UNIONTECH

UNIONTECH

打造操作系统创新生态

并发高负载场景下的负载均衡优化

周鹏：zhoupeng@uniontech.com

日期：2021年10月24日

UNIONTECH

CONTENTS

01. 背景介绍

1)现象

因为业务需要跑多个的cpu密集型进程，为什么这些高负载进程容易被迁移到不同cpu上运行，而不是固定在自己的cpu运行？而且跑的并发进程越多，这种迁移越明显？

2)举例

服务器上并发运行高负载进程，需要性能最优。

服务器配置为16个node, 128个cpu,内存128G.

并发运行128个进程，每个进程占用内存800M左右。

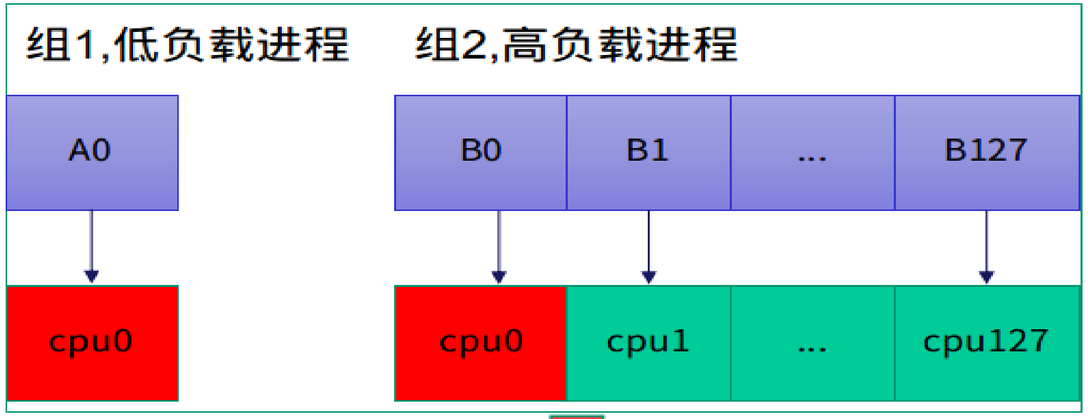
期望结果：

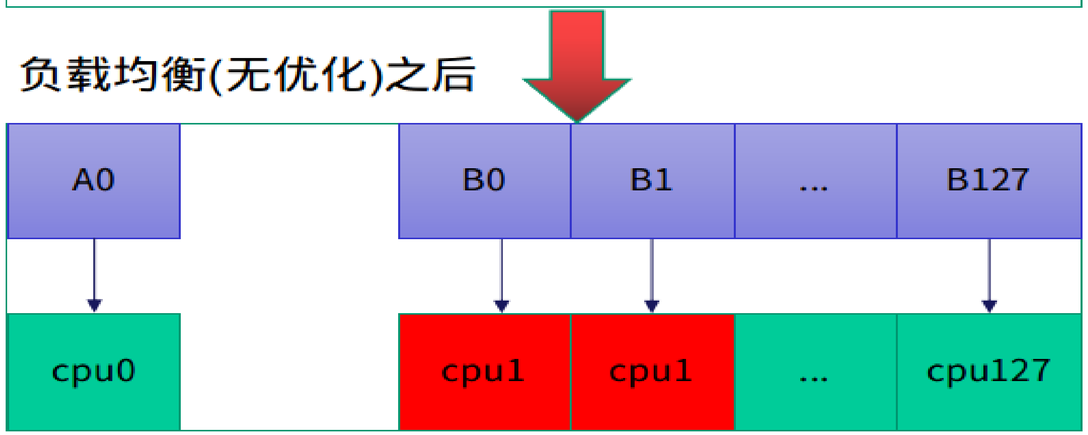
每个进程都在自己的cpu上运行，高负载进程没有迁移，性能最优。

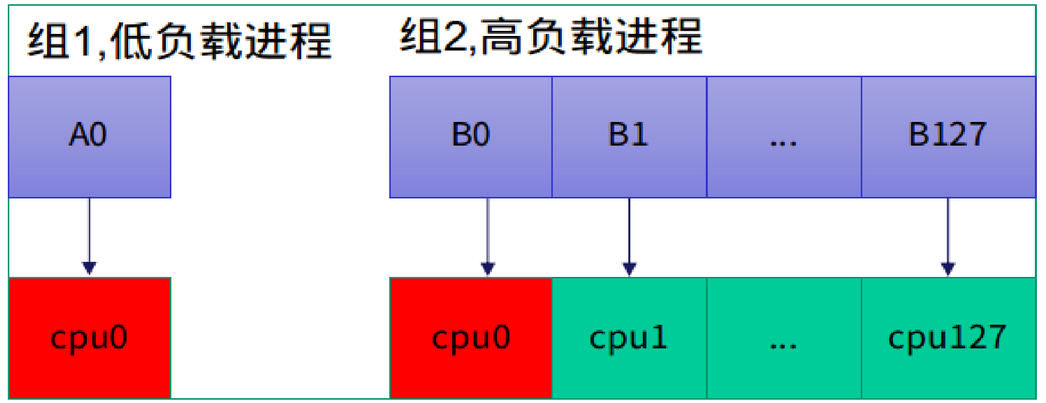
实际结果：

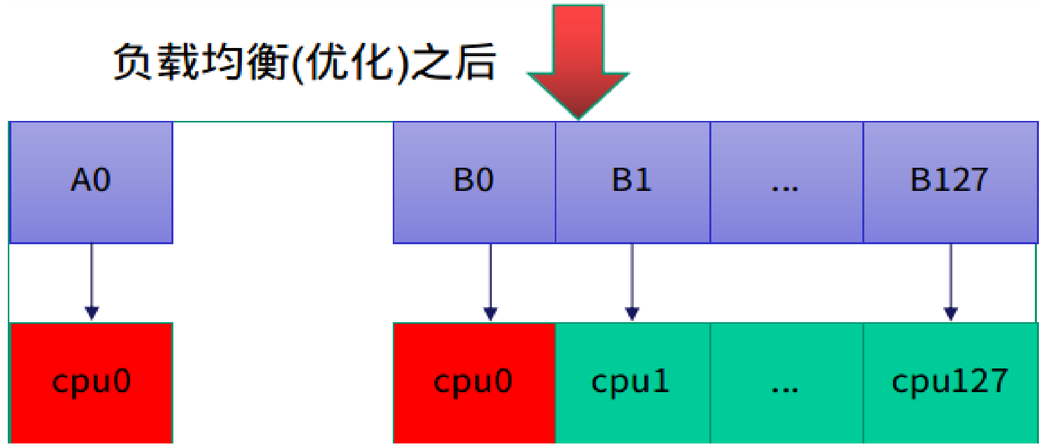
运行过程中发现高负载进程被频繁迁移，性能下降明显

2. 当前问题









左图：无优化时表现，A0低负载进程需要在cpu0上运行，在cpu0上运行的B0高负载进程被迁移到cpu1

右图：A0低负载进程要在cpu0上运行时不会将B0高负载进程迁移到cpu1，每个高负载进程都在自己的cpu上运行

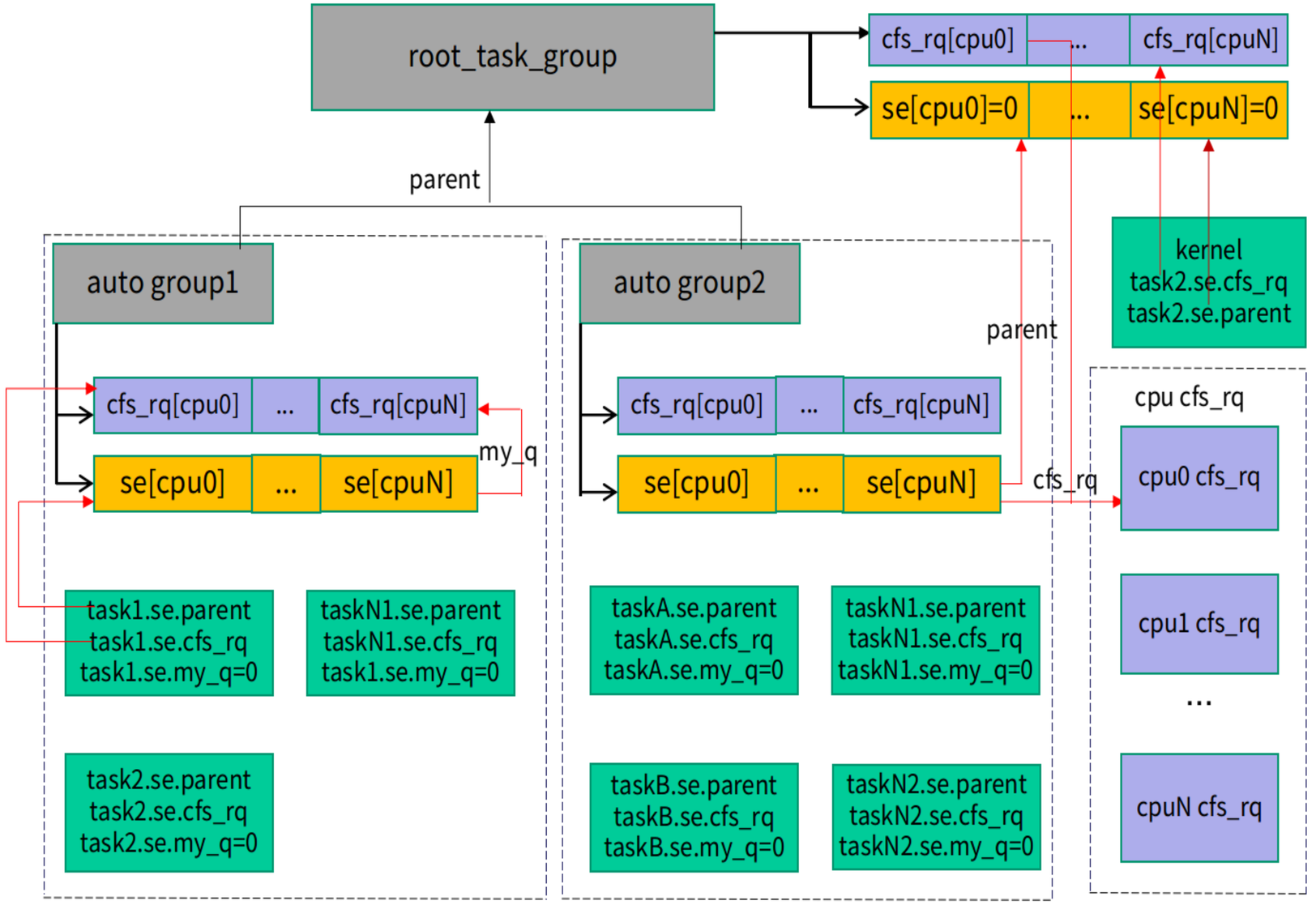
3. taskgroup相关术语介绍

task se	**任务调度实体**，最小运行实体单元，task sched entity ， 简称任务se; 一个进程或一个线程是一个任务 se
taskgroup se	**任务组调度实体**，一个虚拟的se, 它上面记录的负载是包含了它拥有的任务se的负载以某种加权方式计算出来的负载之和。
