



OpenAnolis

龙蜥内核的 Load Averages 剖析

2021-11-17 庞训磊



Load Averages 初窥

```
top - 10:20:02 up 175 days, 15:39, 4 users, load average: 1.00, 1.02, 1.00
```

```
$ uptime
```

```
10:20:02 up 175 days, 15:39, 4 users, load average: 1.01, 1.02, 1.00
```

```
$ cat /proc/loadavg
```

```
1.01 1.02 1.00 1/799 40523
```

/proc/loadavg接口实现:

fs/proc/loadavg.c: loadavg_proc_show()

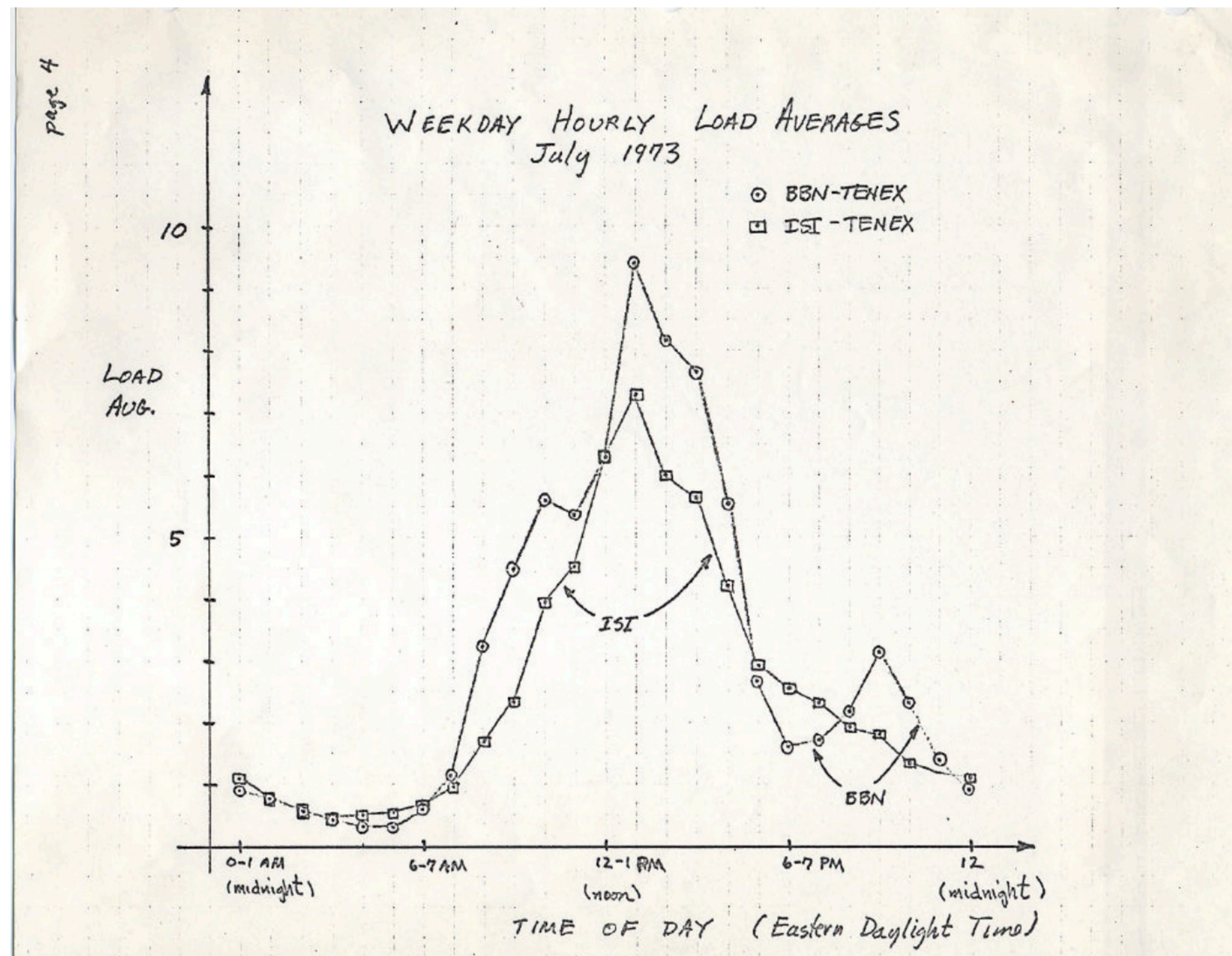
本文基于龙蜥4.19内核代码

git@codeup.openanolis.cn:codeup/storage/cloud-kernel.git devel-4.19



Load Averages 历史

源于一个古老的(1973年) RFC: <https://datatracker.ietf.org/doc/html/rfc546>



The TENEX load average is a measure of CPU demand. The load average is an average of **the number of runnable processes** over a given time period. For example, an hourly load average of 10 would mean that (for a single CPU system) at any time during that hour one could expect to see 1 process running and 9 others ready to run (i.e., not blocked for I/O) waiting for the CPU.



