local->avg_load >= busiest->avg_load || local->avg_load >= sds->avg_load) {
    env->imbalance = 0;
    return;
}
```

b. 双组过载

- **目标**：通过迁移负载使两组趋近调度域平均负载。
- **计算公式**：

$$\text{imbalance} = \frac{\min\left((\text{busiest\_avg\_load} - \text{avg\_load}) \times \text{busiest\_capacity},\ (\text{avg\_load} - \text{local\_avg\_load}) \times \text{local\_capacity}\right)}{\text{SCHED\_CAPACITY\_SCALE}}$$

- **代码**：

```
env->migration_type = migrate_load;
env->imbalance = min(
    (busiest->avg_load - sds->avg_load) * busiest->group_capacity,
    (sds->avg_load - local->avg_load) * local->group_capacity
) / SCHED_CAPACITY_SCALE;
```

4. 关键参数与设计思想

- **归一化负载**：通过 `SCHED_CAPACITY_SCALE`（通常为1024）消除算力差异。

```
local->avg_load = (local->group_load * SCHED_CAPACITY_SCALE) / local->group_capacity;
sds->avg_load = (sds->total_load * SCHED_CAPACITY_SCALE) / sds->total_capacity;
```

- **动态阈值**： `imbalance_pct` （默认117%）避免微小负载波动触发迁移。
- **任务迁移上限**： `sysctl_sched_nr_migrate` （默认32）限制单次迁移任务数。

示例场景

假设调度域包含两个集群：

- **小核集群**： `group_load=600` ， `group_capacity=800` → `avg_load=768` 。
- **大核集群**： `group_load=800` ， `group_capacity=1600` → `avg_load=512` 。

计算步骤：

1. **判断过载**：小核组负载（768）> 大核组负载（512）×117% → 触发迁移。
2. **计算迁移量**：

$$\text{imbalance} = \frac{(768 - 512) \times 800}{1024} = 200 \text{ （负载单位）}$$

3. **迁移任务**：将累计负载值达到200的任务从大核组迁移到小核组。

总结

`calculate_imbalance()` 的核心逻辑是：

1. **优先级分层**：Misfit任务 > 亲和性不均衡 > 过载组 > 空闲资源均衡。
2. **动态计算**：根据归一化负载、算力差异和系统平均负载动态调整迁移量。
3. **稳定性控制**：通过阈值（ `imbalance_pct` ）和任务数上限避免过度迁移。

通过这一机制，Linux内核能够在异构CPU架构下高效平衡负载，同时兼顾性能和能效。

如果使用内核提供的唤醒进程接口函数（如 wake_up_process()）来唤醒一个进程，那么进程唤醒后应该在哪个 CPU 上运行呢？是调用 wake_up_process() 的那个 CPU，还是之前运行该进程的那个 CPU，或者是其他 CPU 呢？

进程唤醒后的CPU选择逻辑

当使用 `wake_up_process()` 唤醒一个进程时，目标CPU的选择由 **调度器策略** 和 **系统状态** 动态决定，具体流程如下：

1. 核心逻辑

内核通过 `select_task_rq_fair()` 函数选择目标CPU，优先级和条件如下：

1. **能量感知调度（EAS）**：
   - **条件**：系统未过载（ `!overutilized` ）且启用EAS。
   - **行为**：调用 `find_energy_efficient_cpu()` ，选择能效最优的CPU。
   - **公式**：综合各CPU算力、任务负载和能耗模型，选择能耗增量最小的CPU：

$$\text{Energy Cost} = \sum(\text{CPU Power} \times \text{Utilization})$$

2. **Wake Affinity（唤醒亲和性）**：
   - **条件**： `wake_wide(p)` 返回 `false` （任务与唤醒者关系紧密，如1:1唤醒模型）。
   - **行为**：优先选择 **`waker CPU`** 或 **`prev_cpu`** 的共享缓存（LLC）域内空闲CPU。
   - **代码关键点**：