taskgroup cfs_rq	**任务组运行队列**，一个虚拟的运行队列，有了它任务se被发送到此运行队列，而不是直接发送到cpu运行队列；
cpu cfs_rq	**cpu运行队列**，真实的运行队列.
task se load_avg	**任务se期望负载**，在运行队列中就会统计其负载，存在时间片到期或者中断运行中断程序，所以不一定总是被运行；
task se runnable_load_avg	同task se load_avg
task se util_avg	**实际运行负载**；只有被cpu真正运行了，才计算这个值，top统计进程的cpu占用率就是用它。
group se load_avg	任务**se->avg.load_avg**负载加权之和,不是真实负载； 该组se下面的任务se的load_avg以加权方式计算出来的负载。
group se runnable_load_avg	任务**se->avg.runnable_load_avg**负载加权之和

3. taskgroup相关术语介绍

group se util_avg	**组的实际运行负载；** 只要有任务se在运行，那么就有负载，只有一个满负载se时是1024, 同时拥有3个满负载se时也是1024, 因为这三个任务se不会同时运行在一个cpu上。
cfs_rq load_avg	**任务se的load_avg之和.**
cfs_rq runnable_load_avg	**任务se的runnable_load_avg之和。**
cfs_rq util_avg	**运行队列的实际运行负载；**
h_load	**加权负载**，cpu 运行队列的load负载、runnable负载；组se的load负载、runnable负载；任务se的task_h_load都是加权负载。 **cpu运行队列的权值为多个组se的权值之和。** **任务se的权值为它所属的组se权值的比例分配。**
imbalance	调度组之间的负载不均衡值，大于0表示出现了不均衡，触发负载均衡

