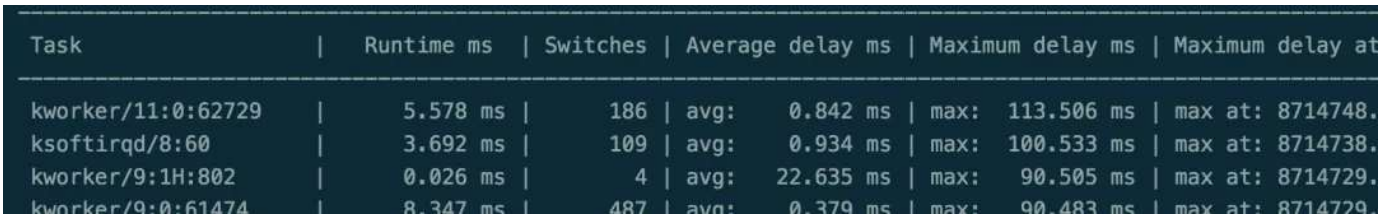
前言

随着内核升级到4.14.67，看上去延迟的问题彻底解决了，然而并没有，只是出现的更加缓慢。几周后，超时报障又找了过来，我们用perf来分析，发现了一些异常。

如图1所示是一个空负载的宿主机升级内核后8天的perf的数据，明显可以看到kworker的max delay已经100ms+，而这次有规律的是，延迟比较大的都是最后四个核，对于12核的节点就是8-11，并且都是同一D厂的宿主机。而上篇中使用新内核后用来验证解决问题的却不是D厂的宿主机，也就是说除了内核，还有其他的因素导致了延迟。



Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/11:0:62729	5.578 ms	186	avg: 0.842 ms	max: 113.506 ms	max at: 8714748.
ksoftirqd/8:60	3.692 ms	109	avg: 0.934 ms	max: 100.533 ms	max at: 8714738.
kworker/9:1H:802	0.026 ms	4	avg: 22.635 ms	max: 90.505 ms	max at: 8714729.
kworker/9:0:61474	8.347 ms	487	avg: 0.379 ms	max: 90.483 ms	max at: 8714729.

图1

NUMA和CPU亲和性绑定

NUMA全称Non-Uniform Memory Access，NUMA服务器一般有多个节点，每个节点由多个CPU组成，并且具有独立的本地内存，节点内部使用共有的内存控制器，节点之间是通过内部互联（如QPI）进行通信。

然而，访问本地内存的速度远远高于访问其他节点的远地内存的速度，通常有几十倍的性能差异，这也正是NUMA 名称的由来。因为NUMA的这个特性，为了更好地发挥系统性能，应用程序应该尽量减少不同节点CPU之间的信息交互。

无意中发现，D厂的机型与其他机型的NUMA配置不一样。假设12核的机型，D厂的机型NUMA节点分配如下图2所示：

图2

而其他厂家的机型NUMA节点分配如下图3所示：

图3

为什么会出现delay都是最后四个核上的进程呢?

经过深入排查才发现，原来相关同事之前为了让k8s的相关进程和普通的用户的进程相隔离，设置了CPU的亲和性，让k8s的相关进程绑定到宿主机的最后四个核上，用户的进程绑定到其他的核上，但后面这一步并没有生效。

还是拿12核的例子来说，上面的k8s是绑定到8-11核，但用户的进程还是工作在0-11核上，更重要的是，最后4个核在遇到D厂家的这种机型时，实际上是跨NUMA绑定，导致了延迟越来越高，而用户进程运行在相关的核上就会导致超时等各种故障。