Load Averages 定义

由上可知，Load Averages在古老的TENEX操作系统上就已经作为一个衡量系统负载的监控指标使用了，它对系统中的runnable任务个数采用1min/5min/15min的指数衰减算法进行数据平滑处理(这也就是Averages的意思)。

具体的指数衰减系数为：

1分钟衰减系数e: 0.920043902

5分钟衰减系数e: 0.983471344

15分钟衰减系数e: 0.994459811

计算规则为： $\text{load} = \text{load} * e + \text{nr_runnable} * (1 - e)$

注：下文将“Load Averages”简称为“Load”



Load 1min/5min/15min 背后的数学原理

Linux会每隔一定时间在时钟中断中进行周期采样(5s一次)当前有多少任务，然后按照公式对Load 1/5/15分别进行计算。

假设第n次采样时的load负载为 L_n ，采样时刻n的Runnable任务总个数为 T_n ，我们可以得到每一次采样周期过后的Load值的计算公式为：

$$\begin{aligned} L_1 &= L_0e + T_1(1-e) = T_1(1-e) \quad /* L_0初始值为0*/ \\ L_2 &= L_1e + T_2(1-e) = T_1(1-e)e + T_2(1-e) \\ L_3 &= L_2e + T_3(1-e) = T_1(1-e)e^2 + T_2(1-e)e + T_3(1-e) \\ &\dots \dots \\ L_n &= T_1(1-e)e^{(n-1)} + T_2(1-e)e^{(n-2)} + \dots + T_n(1-e) \end{aligned}$$

假设 T_n 任务数恒定为T时，套用上面的公式便可得到一个e等比数列：

$$L_n = T(1-e) * (e^{(n-1)} + \dots + e^0) = T(1-e)(1-e^n)/(1-e) = T(1-e^n)$$



Load 1min/5min/15min 背后的数学原理

- Load 1min (1分钟采样 $60/5=12$ 次，15分钟采样 $15*60/5=180$ 次)

$$L_{12} = T(1 - 0.920043902^{12}) = T * 0.632123$$

$$L_{180} = T(1 - 0.920043902^{180}) = T * 1 \quad /*15分钟后上升至真实负载值*/$$

- Load 5min (5分钟采样 $5*60/5=60$ 次)

$$L_{60} = T(1 - 0.983471344^{60}) = T * 0.632123$$

- Load 15min (15分钟采样 $15*60/5=180$ 次)

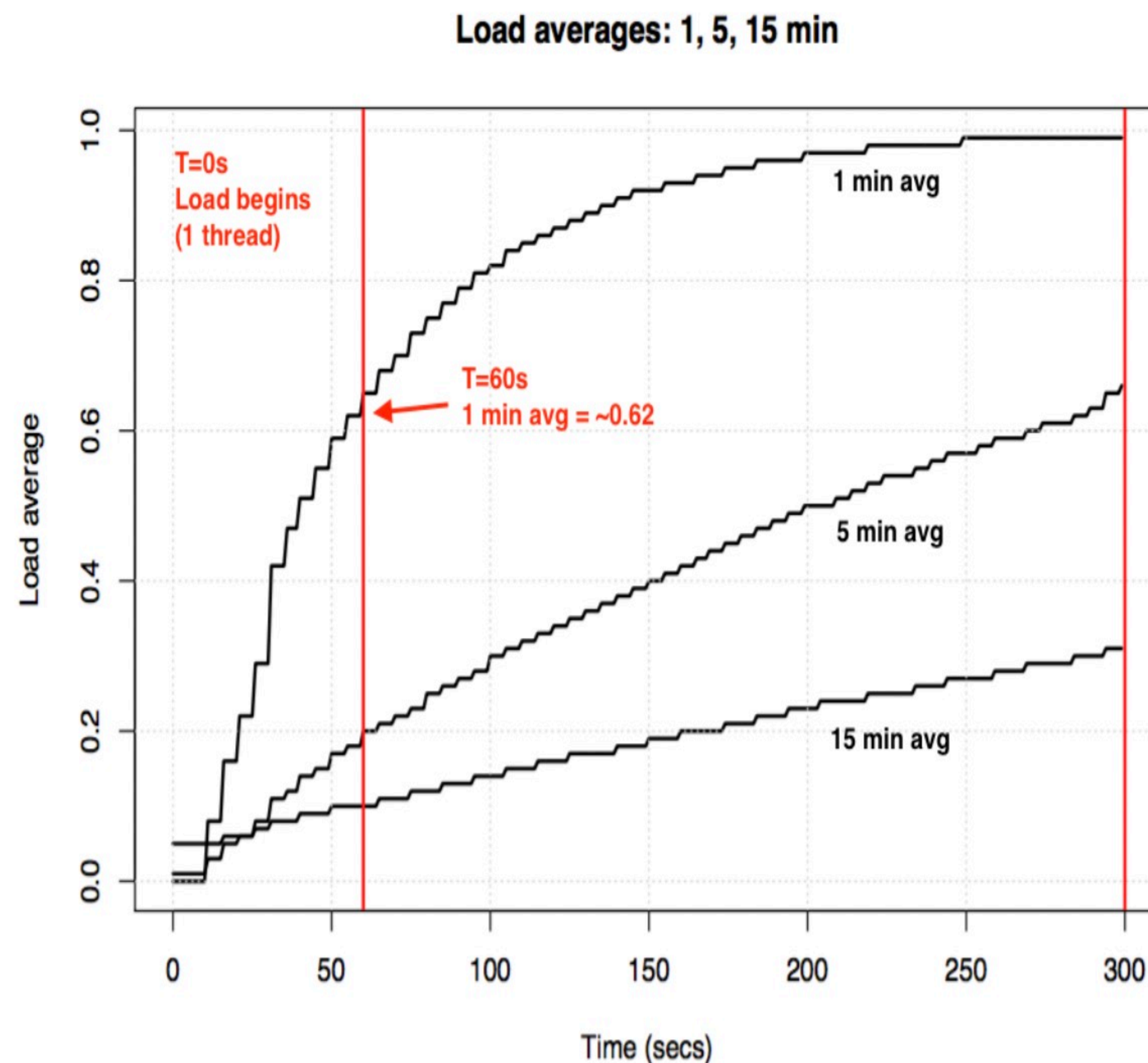
$$L_{180} = T(1 - e^{180}) = T(1 - 0.994459811^{180}) = T * 0.632123$$