3.1 task group之autogroup架构



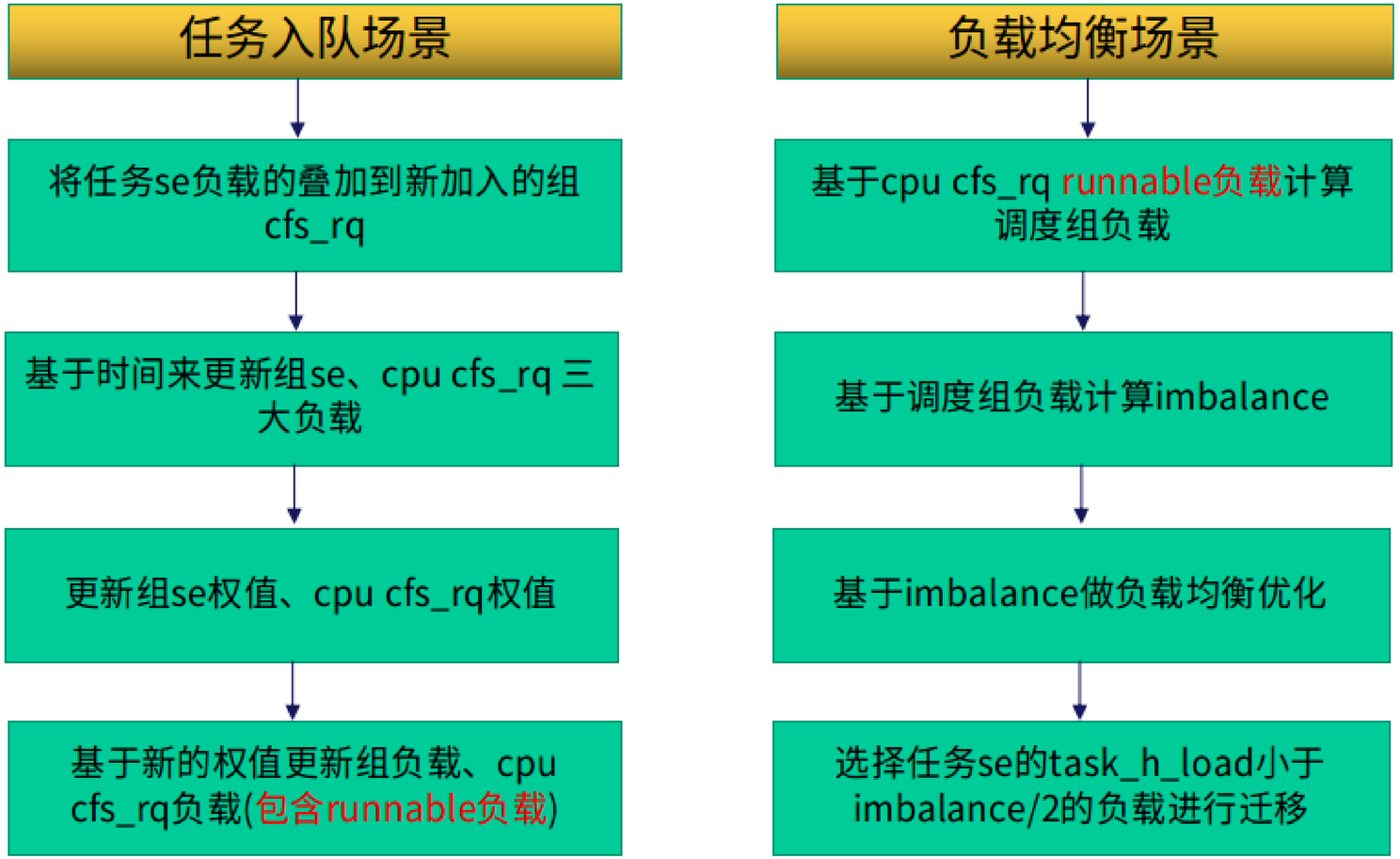
1)root_task_group为所有组的根组，简称rtg

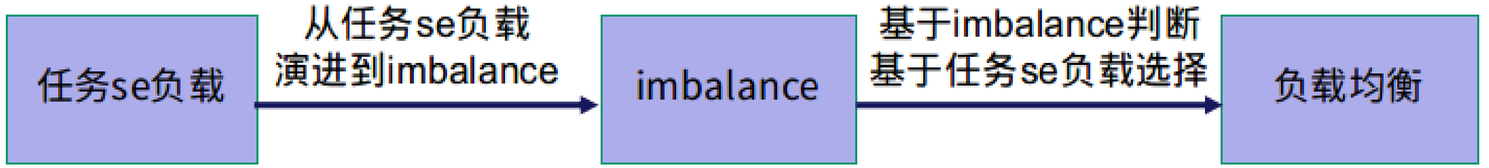
2)ksys_setsid的调用触发autogroup的创建,创建者自己独立为autogroup, parent指向rtg

3)bash启动时调用ksys_setsid,每个bash都是一个autogroup

4)假设cpu个数为N，每一个autogroup都有N个cfs_rq, N个se, 和cpu的个数一致是为了做到autogroup中的任务在任一个cpu中运行时都能统计在此cpu上运行的各种负载以及最终该cpu运行队列的负载

