ARM培训EAS文档总结

一、load balance：

1. load balance也增加workload，比如cache失效，调度时延，增加功耗消耗等，所以策略很重要
2. EAS调度器load balance：

When cpuidle：

1. Idle balance
2. **NOHZ balance？？？**

When load balance interval is due（负载均衡间隔到了）

**1. Rebalance tick ？？？**

When fork，execute or wakeup a task

1. SD\_BALANCE\_FORK
2. SD\_BALANCE\_EXEC
3. SD\_BALANCE\_WAKE
4. **Misfit task ?????**

**二、PELT（Per Entity Load Tracing）**

1. PELT对CFS是有益处的：

* CFS在（> 3.7 version）之后可以在每个run queue里面追踪load
* 在一段时间pi内，一个entity对于系统负载的贡献仅仅是这个entity在这个周期时间占有的部分。
* 对于最近的load给予最高的权重，但是也允许过去的load以衰减的方式影响当前的load

2. PELT主要数据集合和怎样track load？及其衰减因子？

1. struct cfs\_rq{}
2. struct sched\_entity{}
3. struct sched\_avg{}，重点关注，period\_contrib在1024us之内，某个task在这段时间的占比是多少？
4. PELT算法
   1. 一个task的running time怎么计算的，
   2. load\_sum,util\_sum; load\_avg,util\_avg如何计算
   3. decay\_load如何计算
   4. \_\_contrib如何计算
5. PELT decay factor怎么计算的？

**三、WALT（Windown Assisted Load Tracing）**

1、背景

1. PELT在kernel版本3.8之后才加入的，目的是填充调度器的理解差异；
2. PELT没有方法知道，任何给定的进程有多少load依附在当前CPU；
3. 如果信息可用，load balancing能够最佳的分发进程到各个CPU上去；
4. Android framework在userspace，知道所有app的信息，包括（top/foreground/background apps）。

2、WALT vs PELT

1. 在pixel手机上，相对于EAS r1.1版本的PELT，WALT算法引入WALT，能够更快的响应进程的行为变化；
2. EAS r1.2版本的PELT能够完成与WALT，对进程相类似的响应速度。

3、WALT主要数据结构

1. struct ravg{

u64 mark\_start;

u32 demand;

u32 sum\_histort[];

};

1. struct rq{

u64 curr\_runnable\_sum;//表示当前窗口，已经被执行完的，在 一个cpu上，所有任务的总需求

u64 prev\_runnable\_sum;

}

4、WALT 工作规则

* 每一个任务追踪5个执行窗口，典型每个窗口数值为20ms
* 每个task的需求属性被计算，作为这个task在这个cpu上的cpu的需求
* WALT简单的跟踪一组时间窗口期间进程使用的CPU时间量，仅仅包括运行时间，而不包括在等待io的运行时间
* 对于活跃的task，它的观察窗口跨cpu同步的
* 在最近完成的窗口上运行的所以task的总的load会被考虑安置(意思应该是放在当前cpu上执行)或者进行balance migration
* WALT被合入Android common kernel，而没有合入mainline kernel。

5、WALT：window based load for a task（即每个窗口是一个task的瞬时load）

1. 追踪N执行窗口，典型值为5个窗口，每个窗口时间为20ms
2. Task demand被用来觉得cpu的运行频率
3. 有三种方式来计算task的demand
   1. Max，五个窗口取最大的task demand
   2. Average，取五个窗口task demand的平均值
   3. Recent，最近取值
4. Load的计算考虑了cpu的频率和cpu capacity常量。

6、WALT：window based load for one cpu（即每个窗口是整个cpu 上task总的load）

1. 追踪两个窗口，每个20ms
2. prev\_runnable\_load被用来作为cpu的load
3. load计算考虑了频率和cpu的capacity常量

**四、EAS core**

1、EAS-Energy awareness wake up

1. 在work的task wakeup 路径
2. 只有当所有的cpu都不处于过载情况下，才会被使用
3. 有三个wakeup 路径
   1. 如果不是一个wakup task，则使用传统的方式选择一个idlest group和cpu
   2. 如果是一个wakeup task
      1. 如果所有的cpu没有一个过载，则基于awareness方法选择cpu
      2. 如果有一个cpu过载，则选择临近的idlest cpu

2、Energy aware wakeup path

* energy\_aware\_wake\_cpu找出有足够capacity的一个组和cpu
* energy\_diff 估算利用率变化对总能源的影响
* sched\_group\_energy：估算在特定的调度组里，所有cpu的能耗
* energy\_diff \_evaluta：基于所消耗的能效影响来评估和选择一个cpu。

3、energy difference主要的数据结构

* nrg.before:一个task放在前个cpu上的能耗
* nrg.after：一个task放在目标cpu上的能耗
* nrg.diff：after和before能耗的差值
* nrg.delta：是归一化能耗，范围[0,SCHED\_LOAD\_SCALE]
* cap.before：一个task放在前个cpu上的capacity
* cap.after：。。。。
* cap.delta：。。。

struct energy\_env{

struct{

int before;

int after;

int delta;

int diff;