```
want_affine = !wake_wide(p) && cpumask_test_cpu(cpu, p->cpus_ptr);
new_cpu = wake_affine(...); // 在waker和prev_cpu中选择
```

3. **快速路径（Fast Path）**：
   - **条件**：未启用EAS或Wake Affinity，但存在空闲CPU。
   - **行为**：调用 `select_idle_sibling()` ，优先在 **`prev_cpu`** 或 **`waker CPU`** 的LLC域内选择空闲CPU。
   - **示例**：
     - 若 `prev_cpu` 空闲 → 选择 `prev_cpu` 。
     - 若 `waker CPU` 空闲 → 选择 `waker CPU` 。
4. **慢速路径（Slow Path）**：
   - **条件**：快速路径未找到合适CPU。
   - **行为**：调用 `find_idlest_cpu()` ，在调度域内选择负载最低的CPU。
   - **公式**：比较各CPU的归一化负载：

$$\text{Load}_{\text{cpu}} = \frac{\text{Util}_{\text{cfs}} + \text{Util}_{\text{rt}} + \text{Util}_{\text{irq}}}{\text{Capacity}_{\text{cpu}}}$$

2. 关键结论

- **EAS优先**：轻载时优先考虑能效，选择能耗最低的CPU。
- **Wake Affinity**：若任务与唤醒者关系紧密（如1:1模型），优先选择共享缓存的CPU（ `prev_cpu` 或 `waker CPU` ）。
- **快速路径**：优先利用空闲CPU，减少调度延迟。
- **慢速路径**：负载均衡选择最闲CPU，避免局部过载。

最终目标CPU可能是 **`prev_cpu`** 、 **`waker CPU`** 或 **其他空闲/低负载CPU**，具体由系统状态和调度策略动态决定。

绿色节能调度器如何衡量一个进程的计算能力?

绿色节能调度器（EAS）如何衡量进程计算能力?

在 Linux 内核的 **Energy Aware Scheduling (EAS)** 中，进程的计算能力通过 **Energy Model (EM)** 和 **CPU 算力（Capacity）** 来衡量。以下是结合代码和原理的详细分析：

一、核心概念：Energy Model (EM)

EM 是 EAS 的基石，它将 CPU 的 **算力**、**频率** 和 **功耗** 映射为统一的性能模型。关键数据结构如下：

```
// linux-5.4.24/include/linux/energy_model.h

struct em_perf_state {
    unsigned long frequency;  // 频率（KHz）
    unsigned long power;      // 功耗（毫瓦）
    unsigned long capacity;   // 算力（归一化值）
};

struct em_perf_domain {
    struct em_perf_state *table; // 不同频率下的性能状态
    int nr_perf_states;          // 状态数量
    unsigned long max_capacity; // 最大算力（如小核为512）
};
```

每个 CPU 簇（Cluster）对应一个 `em_perf_domain` 。例如：

- **小核簇**：最大算力 512，频率范围 500MHz–1GHz。
- **大核簇**：最大算力 1024，频率范围 1GHz–2GHz。

二、进程计算能力的衡量方法

进程的计算能力通过其 **负载（Utilization）** 表示，负载是进程对 CPU 算力的需求。EAS 使用以下两种方式衡量：

1. **静态算力（Static Capacity）**
   每个 CPU 的算力是固定的，由硬件架构决定。例如：

```
// 小核的算力（归一化为1024中的512）
arch_scale_cpu_capacity(cpu) = 512;
```

2. **动态负载（Dynamic Utilization）**
   进程的负载通过 **PELT（Per-Entity Load Tracking）** 跟踪，反映其短期和长期 CPU 使用率：

```
// linux-5.4.24/kernel/sched/fair.c

struct sched_entity {
    struct load_weight load; // 权重（优先级）
    unsigned long runnable_weight; // 可运行状态权重
    u64 sum_exec_runtime;    // 累计运行时间
};
```

三、代码实现：EAS 如何选择 CPU？

EAS 在 `find_energy_efficient_cpu()` 中计算不同 CPU 的能耗，选择最优目标：

```
// linux-5.4.24/kernel/sched/fair.c

static int find_energy_efficient_cpu(...) {
    for_each_perf_domain(pd) { // 遍历所有性能域
        // 1. 计算未放置任务时的基础能耗
        base_energy = compute_energy(...);

        // 2. 找到候选 CPU（最大剩余算力或 prev_cpu）
        for_each_cpu(cpu, pd->cpus) {
            spare_cap = capacity_of(cpu) - cpu_util(cpu);
            if (spare_cap > max_spare_cap)
                max_spare_cpu = cpu;
        }

        // 3. 计算放置任务后的能耗
        energy_delta = compute_energy(p, max_spare_cpu) - base_energy;

        // 4. 选择能耗最低的 CPU
        if (energy_delta < best_energy_delta)
            best_energy_delta = energy_delta;
    }
    return best_cpu;
}
```

关键逻辑：

1. **算力匹配**：选择剩余算力（ `spare_cap` ）最大的 CPU，避免触发升频。
2. **能耗估算**：通过 `em_pd_energy()` 计算性能域的能耗：

```
// linux-5.4.24/kernel/sched/energy.c

static inline unsigned long em_pd_energy(...) {
    // 根据 sum_util 和 max_util 选择频点
    freq = map_util_to_freq(max_util, max_cap);