3.2 autogroup下的各层级负载更新





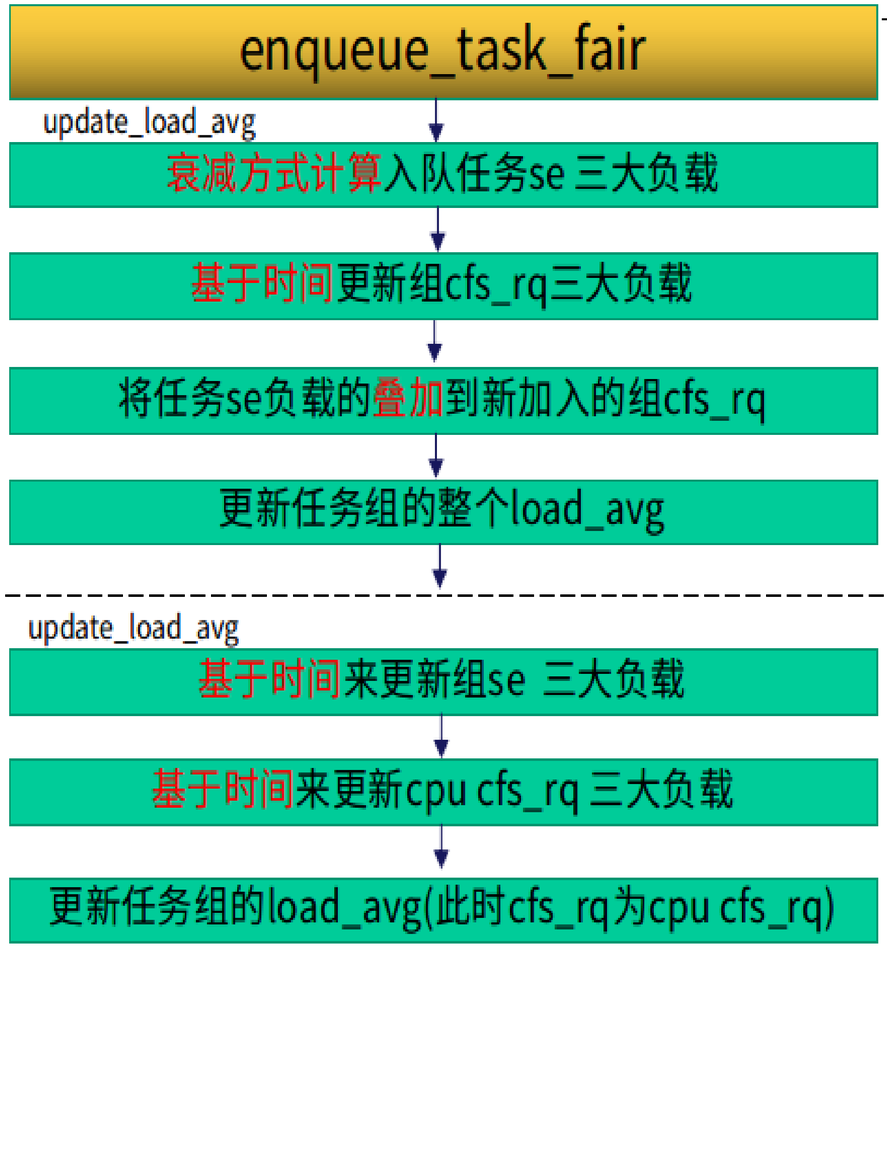
enqueue_task_fair只是计算负载的其中一个场景

1)整个负载流程分析基于autogroup两层架构分析，这也是大部分进程默认的运行架构

2)分析了任务入队这个场景，还有出队场景、任务切换场景都会涉及各层级负载的更新。

3)整个流程的目标主要讲解从任务se负载更新开始，到计算出调度组之间的imbalance值，然后基于imbalance决策是否需要进行任务迁移

3.3 autogroup下的各层级负载更新细节



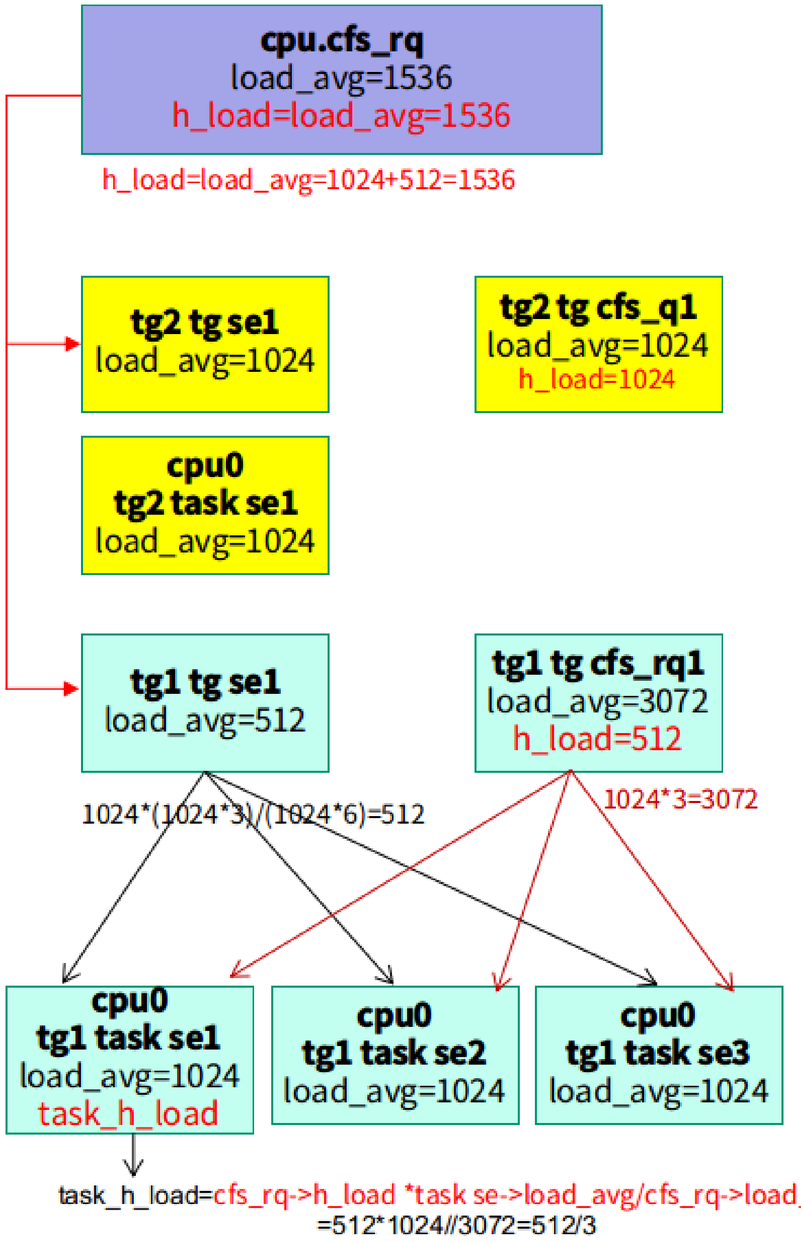
enqueue_task_fair只是计算负载的其中一个场景

1)update_load_avg第一次调用
负载计算从底往上进行，从任务se开始计算。入队场景下，新的任务se之前没有在此cpu上运行，负载直接衰减, 然后叠加负载

2)update_load_avg第二次调用
对于目标cpu, 组se肯定有某个任务在运行，因此其负载基于时间来计算。
基于时间更新负载

3)update_cfs_group
新加入任务到组se了，需要更新组se权值及负载

3.4 各层级负载计算公式



1)组cfs rq->load_avg
tg cfs_rq->load_avg=sum(task se->load_avg)
=1024*3=3072

2)组se load_avg
tg se->load_avg= load * cfs_rq->load_avg/sum(tg cfs_rq->load_avg)
= 1024 * (1024*3)/(1024*6)=512

3)组cfs rq->h_load
tg cfs_rq->h_load = tg se->load_avg=512

4)cpu cfs_rq->load_avg
cpu cfs_rq->load_avg = sum(tg se->load_avg)
= tg1 se->load_avg + tg2 se->load_avg=512+1024=1536