确定问题后，解决起来就简单了。将所有宿主机的绑核去掉，延迟就消失了，以下图4是D厂的机型去掉绑核后开机26天perf的调度延迟，从数据上看一切都恢复正常。

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
revalidator34:1232	161.104 ms	465	avg: 0.240 ms	max: 12.007 ms	max at: 2409980.82
kubelet:(8)	1277.912 ms	14535	avg: 0.025 ms	max: 11.434 ms	max at: 2409979.90
revalidator36:1234	163.901 ms	360	avg: 0.353 ms	max: 10.959 ms	max at: 2409983.85
revalidator37:1235	154.776 ms	402	avg: 0.319 ms	max: 10.915 ms	max at: 2409983.85
rsyslogd:(110)	100.207 ms	7165	avg: 0.017 ms	max: 10.422 ms	max at: 2409979.60
revalidator35:1231	211.293 ms	352	avg: 0.250 ms	max: 9.564 ms	max at: 2409986.10
rs:main Q:Reg:865	4.403 ms	49	avg: 0.510 ms	max: 9.094 ms	max at: 2409984.75
redis-server:(30)	6240.949 ms	211674	avg: 0.002 ms	max: 8.992 ms	max at: 2409984.53

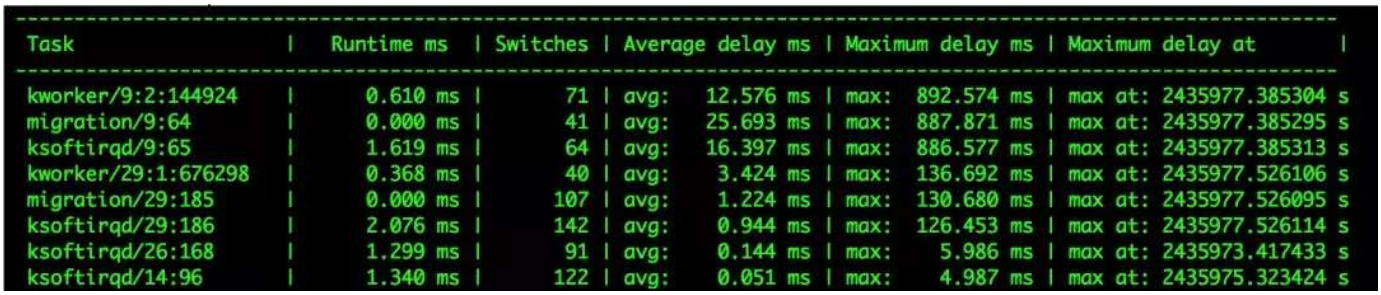
图4

新的问题

大约过了几个月，又有新的超时问题找到我们。有了之前的排查经验，我们觉得这次肯定能很轻易的定位到问题，然而现实无情地给予了我们当头一棒，4.14.67内核的宿主机，还是有大量无规律超时。

深入分析

perf看调度延迟，如图5所示，调度延迟比较大但并没有集中在最后四个核上，完全无规律，同样turbostat依然观察到TSC的频率在跳动。



Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
kworker/9:2:144924	0.610 ms	71	avg: 12.576 ms	max: 892.574 ms	max at: 2435977.385304 s
migration/9:64	0.000 ms	41	avg: 25.693 ms	max: 887.871 ms	max at: 2435977.385295 s
ksoftirqd/9:65	1.619 ms	64	avg: 16.397 ms	max: 886.577 ms	max at: 2435977.385313 s
kworker/29:1:676298	0.368 ms	40	avg: 3.424 ms	max: 136.692 ms	max at: 2435977.526106 s
migration/29:185	0.000 ms	107	avg: 1.224 ms	max: 130.680 ms	max at: 2435977.526095 s
ksoftirqd/29:186	2.076 ms	142	avg: 0.944 ms	max: 126.453 ms	max at: 2435977.526114 s
ksoftirqd/26:168	1.299 ms	91	avg: 0.144 ms	max: 5.986 ms	max at: 2435973.417433 s
ksoftirqd/14:96	1.340 ms	122	avg: 0.051 ms	max: 4.987 ms	max at: 2435975.323424 s

图5

在各种猜想和验证都被一一证否后，我们开始挨个排除来做测试：

1、我们将某台A宿主机实例迁移走，perf看上去恢复了正常，而将实例迁移回来，延迟又出现了。

2、另外一台B宿主机上，我们发现即使将所有的实例都清空，perf依然能录得比较高的延迟。

3、而与B相连编号同一机型的C宿主机迁移完实例后重启，perf恢复正常。这时候看B的TOP，只有一个kubelet在消耗CPU，将这台宿主机上的kubelet停掉，perf正常，开启kubelet后，问题又依旧。