由上可知，Load 1/5/15不同e值的选取是为了保证相应的时间窗得到相同的结果，并且15分钟后Load1可以打满到真实情况。

同理，系统彻底空闲后Load下降情况也类似，15分钟后Load1从T降到0。不过Linux下的实际负计算是有损的(用的是放大后的整型数进行的计算)，实际比这个理论上的变化趋势要快不少。



Load 1min/5min/15min 变化曲线示例



Load1 -> Load5 -> Load15

- 响应越来越迟钝；Load1最常用。

Load1 > Load5 > Load15

- 表示系统负载在增加。

Load1 < Load5 < Load15

- 表示系统负载在减少。

- Load 1/5/15需要配合CPU个数进行分析，只有当值大于CPU个数时，我们才能说系统很有可能过载了。



Linux内核的Load语义扩展

Linux不只统计Runnable任务，还加入了Uninterruptible的任务。

From: Matthias Urlichs <urlichs@smurf.sub.org>

Subject: Load average broken ?

Date: Fri, 29 Oct 1993 11:37:23 +0200

The kernel only counts "runnable" processes when computing the load average. I don't like that; the problem is that processes which are swapping or waiting on "fast", i.e. noninterruptible, I/O, also consume resources.

It seems somewhat nonintuitive that the load average goes down when you replace your fast swap disk with a slow swap disk...

Anyway, the following patch seems to make the load average much more consistent WRT the subjective speed of the system. And, most important, the load is still zero when nobody is doing anything. ;-)

--- kernel/sched.c.orig Fri Oct 29 10:31:11 1993

+++ kernel/sched.c Fri Oct 29 10:32:51 1993

@@ -414,7 +414,9 @@

unsigned long nr = 0;

for(p = &LAST_TASK; p > &FIRST_TASK; --p)

- if (*p && (*p)->state == TASK_RUNNING)

+ if (*p && ((*p)->state == TASK_RUNNING) || (*p)->state == TASK_UNINTERRUPTIBLE) ||

In linux, “Load” reflects the load of the whole system other than that of CPU.



Linux内核的Load算法的挑战

kernel/sched/loadavg.c

```
1 // SPDX-License-Identifier: GPL-2.0
2 /*
3  * kernel/sched/loadavg.c
4  *
5  * This file contains the magic bits required to compute the global loadavg
6  * figure. Its a silly number but people think its important. We go through
7  * great pains to make it work on big machines and tickless kernels.
8  */
9 #include "sched.h"
```

主要挑战：

- 多核系统下，近乎零干扰的Load算法实现
- 应对tickless(NO_HZ)场景
- 实现per-cgroup Load (龙蜥内核)



Linux内核的Load衰减系数

```
include/linux/sched/loadavg.h:
```

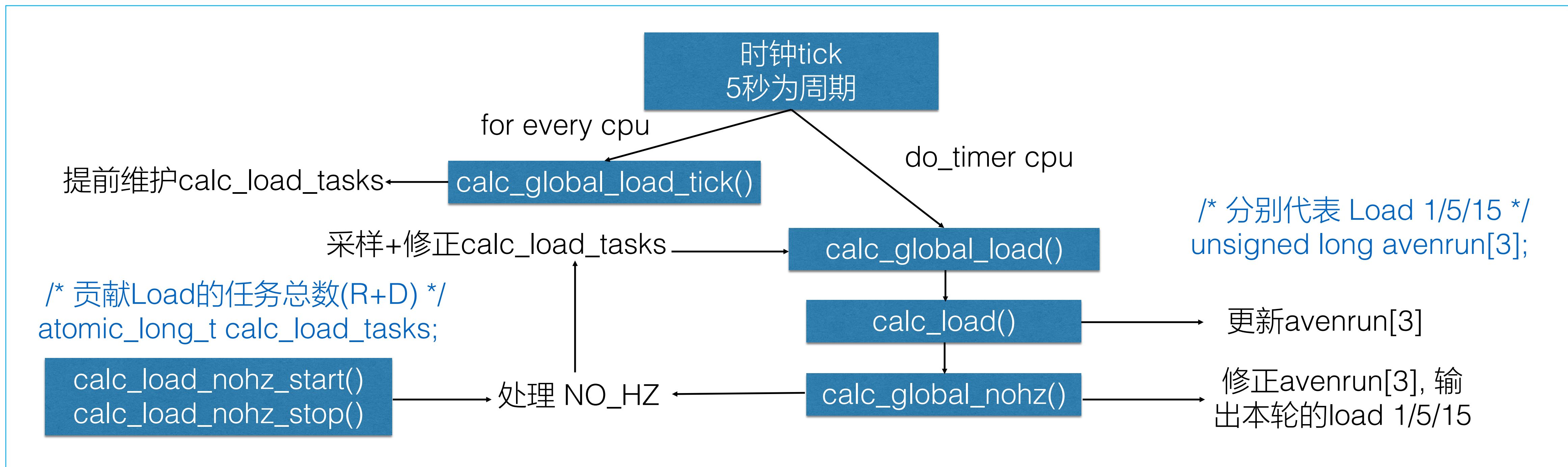
```
#define FSHIFT          11
#define FIXED_1         (1<<FSHIFT)
#define LOAD_FREQ       (5*HZ+1)
#define EXP_1           1884
#define EXP_5           2014
#define EXP_15          2037
```