5)cpu cfs_rq->h_load
cpu cfs_rq->h_load = cpu cfs_rq->load_avg=1536

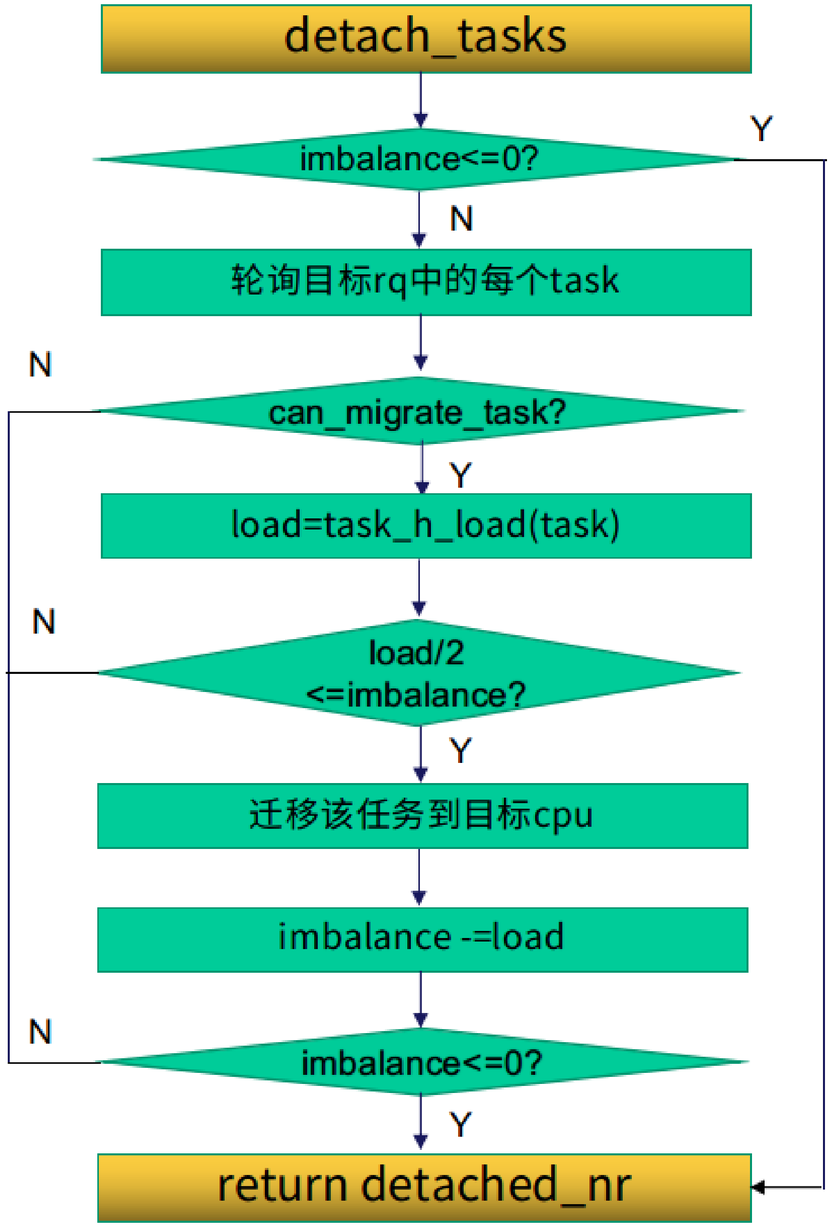
6)任务se->task_h_load
task se->task_h_load=cfs_rq->h_load * task se->load_avg / cfs_rq->load_avg
=512 * 1024/3072=512/3

3.5 imbalance的计算

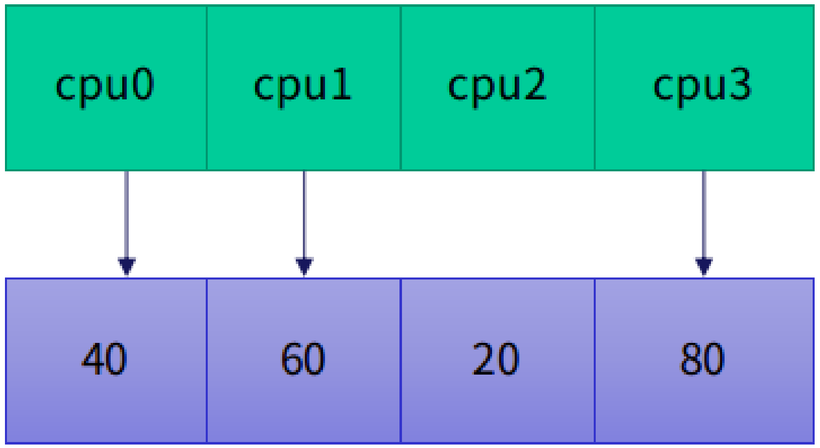
avg_load表示调度组之间的平均负载；
busiest是负载最高的调度组；
local表示当前正在遍历的调度组；
imbalance=min(abs(avg_load - local->avg_load), abs(busiest->avg_laod - avg_load)

假设系统有cpu0~cpu3四个核，每个核是一个调度组，四个调度组形成一个调度域。
四个调度组的avg_load分别为40,60,20,80, 遍历到调度组cpu0时
local=40 avg_load=(40+60+20+80)/4=50 busiest=80
imbalance=min(50-40， 80-50)=10 > 0, 出现了不均衡, 需要做负载均衡

3.6 负载均衡流程


detach_tasks
imbalance<=0?
Y
N
轮询目标rq中的每个task
N
can_migrate_task?
Y
load=task_h_load(task)
N
load/2
<=imbalance?
Y
迁移该任务到目标cpu
imbalance -=load
N
imbalance<=0?
Y
return detached_nr