    // 计算该频点下的总功耗
    power = ps->power * (sum_util / scale_cpu);
    return power * time; // 总能耗
}
```

四、示例：进程放置的能耗计算

假设一个进程的负载为 **300**，系统有两个性能域：

1. **小核簇（Capacity=512）**：当前总负载 400，最高负载 400。
2. **大核簇（Capacity=1024）**：当前总负载 600，最高负载 600。

放置位置	新总负载	新最高负载	频点	功耗
小核簇	400+300=700	700	1.5GHz	700mW
大核簇	600+300=900	900	1.8GHz	900mW

EAS 选择小核簇，因为总功耗更低（700mW < 900mW）。

五、总结

绿色节能调度器通过以下方式衡量进程计算能力：

1. **静态算力**：CPU 硬件决定的归一化 Capacity。
2. **动态负载**：进程的 PELT 负载，反映其对 CPU 的需求。
3. **能耗模型**：通过 EM 计算不同 CPU 的能耗，选择最优目标。

EAS 的设计在异构处理器（如 Arm big.LITTLE）中显著优化了能效，同时保证系统吞吐量。

当一个进程被唤醒时，绿色节能调度器如何选择在哪个 CPU 上运行?

在 Linux 内核的 **Energy Aware Scheduling (EAS)** 中，进程的计算能力通过 **Energy Model (EM)** 和 **CPU 算力（Capacity）** 来衡量。以下是结合代码和原理的详细分析：

一、核心概念：Energy Model (EM)

EM 是 EAS 的基石，它将 CPU 的 **算力**、**频率** 和 **功耗** 映射为统一的性能模型。关键数据结构如下：

```
// linux-5.4.24/include/linux/energy_model.h

struct em_perf_state {
    unsigned long frequency;  // 频率（KHz）
    unsigned long power;      // 功耗（毫瓦）
    unsigned long capacity;   // 算力（归一化值）
};

struct em_perf_domain {
    struct em_perf_state *table; // 不同频率下的性能状态
    int nr_perf_states;          // 状态数量
    unsigned long max_capacity; // 最大算力（如小核为512）
};
```

每个 CPU 簇（Cluster）对应一个 `em_perf_domain`。例如：

- **小核簇**：最大算力 512，频率范围 500MHz–1GHz。
- **大核簇**：最大算力 1024，频率范围 1GHz–2GHz。

二、进程计算能力的衡量方法

进程的计算能力通过其 **负载（Utilization）** 表示，负载是进程对 CPU 算力的需求。EAS 使用以下两种方式衡量：

1. **静态算力（Static Capacity）**
   每个 CPU 的算力是固定的，由硬件架构决定。例如：

   ```
   // 小核的算力（归一化为1024中的512）
   arch_scale_cpu_capacity(cpu) = 512;
   ```

2. **动态负载（Dynamic Utilization）**
   进程的负载通过 **PELT（Per-Entity Load Tracking）** 跟踪，反映其短期和长期 CPU 使用率：

   ```
   // linux-5.4.24/kernel/sched/fair.c

   struct sched_entity {
       struct load_weight load; // 权重（优先级）
       unsigned long runnable_weight; // 可运行状态权重
       u64 sum_exec_runtime;    // 累计运行时间
   };
   ```

三、代码实现：EAS 如何选择 CPU？

EAS 在 `find_energy_efficient_cpu()` 中计算不同 CPU 的能耗，选择最优目标：

```
// linux-5.4.24/kernel/sched/fair.c

static int find_energy_efficient_cpu(...) {
    for_each_perf_domain(pd) { // 遍历所有性能域
        // 1. 计算未放置任务时的基础能耗
        base_energy = compute_energy(...);

        // 2. 找到候选 CPU（最大剩余算力或 prev_cpu）
        for_each_cpu(cpu, pd->cpus) {
            spare_cap = capacity_of(cpu) - cpu_util(cpu);
            if (spare_cap > max_spare_cap)
                max_spare_cpu = cpu;
        }

        // 3. 计算放置任务后的能耗
        energy_delta = compute_energy(p, max_spare_cpu) - base_energy;

        // 4. 选择能耗最低的 CPU
        if (energy_delta < best_energy_delta)
            best_energy_delta = energy_delta;
    }
    return best_cpu;
}
```

关键逻辑：

1. **算力匹配**：选择剩余算力（ `spare_cap` ）最大的 CPU，避免触发升频。
2. **能耗估算**：通过 `em_pd_energy()` 计算性能域的能耗：

```
// linux-5.4.24/kernel/sched/energy.c

static inline unsigned long em_pd_energy(...) {
    // 根据 sum_util 和 max_util 选择频点
    freq = map_util_to_freq(max_util, max_cap);

    // 计算该频点下的总功耗
    power = ps->power * (sum_util / scale_cpu);
    return power * time; // 总能耗
}
```

四、示例：进程放置的能耗计算

假设一个进程的负载为 **300**，系统有两个性能域：

1. **小核簇（Capacity=512）**：当前总负载 400，最高负载 400。
2. **大核簇（Capacity=1024）**：当前总负载 600，最高负载 600。

放置位置	新总负载	新最高负载	频点	功耗
小核簇	400+300=700	700	1.5GHz	700mW
大核簇	600+300=900	900	1.8GHz	900mW

EAS 选择小核簇，因为总功耗更低（700mW < 900mW）。

五、总结

绿色节能调度器通过以下方式衡量进程计算能力：

1. **静态算力**：CPU 硬件决定的归一化 Capacity。
2. **动态负载**：进程的 PELT 负载，反映其对 CPU 的需求。
3. **能耗模型**：通过 EM 计算不同 CPU 的能耗，选择最优目标。

EAS 的设计在异构处理器（如 Arm big.LITTLE）中显著优化了能效，同时保证系统吞吐量。