因为浮点数e不好在内核态计算，Linux给它放大了一下即乘以FIXED_1得到了上面的值(例如1分钟的系数 $0.920043902 * 2048 = 1884$)：

1分钟衰减系数e: 1884
5分钟衰减系数e: 2014
15分钟衰减系数e: 2037

LOAD_FREQ为采样周期，即5秒固定周期采样一次并进行指数衰减计算。

Linux内核的Load实现流程



- scheduler_tick()->calc_global_load_tick(), 当每一个CPU在时钟中断里，以5秒为周期更新全局变量“calc_load_tasks”。
- do_timer()->calc_global_load(), 5秒固定周期采样并更新calc_load_tasks，进行指数衰减计算，依次对load 1/5/15作全局计算calc_load()。
- 处理tickless情况下的Load数据进行修正, 更新calc_load_nohz[2]。



calc_load()代码实现

```
include/linux/sched/loadavg.h:
/*
 * a1 = a0 * e + a * (1 - e)
 */
static inline unsigned long
calc_load(unsigned long load, unsigned long exp, unsigned long active)
{
    unsigned long newload;

    newload = load * exp + active * (FIXED_1 - exp);
    if (active >= load)
        newload += FIXED_1 - 1;

    return newload / FIXED_1;
}
```

调用关系：

```
calc_global_load() ->
calc_load(avenrun[0], EXP_1, active)
calc_load(avenrun[0], EXP_5, active)
calc_load(avenrun[0], EXP_15, active)
```

定点数转换为浮点数显示：

```
LOAD_INT(avnrun[0])
LOAD_FRAC(avnrun[0])
```

```
#define LOAD_INT(x) ((x) >> FSHIFT)
#define LOAD_FRAC(x)
LOAD_INT(((x) & (FIXED_1 - 1)) * 100)
```



Linux内核Load的不足之处

社区内核

Load是内核5秒周期性打点统计R状态和D状态的任务数，以指数平滑算法计算出来的一个全局数值。

存在如下问题：

- 依靠时钟中断打点采样，理论上存在着误差。
- Load包括R和D状态，这两种状态的差异性很大，但并没有区分。
- 不支持容器化。



龙蜥内核对Load的补充

- 引入 Load.r 和 Load.d
- 引入 per-cgroup Load(包括Load.r和Load.d)

```
$ cat /sys/fs/cgroup/cpuacct/cpuacct.proc_stat
```

...

```
load average(1min) 108
```

```
load average(5min) 102
```

```
load average(15min) 101
```

```
nr_running 1
```

```
nr_uninterruptible 1
```

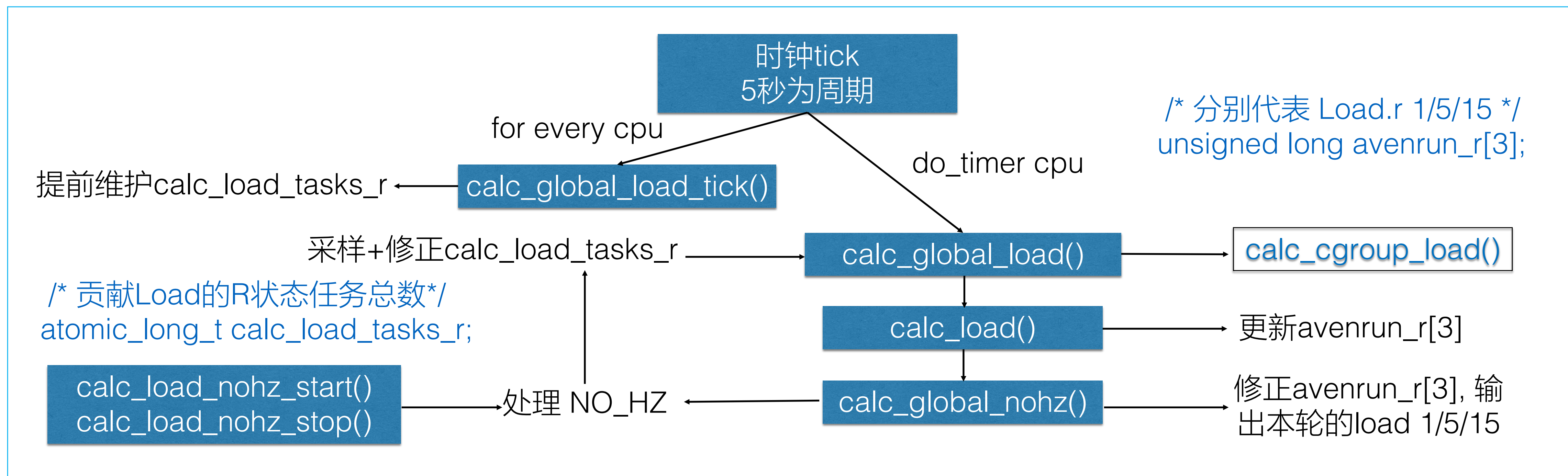
```
running load average(1min) 8
```

```
running load average(5min) 2
```

```
running load average(15min) 1
```