3.7 负载均衡导致任务迁移举例



平均负载=(40+60+20+80)/4=50

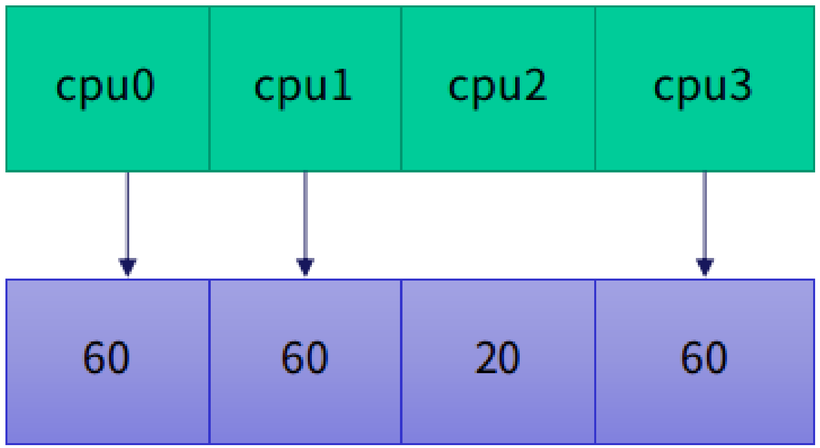
最忙负载=80

local负载=40

imbalance=min(50-40, 80-50)= 10

迁移判断：
cpu0--cpu3之间负载均衡，假设 cpu3上两个任务p30=20 p31=60
20/2 <=imbalance, 将p30迁移到cpu0

负载均衡之后

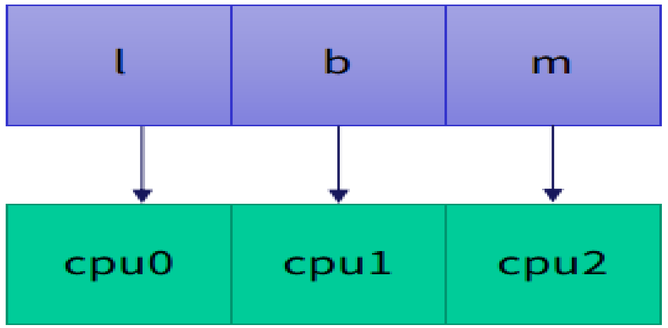


均衡之前：最大负载差为80-20=60
方差为：((40-50)^2+(60-50)^2
+(20-50)^2 +(80-50)^2)/4=2000/4=500

均衡之后：最大负载差为60-20=40
方差为：((60-50)^2 + (60-50)^2
+ (20-50)^2 + (60-50)^2)/4=300

负载方差从500减少到了300

3.8 负载均衡方差减少证明



为了方便证明，假设只有三个调度组，每个调度组有一个cpu, 在这三个调度组之间做负载均衡

如右图所示 l为cpu0负载， b为cpu1负载， m为cpu2负载

local =l　busiest=m　u = (l+b+m)/3

已知:l < b < m

imbalance (简称im)= min(u - local, m -u)=(m+b-2l)/3

此时能迁移过来的最大负载为：load=2*(m+b-2l)/3

s1=((l -u)^2 + (b-u)^2 + (m-u)^2)/3

= ((l-m)^2 + (l-m)^2 + (m-b)^2)/9

3.9 负载均衡方差减少证明

迁移之后：

l_1=(l + load) b_1 = b m_1 = (m - load)

s2 =((l_1 - u)^2 + (b_1 - u)^2 + (m_1 - u)^2)/3

=(14b^2 + 14l^2 - 2m^2 - 26bl - 2mb - 2lm)/

s2 -s1 = [(b-l)^2 - (m-b)^2 + (2b-l-m)(b-l)]*4/2

红色部分相减小于0, 绿色部分相减也小于0,所以s2比s1小，方差在减小

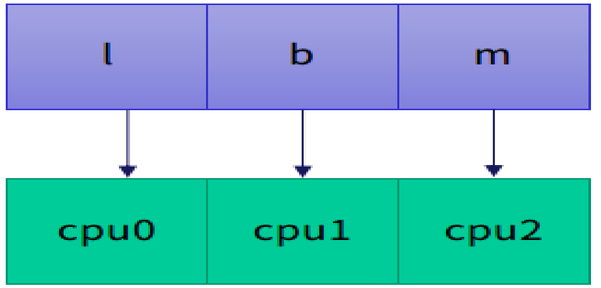
m-b > b-l > 0 得到(b-l)^2 - (m-b)^2 < 0

b > l l+m-2b>0 得到 (2b-l-m)(b-l) < 0

所以s2 - s1 < 0

所以如果按照im = min(u - local, m-u)这个不均衡条件进行任务迁移，会让负载方差越来越小，调度组之间的负载差异越来越小了。

3.10 负载均衡方差减少反证



假设从im中取最大的那个值

即 im=max(u-local, m-u)=max((m+b-2l)/3, (2m-l-b)/3)=(2m-l-b)/

此时能迁移过来的最大负载为：load=2*(2m-l-b)/3

已知u=(l+b+m)/3

迁移之前负载方差为s1

s1=((l -u)^2 + (b-u)^2 + (m-u)^2)/3

= ((l-m)^2 + (l-m)^2 + (m-b)^2)/9

3.11 负载均衡方差减少反证

迁移之后：

l_1=(l + load) b_1 = b m_1 = (m - load)