}nrg;

struct{

int before;

int after;

int delta;

}cap;

}

4、energy difference的计算

1. 找到当前P状态的capacity 索引

For CPU in GROUP

Find max util in GROUP

Find 对于的capacity index1

1. 找到idle状态

For CPU in GROUP

找到处于最浅idle状态的index2

1. 计算GROUP的能耗

For each domain of CPU

For CPUS in Group of CPU

busy\_energy+=group\_util\*cap\_state[index1].power

idle\_energy+=(SCHED\_LOAD\_SCALE-group\_util)\*idle\_state[index2].power

total\_energy+=busy\_energy+idle\_energy

End

end

1. cap\_state[index1].power,idle\_state[index2].power数值在DT里面实现设定好的
2. 计算energy difference

ENERGY\_DIFF = TOTAL\_ENERGY\_AFTER - TOTAL\_ENERGY\_BEFORE

注意可能会为负值？？？

5、估算能效影响

1. 分四个象限

* O 最优化区域
* S 次优化区域
* PB 性能boost区域
* PC 性能限制区域

1. O

* nrg\_delta<0 && cap\_delta>0

1. S

* nrg\_delta>0 && cap\_delta<0

1. PB

* nrg\_delta>=0 && cap\_delta>=0
* cap\_delta/nrg\_delta > cap\_gain/nrg\_gain

1. PC

* nrg\_delra< 0 && cap\_delta < 0
* cap\_delta/nrg\_delta<cap\_gain/nrg\_gain

6、估算能效影响--设计思路

* boost数值被用来计算PELT load的余量
  + 余量与原始备用capacity的补充成正比
  + MARGIN=BOOST\*(SCHED\_LOAD\_SCALE-CURRENT\_LOAD)
* 决策者
  + 对于更低的boost，为了节省能耗，能够接受巨大性能的改变，不管性能是增加还是减少
  + 对于更大的boost，为了增加性能，能够接受大的能耗改变，不管能耗增加还是减少

**五、load balance**

1、big.LITTLE MP load balance overview

* Task 负载均衡被考虑，为了BL cores

1. 在task创建时间（Frok migration）
2. 在task wakeup时间（wakeup migration）
3. 在task运行期间（force migration）
4. 在cpu idle时间（idle-cpu migration）
5. 在周期性的load balance（offload migration）

* task负载均衡是在每个域级别中，每个GROUP的自下而上的负载平衡

2、load balance cases

* 当一个cpu是idle

1. 把task从最繁忙的cpu上迁移到idle cpu
2. 当一个cpu变成了新的idle状态，调用idle\_balance
3. 当有一个cpu处于idle状态，调用nohz\_idle\_balance，被触发在每个scheduler tick里面

* 当load balance间隔达到

1. 把task从最繁忙的cpu上迁移到更轻loading的CPU上
2. 被触发在每个scheduler tick，应该是周期性的
3. 调用load\_balance

* 当fork，execute or wakeup a task

1. 选择最轻loaded的cpu
2. 三个不用的场景通过一个对应的domain flag被标记

* SD\_BALANCE\_FORK：新创建的task，第一次被wakeup
* SD\_BALANCE\_EXEC：新task执行
* SD\_BALANCE\_WAKE：正在waking up的task

3、capacity awareness

1. 当选择一个idle/busy cpu/group的时候，考虑当前cpu的capacity
2. 一个cpu的最大capacity动态update

* 当一个cpu的频率被限制的时候，它的capacity相应的按照比例更新(thermal)
* 一个cpu的capacity被减少，是由于RT/DL tasks在同一个run queue里

1. 频率限制的公式

* scale\_cpu\_capacity=(max\_capacity\*freq\_scale)/1024

4、fork migration：challenge vs solution

5、load balance-----periodic/idle

1. 每个balance间隔之后load balance

* Load balance周期性触发
* Imbalance 存在
* 在对繁忙的cpu上，超过一个可以运行的task

1. 如果只有一个可以运行的task在最繁忙的CPU上，则有时候需要强制触发load balance

1. 四组imbalance types
2. 测量所以cpu和group的load

* 测量所有cpu和group的load
* 测量group的imbalance 类型

1. 找到最繁忙的cpu

* 挑选最繁忙的group并计算起load
* 挑选最繁忙的run queue

1. 迁移一个task从最繁忙的cpu到idle cpu上

* 从最繁忙的cpu上解除task
* 将task绑定到一个idle cpu上

6、load balance--Misfit task

* 一个misfit task必须满足下面三个条件

1. 这个cpu的capacity不是系统中最大的cpu capacity
2. 这个cpu的capacity不超过系统最大capacity的80%
3. Task的task\_util超过这个cpu capacity的80%

* 调度器什么时候检测misfit task

1. 当挑选一个task来调度时，检测是否有misfit task
2. 当更新group的负载均衡状态时，检测一个group是否有misfit task

* 如果在一个group内有misfit task，则强制做迁移action