龙蜥内核的Load补充流程



- 引入calc_load_tasks_r和avenrun_r[3]，实现全局Load.r 1/5/15。
- do_timer()->calc_global_load()->calc_cgroup_load()，遍历计算per-cgroup Load。
- 引入calc_load_nohz_r[2]处理NO_HZ情况。



龙蜥内核的Load补充流程实现难点

- Per-cgroup Load计算时需避免由于多次遍历(想像一下成千上万个cgroup)导致的长时间关中断，必要时采取异步方式。
- 异步方式下的及时性保证，如果不及时会影响Load的准确性。
- 需要维护per-cgroup R和D状态任务数目，特别是D状态的，包含CFS，RT, Deadline等。
- 不同于全局Load, per-cgroup Load需要考虑Bandwidth Control throttle导致的Load.r虚高。
- 不能影响业务性能。
- Per-cgroup Load计算考虑NO_HZ。(TODO)



Load 高实战分享



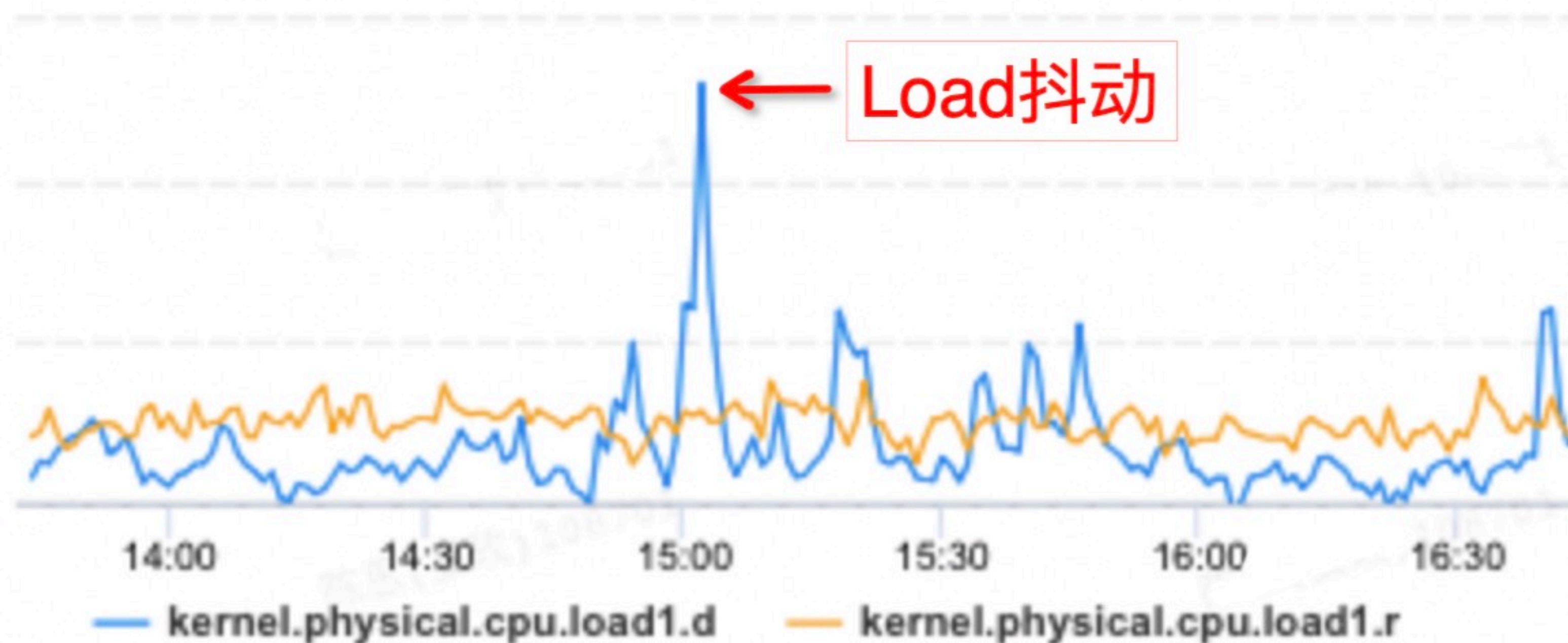
数据中心集群最典型的问题 – Load高

“Load高” 是好事还是坏事？

影响load高的常见因素:

- 运行任务数目本身就多(例如有突发业务行为) -> Load.r高
- 调度争抢大 -> Load.r高, steal高
- 类spin_lock锁竞争瓶颈 -> Load.r高, steal不高, sys高
- 任务大量block阻塞(例如mutex锁阻塞, io阻塞, 显式设成D状态等) -> Load.d高

线上Load高 - 随机抖动问题



如何定位原因？ 更多实战经验进行中



Load高分析的可行解决方案

	实现方式	优 点	缺 点
经典方案	perf/ftrace/火焰图	现成标准 信息全面	资源开销大 数据量大 精准性差 很可能不是第一现场 不适合常态化在线部署
用户态方案	监视proc接口	简单安全 通用性好	用户态功能受限 资源消耗大 某些场景的精准性差 常态化在线部署挑战大
内核态方案	kprobe方式 hotfix ko方式	资源开销小 精准捕获第一现场 手段丰富灵活 可常态化在线部署	稳定性风险高 不同内核的适配成本高



经典方案分析load高 - perf

脚本实现逻辑

- 1) 读取 /proc/loadavg
- 2) 当load超出门限时，调用perf工具采集现场，以及生成火焰图
`perf record -a -g -e cycles:k -- sleep 3`
- 3) 采集过后放大load门限避免反复perf采集

方案特点：

适合抓取某段时间Load持续高的情况，例如内存压力大导致的Load高，正在发生内存回收。

用于临时抓取，不宜常态化部署。

perf实现过于复杂，曾遇到过不少BUG导致系统宕机。



用户态方案分析load高 – proc方式

脚本实现逻辑

1) 周期性读取 /proc/loadavg中的load1

2) 当load1超出某个设定门限时，打印出proc中的stack信息:

```
for T in $(ps -e -L h o state,pid | awk '{if($1=="R"||$1=="D"){print $2}}'); do cat /proc/$T/stack; done
```

3) 采集过后放大load门限避免反复采集

方案特点：

适合系统load.d高的情况，抓取D状态的任务stack。

proc全部遍历的成本比较高，用于临时抓取，不宜常态化部署。



用户态方案分析load高 - ssar

ssar (Site Reliability Engineering System Activity Reporter)，一款阿里自研并贡献至龙蜥社区的系统性能监控工具，已在阿里内部规模化部署。

作者：闻茂泉 (阿里花名 广成)

龙蜥社区开源项目地址：

https://codeup.openanolis.cn/codeup/tracing_diagnosis/ssar.git

方案特点：

ssar资源消耗低：CPU 1-3%；内存小于4MiB；非常适合做常态化部署。

ssar load5s指标: 可以让load1的准确性提升到5秒钟的精度。



ssar load5s指标

load5s指标可以让我们将load1的准确性提升到5秒钟的精度。

右图中的数据，我们可以分部分来理解：

- 绿色时间区域，load5s和load1都处于一个低水位，毫无疑问说明当时机器压力很低；
- 红色时间区域，load5s和load1都同时升高，load5s准确的代表当时机器R+D状态线程数之和，而此时load1虽然有突增到796，但却和真实的R+D状态线程数之和（9823）相差甚远。
- 蓝色时间区域，load5s已经迅速跌回到一个低水位，机器的R+D状态线程已经几乎没有了，此时机器已经没有任何压力了。但是以16:53:42这个时间点为例，load1还保持在733的高值。这是及其不准确的，这时就可以用load5s指标来补充load1指标

附load5s的计算算法：

$$\text{Load}\langle n+1 \rangle = \text{load}\langle n \rangle * e + \text{nr}\langle n+1 \rangle * (1 - e)$$