s2 =((l_1 - u)^2 + (b_1 - u)^2 + (m_1 - u)^2)/3

=(14m^2 + 14b^2+2l^2 - 26mb - 2lm - 2lb)/27

s2 -s1 = (8m^2 + 8b^2 - 4l^2 - 20mb + 4lm + 4b

= [(m-b)^2 - (l-b)^2 + (m-b)(l+m-2b)]*4/2

红色部分相减大于0, 绿色部分相减也大于0,所以s2比s1大，方差在增加。

m-b > b-l > 0 得到(m-b)^2 - (l-b)^2 > 0

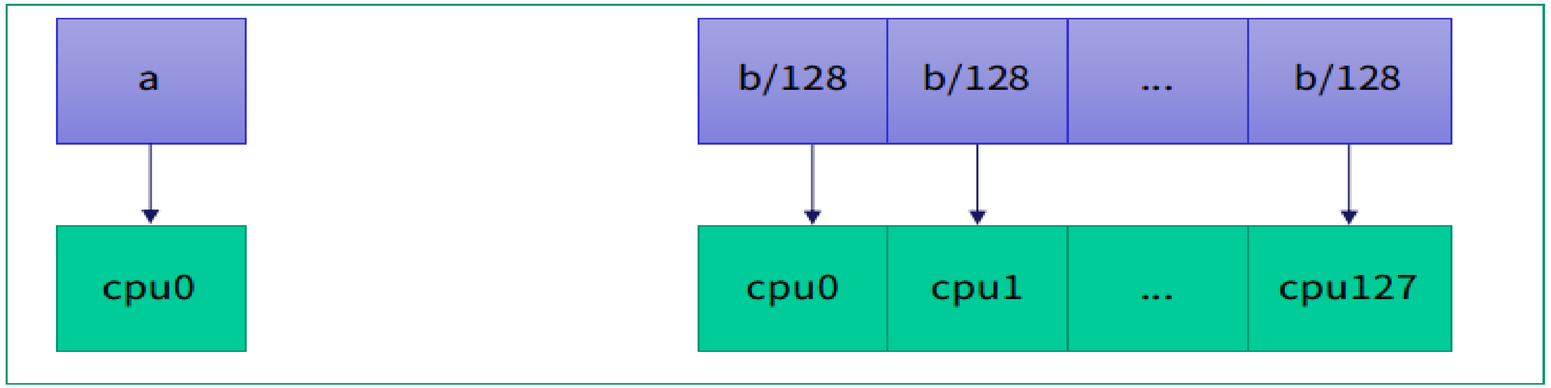
m-b > 0 l+m-2b>0 得到 (m-b)(l+m-2b) > 0

所以s2 - s1 > 0

如果按照im = max(u - local, m-u)做负载均衡，导致更大的不均衡。

因此按照im=min(u-local, m-u)这个不均衡条件做负载均衡是正确的负载均衡

3.12 基于加权负载导致任务迁移理论分析



u(cpu平均负载)：((a+b/128) + b/128 + b/128 ...+ b/128)/128=b/128 + a/128

busiest: a+b/128

假设cpu1在做负载均衡，此时cpu1为local cpu，local = b/128

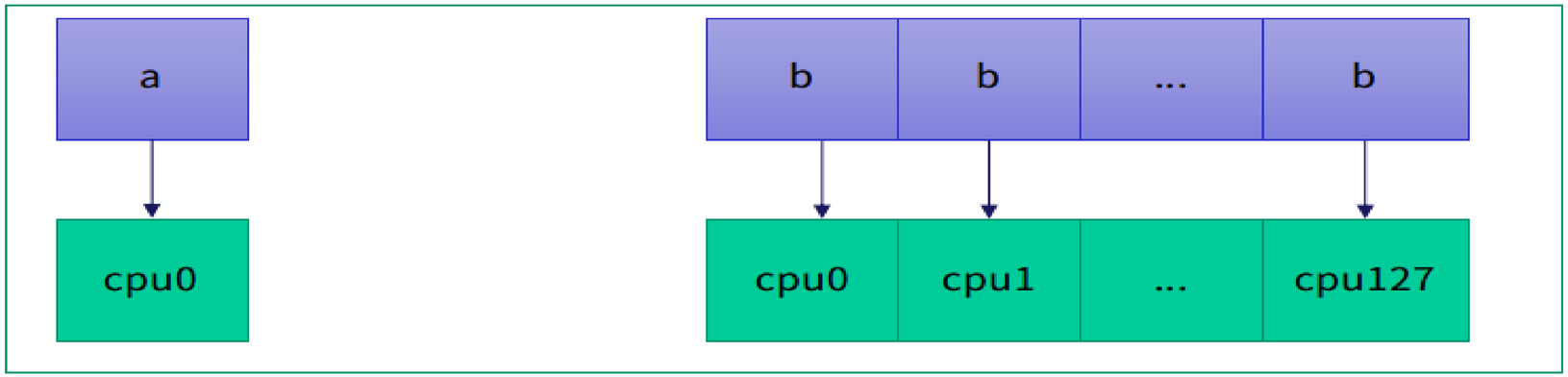
imbalance = min(a+b/128 - b/128 - a/128, a/128+b/128 - b/128)=a/128

对于任务a
a/2 < a/128 不成立，不可能迁移

对于任务b
b/128/2 < a/128, 可能成立，存在迁移可能

cpu0上高负载任务b被迁移到其它cpu上运行了，除了出现远端内存访问问题，还会导致多个高负载任务集中到一个cpu上运行，两个进程同时受影响

3.13 基于实际负载不会导致任务迁移理论分析



u(cpu平均负载)：((a+b) + b + b ...+ b)/128=b + a/128

busiest: a+b

假设cpu1在做负载均衡，此时cpu1为local cpu，local = b

imbalance = min(a+b - b - a/128, a/128+b - b) = a/128

对于任务a
a/2 < a/128 不成立，不可能迁移

对于任务b
b/2 < a/128, 因为b的负载比a高，不可能成立，不可能迁移

基于实际负载判断不均衡状态时，
不会出现跑一个低负载任务导致高负载进程被迫迁移

3.14 总结

1)基于imbalance不均衡值选择迁移的任务会让调度域（单node情况下为cpu）之间的负载方差越来越小，那么taskgroup情况下它真的做到了负载方差越来越小了吗？

回答：如果是加权任务，那么它就做到了从加权维度做到了负载方差越来越小；但是从真实负载维度来看就不是了，很有可能一个cpu在跑一个低负载任务，而另外一个cpu在跑两个实际负载很高的任务，因为此时只是前者的加权负载比后两者的加权负载还要大而已。这种迁移策略只要是为了保证组与组之间的“公平”，好处是交互式场景下，交互式任务获取到的运行时间片更多。

2)cpu密集型进程，为什么这些高负载进程容易被迁移到不同cpu上运行，而不是固定在自己的cpu运行？

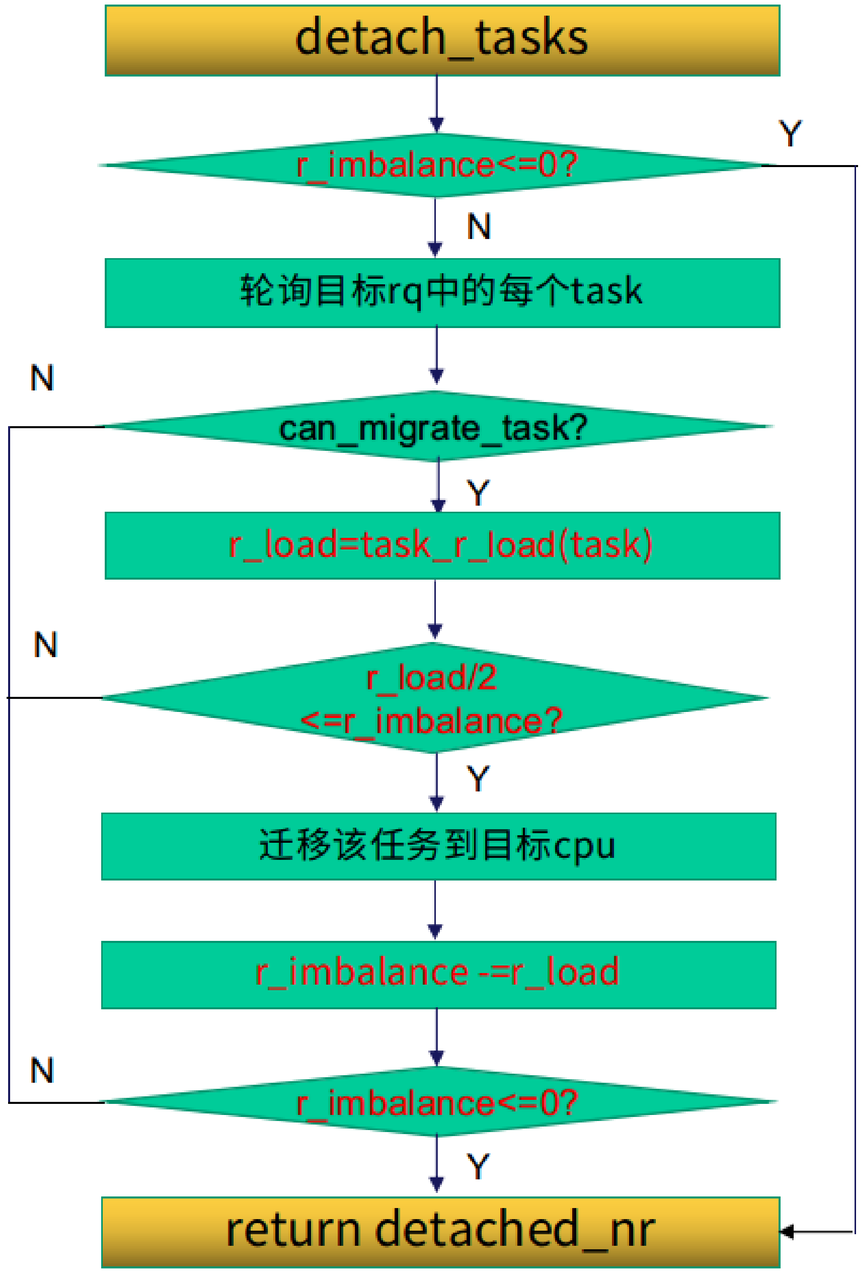
回答：因为任务的迁移基于组内任务的加权负载，不是真实负载，组内任务过多时，导致每个任务的加权负载很小，很容易比其它组任务的低负载任务的加权负载小，当这些低负载任务运行时，真实的高负载任务被迁移了。