这样我们基本可以确定kubelet的某些行为导致了宿主机卡顿和实例超时，对比正常/非正常的宿主机kubelet日志，每秒钟都在获取所有实例的监控信息，在非正常的宿主机上，会打印以下的日志。如图6所示：

```
I0613 16:36:43.336181   12039 container.go:529] [/system.slice] Housekeeping took 456.815669ms
I0613 16:36:45.574415   12039 container.go:529] [/] Housekeeping took 647.602654ms
I0613 16:36:56.419252   12039 container.go:529] [/] Housekeeping took 677.489223ms
```

图6

而在正常的宿主机上没有该日志或者该时间比较短，如图7所示：

```
I0613 20:53:21.500358   12039 container.go:529] [/] Housekeeping took 110.780762ms
I0613 20:53:31.779227   12039 container.go:529] [/] Housekeeping took 123.377625ms
```

图7

到这里，我们怀疑这些LOG的行为可能指向了问题的根源。查看k8s代码，可以知道在获取时间超过指定值longHousekeeping（100ms)后，k8s会记录这一行为，而updateStats即获取本地的一些监控数据，如图8代码所示：

```go
func (c *containerData) housekeepingTick(timer <-chan time.Time, longHousekeeping time.Duration) bool {
   select {
   case <-c.stop:
      // Stop housekeeping when signaled.
      return false
   case finishedChan := <-c.onDemandChan:
      // notify the calling function once housekeeping has completed
      defer close(finishedChan)
   case <-timer:
   }
   start := c.clock.Now()
   err := c.updateStats()
   if err != nil {
      if c.allowErrorLogging() {
         klog.Warningf("Failed to update stats for container \"%s\": %s", c.info.Name, err)
      }
   }
   // Log if housekeeping took too long.
   duration := c.clock.Since(start)
   if duration >= longHousekeeping {
      klog.V(3).Infof("[%s] Housekeeping took %s", c.info.Name, duration)
   }
   c.notifyOnDemand()
   c.statsLastUpdatedTime = c.clock.Now()
   return true
}
```

图8

在网上搜索相关issue，问题指向cadvisor的消耗CPU过高，
https://github.com/kubernetes/kubernetes/issues/15644
https://github.com/google/cadvisor/issues/1498
而在某个issue中指出
（https://github.com/google/cadvisor/issues/1774），
echo2 > /proc/sys/vm/drop_caches

可以暂时解决这种问题。我们尝试在有问题的机器上执行清除缓存的指令，超时问题就消失了，如图9所示。而从根本上解决这个问题，还是要减少取metrics的频率，比较耗时的metrics干脆不取或者完全隔离k8s的进程和用户进程才可以。

图9

硬件故障

在排查cadvisor导致的延迟的过程中，还发现一部分用户报障的超时，并不是cadvisor导致的，主要表现在没有Housekeeping的日志，并且perf结果看上去完全正常，说明

没有调度方面的延迟，但从TSC的获取上还是能观察到异常。

由此我们怀疑这是另一种全新的故障，最重要的是我们将某台宿主机所有业务完全迁移掉，并关闭所有可以关闭的进程，只保留内核的进程后，TSC依然不正常并伴随肉眼可见的卡顿，而这种现象跟之前DBA那台物理机卡顿的问题很相似，这告诉我们很有可能是硬件方面的问题。

从以往的排障经验来看，TSC抖动程度对于我们排查宿主机是否稳定有重要的参考作用。这时我们决定将TSC的检测程序做成一个系统服务，每100ms去取一次系统的TSC值，将TSC的差值大于指定值打印到日志中，并采集单位时间的异常条目数和最大TSC差值，放在监控系统上，来观察异常的规律。如图10所示。

图10

这样采集有几个好处：

1、程序消耗比较小，仅仅消耗几个CPU cycles的时间，完全可以忽略不计；
2、对于正常的宿主机，该日志始终为空；

3、对于有异常的宿主机，因为采集力度足够小，可以很清晰地定位到异常的时间点，这样对于宿主机偶尔抖动情况也能采集到；

恰好TSC检测的服务上线不久，一次明显的故障说明了它检测宿主机是否稳定的有效性。如图11，在某日8点多时，一台宿主机TSC突然升高，与应用的告警邮件和用户报障同一时刻到来。如图12所示：