根据上面公式从load<n+1>和load<n>的采样值，反推出的nr<n+1>就是load5s。

```
$ ssar load5s -z
```

collect_datetime	threads	load1	runq	load5s	styp
2021-07-15T16:52:42	48239	9.87	8	6	5s
2021-07-15T16:52:47	48237	9.72	8	8	5s
2021-07-15T16:52:52	48237	10.06	7	14	5s
2021-07-15T16:52:57	48236	10.3	9	13	5s
2021-07-15T16:53:02	48239	10.12	14	8	5s
2021-07-15T16:53:07	48253	11.39	15	26	5s
2021-07-15T16:53:12	48282	12.24	9	22	5s
2021-07-15T16:53:17	48396	11.98	7	9	5s
2021-07-15T16:53:22	48446	11.5	9	6	5s
2021-07-15T16:53:27	48444	11.06	9	6	5s
2021-07-15T16:53:32	48405	10.81	9	8	5s
2021-07-15T16:53:37	46420	796.56	8173	9823	5s
2021-07-15T16:53:42	38433	733.41	18	8	5s
2021-07-15T16:53:47	38433	675.4	8	9	5s
2021-07-15T16:53:52	38416	621.56	13	3	5s
2021-07-15T16:53:57	38421	572.5	6	9	5s
2021-07-15T16:54:02	38427	527.22	9	7	5s
2021-07-15T16:54:07	38431	485.4	9	5	5s
2021-07-15T16:54:12	38426	447.25	3	9	5s
2021-07-15T16:54:17	38430	412.08	10	8	5s
collect_datetime	threads	load1	runq	load5s	styp
2021-07-15T16:54:22	38423	379.56	5	6	5s
2021-07-15T16:54:27	38419	349.96	7	10	5s
2021-07-15T16:54:32	38412	322.18	7	3	5s
2021-07-15T16:54:37	38417	297.1	5	9	5s
2021-07-15T16:54:42	38419	273.71	6	5	5s
2021-07-15T16:54:47	38448	252.51	8	9	5s
2021-07-15T16:54:52	38451	232.93	9	8	5s
2021-07-15T16:54:57	38439	215	9	9	5s
2021-07-15T16:55:02	38438	198.58	4	10	5s
2021-07-15T16:55:07	38437	183.48	5	10	5s
2021-07-15T16:55:12	38454	169.35	16	7	5s
2021-07-15T16:55:17	38454	156.75	6	12	5s
2021-07-15T16:55:22	38450	145.4	5	15	5s
2021-07-15T16:55:27	38453	134.32	5	7	5s
2021-07-15T16:55:32	38443	124.44	5	11	5s
2021-07-15T16:55:37	38448	115.04	11	7	5s
2021-07-15T16:55:42	38452	106.47	11	8	5s
2021-07-15T16:55:47	38443	98.9	4	12	5s
2021-07-15T16:55:52	38452	91.46	13	6	5s
2021-07-15T16:55:57	38454	85.1	9	12	5s



ssar Load.r 剖析案例

- 对于Load高主要是load.r高的情况，除了常规的机器调度的进程资源过多的原因之外，内核空间SYS CPU资源占用过高也是一个非常普遍的原因。
- 内核CPU占到了整个CPU资源的绝大比例，纯空闲CPU资源被挤压的占比非常少。这种场景下，造成load.r高的原因主要是sys CPU挤占了正常应该分配给user空间的CPU资源。
- 导致这种sys CPU使用突增原因一般是内核中忙等锁出现了抢锁情况，直接证据需要使用perf等采样工具，获取火焰图分析问题所在。根据过往经验，目前最常见的原因之一，就是整机内存紧张，并伴随内存碎片化。
- 可以看到sys CPU高的相同时刻整机Free内存只有3GiB，同时buddy中的order3内存库存枯竭。此时一些需要依赖order3内存的内核代码（如网络skb）申请内存就会陷入忙等状态。

```
$ tsar2 --cpu -i 1
```

Time	--cpu--	--cpu--	--cpu--	--cpu--	--cpu--	--cpu--
Time	user	sys	wait	hirq	sirq	util
29/11/21-20:37	55.58	7.35	0.01	0.00	0.65	63.87
29/11/21-20:38	44.30	9.46	0.00	0.00	0.89	54.96
29/11/21-20:39	44.29	10.27	0.10	0.00	0.28	55.07
29/11/21-20:40	77.12	11.06	0.00	0.00	0.55	89.07
29/11/21-20:41	64.49	10.34	0.02	0.00	1.03	76.20
29/11/21-20:42	66.04	10.45	0.01	0.00	1.06	77.90
29/11/21-20:43	74.61	15.71	0.01	0.00	0.87	91.59
29/11/21-20:44	72.59	20.79	0.01	0.00	0.65	94.46
29/11/21-20:45	7.73	89.61	0.00	0.00	0.33	98.04
29/11/21-20:46	72.32	10.78	0.02	0.00	0.88	84.41
29/11/21-20:47	70.21	9.62	0.00	0.00	0.80	80.99

```
$ tsar2 --mem -i 1
```

Time	--mem--	--mem--	--mem--	--mem--	--mem--	--mem--
Time	free	used	buff	cach	total	util
29/11/21-20:37	38.6G	76.7G	7.5G	380.6G	503.4G	15.23
29/11/21-20:38	33.4G	78.9G	7.6G	383.5G	503.4G	15.68
29/11/21-20:39	28.9G	80.6G	7.7G	386.2G	503.4G	16.01
29/11/21-20:40	27.9G	81.4G	7.7G	386.4G	503.4G	16.18
29/11/21-20:41	25.7G	86.7G	7.6G	383.4G	503.4G	17.21
29/11/21-20:42	29.5G	83.0G	7.6G	383.2G	503.4G	16.49
29/11/21-20:43	17.9G	105.0G	7.5G	373.0G	503.4G	20.85
29/11/21-20:44	11.7G	113.9G	7.4G	370.4G	503.4G	22.62
29/11/21-20:45	3.0G	131.9G	6.5G	362.0G	503.4G	25.60
29/11/21-20:46	20.3G	130.4G	5.9G	346.7G	503.4G	25.90
29/11/21-20:47	19.1G	134.9G	5.8G	343.5G	503.4G	26.80

```
$ ssar --node0 -i 1
```