3)基于真实负载计算imbalance，基于真实负载选择迁移任务能保证调度域之间的方差越来越小吗？

回答：能做到，此时的方差越来越小就是指真实负载的方差越来越小了。

基于真实负载的迁移策略更适合追求高负载性能场景，它避免了后台干扰导致的任务迁移，cpu资源访问受影响，内存缓存因为亲和行也会受影响

4.优化方案


detach_tasks
r_imbalance<=0?
Y
N
轮询目标rq中的每个task
N
can_migrate_task?
Y
r_load=task_r_load(task)
N
r_load/2
<=r_imbalance?
Y
迁移该任务到目标cpu
r_imbalance -=r_load
N
r_imbalance<=0?
Y
return detached_nr

5.1 整型计算优化验证

在ARM 桌面4核系统上进行验证：

测试程序并发4线程满负荷进行整型运算，同时系统存在2个20%CPU占有率的后台进程干扰

测试结果如下：

	测试程序 总迁移次数	测试程序运行性能 (每秒平均运算次数)
优化前	800	82370429
优化后	0	89141091

测试结论：

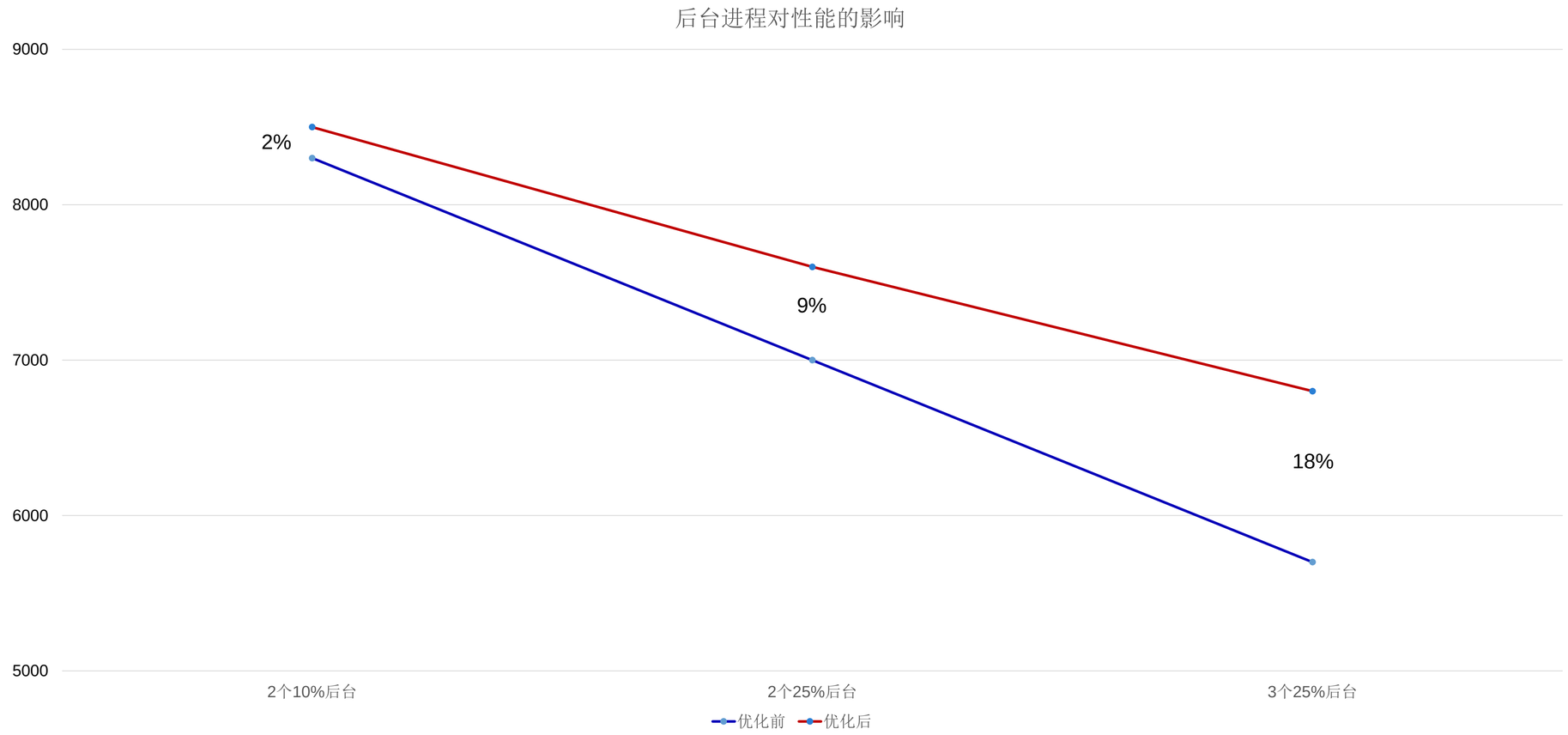
由于4个测试线程属于同一个组，满负荷运行时加权负载最多只有1024/4，因此容易受到20%占有率的后台进程影响而发生CPU迁移，导致两个测试进程竞争CPU，性能下降。

而优化后，由于满负荷线程负载为1024,不会发生迁移，性能明显提升。

5.2 浮点型计算优化验证

在ARM 桌面4核系统上进行验证：

测试程序并发4线程满负荷进行浮点运算，同时系统存在部分不同占有率的后台进程干扰，测试结果如下：



5.3 内存型优化验证

在某ARM 服务器128核系统上进行验证：

测试程序并发128线程满负荷进行指定大小数据进行整型和浮点型运算，观察迁移对内存访问速度的影响

测试结果如下：

	测试程序运行时间	进程迁移次数
优化前	10min	3879
优化后	6min	0

测试结论：

由于128个测试线程属于同一个组，满负荷运行时分层负载一般只有1024/128，因此容易受到系统的后台进程影响而发生CPU迁移。

由于服务器是多node架构，当进程迁移到远node时，如果内存(如缓存内存)没有得到迁移，此时性能恶化较明显。如果基于真实负载控制高负载进程的迁移，迁移次数下降到0, 性能提升明显。

THANKS

400-858-8488

统信软件技术有限公司

UnionTech Software Technology Co., Ltd.