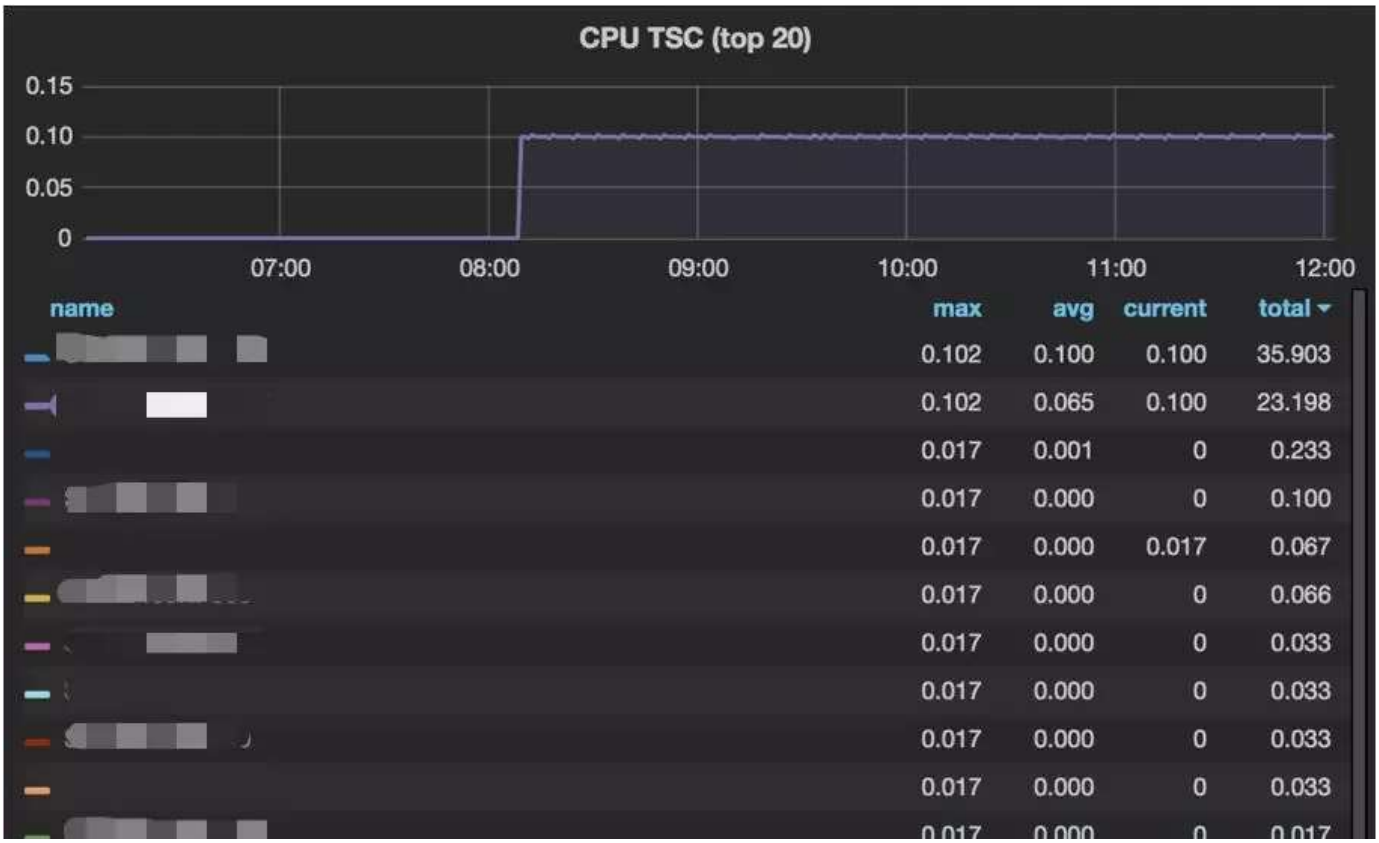


图11

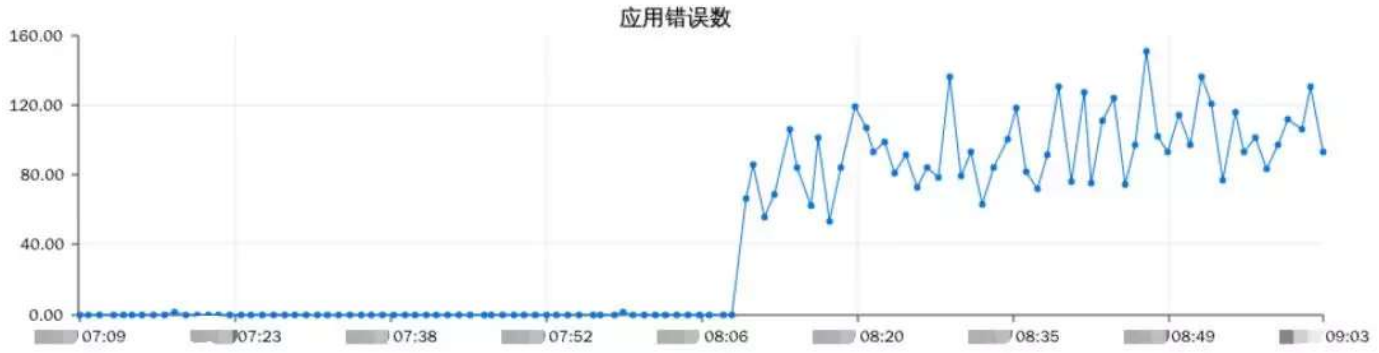


图12

将采集的日志这样展示后，我们一眼就发现问题都集中在某几批次的同一厂商的宿主机上，并且我们找到之前DBA卡顿的物理机，也是这几批次中的一台。我们收集了几台宿主机的日志详情，反馈给厂商后，确认是硬件故障，无规律并且随时可能触发，需升级BIOS，如图13厂商技术人员答复的邮件所示，至此问题得到最终解决。

【分析结论】服务器短时间产生大量UR错误，上报SMI中断到CPU，占用大量CPU资源导致OS卡顿，建议升级BIOS到的105以后版本，该版本开始在服务器端屏蔽了UR错误。

【理论分析】

1、 服务器当前版本处理UR错误的机制，当BIOS检测到UR错误时会发送SMI中断请求到CPU, 如果短时间内UR错误太多（UR风暴），就会使得CPU一直处理中断而无法响应OS，导致OS 卡顿（hang）。

2、 OS TSC时钟，内核在启动过程中会根据既定的优先级选择时钟源。优先级的排序根据时钟的精度与访问速度。
其中CPU中的TSC寄存器是精度最高（与CPU最高主频等同），访问速度最快（只需一条指令，一个时钟周期）的时钟源，因此内核优选TSC作为计时的时钟源。其它的时钟源，如HPET, ACPI-PM，PIT等则作为备选。
正常来说，TSC的频率很稳定且不受CPU调频的影响（如果CPU支持constant-tsc）。内核不应该侦测到它是unstable的。但是，计算机系统中存在一种名为SMI（System Management Interrupt）的中断，该中断不可被操作系统感知和屏蔽。如果内核校准TSC频率的计算过程quick_ pit_ calibrate ()被SMI中断干扰，就会导致计算结果偏差较大（超过1%），结果是tsc基准频率不准确。最后导致机器上的时间戳信息都不准确，可能偏慢或者偏快。

总结：服务器短时间产生大量UR错误时，上报给CPU的SMI中断太多，会导致OS卡顿，并且CPU TSC时钟异常。

图13

总结

本系列的两篇文章基本上描述了我们遇到的容器偶发性超时问题分析的大部分过程，但排障过程远比写出来要艰难。

总的原则还是大胆假设，小心求证，设法找到无规律中的规律性，保持细致耐心的钻研精神，相信这些疑难杂症终会被一一解决。