collect_datetime	order0	order1	order2	order3	order4
2021-11-29T20:37:00	191935	218898	1337235	203653	624
2021-11-29T20:38:00	187127	218624	1255946	203223	362
2021-11-29T20:39:00	184403	220489	1133855	201500	84
2021-11-29T20:40:00	185363	244531	1169886	201296	134
2021-11-29T20:41:00	240329	304251	993233	191585	98
2021-11-29T20:42:00	217942	250962	1147131	199939	171
2021-11-29T20:43:00	174385	281850	628341	140807	0
2021-11-29T20:44:00	300297	273014	220858	26177	0
2021-11-29T20:45:00	238505	174386	26872	0	0
2021-11-29T20:46:00	1773975	868863	328171	14800	0
2021-11-29T20:47:00	417991	765303	451958	36589	22



ssar Load.d 剖析案例

- 对于 load5s 值大于 CPU 核数一定倍数的情况，会触发 load 详情的采集，其中针对 D 状态线程会抽样记录内核调用栈。使用 load2p 子命令可以显示详细的 D 状态调用栈信息。
- 通过load5s指标分析，loadD高从17时25分04秒到25分32秒，只持续了28秒后就迅速恢复。选取其中25分12秒采集到的调用栈（右图）分析可知引起大量线程D状态的原因是通过write系统调用大量写入磁盘，且在__jbd2_log_wait_for_space处引起了D状态。通过分析内核代码可知这是由于磁盘格式化时分配的journal 空间大小不足。
- 通过查看磁盘IO指标，在17点25分时，发生了大的IO写入，并且将磁盘util打满到100%。

```
$ ssar load2p -c 2021-09-08T17:25:12 --stackinfo
system_call_fastpath
  Sys_write
    vfs_write
      do_sync_write
        ext4_file_write
          generic_file_aio_write
            __generic_file_aio_write
              file_update_time
                ext4_ext_update_time
                  generic_update_time
                    __mark_inode_dirty
                      ext4_dirty_inode
                        __ext4_journal_start_sb
                          jbd2__journal_start
                            start_this_handle
                              __jbd2_log_wait_for_space
```

```
$ tsar2 --io -I sda5 -i 1
Time          -sda5--  -sda5--  -sda5--  -sda5--  -sda5--
Time          rs      ws      rsecs   wsecs   util
08/09/21-17:21 204.80  104.08  10.0K   6.3K    1.49
08/09/21-17:22 34.52   231.48  3.3K    33.5K   1.75
08/09/21-17:23 8.68    181.28  70.00   25.0K   1.24
08/09/21-17:24 2.6K    116.88  144.3K  2.6K    6.52
08/09/21-17:25 342.11  3263.83  3.1K    3.4M    100.00
08/09/21-17:26 141.48  94.88   9.0K    2.0K    1.08
08/09/21-17:26 8.88    96.85   71.07   2.0K    0.13
```

进一步排查写IO相关的活动，发现当时发生了一个java进程的大coredump，找到了root cause。

```
$ ls -lhr /corefile/
-rw----- 1 admin admin 254G Sep 8 17:25 core-java-59364-1603705690-test001.aliyun
```




内核态方案治理load高 – 思路

想尽一切办法抓取第一现场calltrace:

- 资源消耗低，对业务的干扰小，满足常态化部署的条件。
- 门限触发及调节策略: 触发后调指数级放大门限，一定周期内不活跃，恢复门限。
- 在中断上下文收集当前CPU的堆栈，保证信息收集的实时性和准确性。
- 周期性监控avenrun[0]变量，超出所配置一个门限值，就打印出R和D状态的堆栈。
- 注意控制打印的量，防止出现大量的R和D状态的任务，打印造成性能影响。



内核态方案治理load高 – 示例

API: `stack_trace_save_tsk()`

- 中断上下文获取当前任务栈，why? 试想一下捕获`spin_lock_irq()`的调用
- 获取D状态的任务栈，包括io阻塞和非io阻塞。
- 将超出指定门限的结果存入内存buffer中。
- 支持不同的trigger rules。
- 支持容器化。



不止可以抓取Load高

- 长时间关中断
- 长时间关抢占
- 检测调度时延
- 检测调度阻塞，hung task
- 内核全局锁竞争
- sys高
- iowait高
- 内存回收和compaction



其它思考

Load本身就是一个系统指标，所以分析清楚Load高其实也是一个系统工程！

有了calltrace后，就保证能分析出真正原因吗？

五花八门的真实线上Load高案例：

- CFS调度器有BUG，导致vruntime出了问题。
- 内核mm semaphore优先级反转，导致大量的线程阻塞在该信号量上。
- 内核发生了race引发io状态不对，导致大量的D状态任务等待。
- 不同机器上的同类型业务容器Load差异大，发现内存带宽竞争导致CPI不同。
- Disk硬件发生故障，导致io hung，导致大量的D状态任务等待io。
- ...



sysAK loadtask

sysAK(system analyse kit)，目前主要来自于阿里百万服务器运维经验，通过对这些经验进行抽象总结出典型场景，提供了一系列工具针对不同的运维需求。

龙蜥社区开源项目地址：

<https://codeup.openanolis.cn/codeup/sysAK/sysak.git>


Load高分析命令: sysak loadtask -s



OpenAnolis

钉钉扫码加入龙蜥 kernel SIG 群：
<https://openanolis.cn/sig/Cloud-Kernel>



 扫一扫群二维码，立刻加入该群。



奥运会全球指定云服务商