# 深入理解irqbalance

2015/11/23

renyl

# 1 介绍

1. irqbalance 服务用于自动优化硬件中断分配。它通过对定期采集（每隔10秒采集一次）的系统数据（CPU softirq和irq负载）进行分析，根据分析的结果来修改文件（/proc/irq/N/smp\_affinity）的值（即改变中断号的亲属性），从而达到优化硬件中断分配的目的。
2. irqbalance分为两种模式：Performance mode（默认模式）和 Power-save mode。
3. Performance mode时，irqbalance会将中断尽可能均匀地分配给每个CPU，从而充分利用CPU多核特性来提升性能。
4. Power-save mode时，irqbalance会将中断集中分配给某个CPU，从而保证其它空闲 CPU的睡眠时间，降低系统能耗。

注1：本文在如下平台进行irqbalance研究。

|  |  |
| --- | --- |
| - | 描述 |
| os | RHEL7.0\_x86\_64 |
| kernel | kernel-3.10.0-110.el7.x86\_64 |
| cpu | Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz |
| glibc | glibc-2.17-52.el7.x86\_64。 |
| irqbalance | v1.0.9 commit\_id:8922ff13704dd0e069c63d46a7bdad89df5f151c |

注2：irqbalance代码托管地址：<https://github.com/Irqbalance/irqbalance>

# 2 背景知识

在介绍irqbalance的实现原理之前，先了解下硬件中断的相关背景知识。系统的硬件中断主要有两大来源：网卡中断和iSCSI中断，其中网卡中断又最为常见。

## 2.1 硬件中断的角色

以网卡的接收数据为例，来看下硬件中断在其中扮演的角色，其过程如下：

1. 硬件接受：网卡通过物理层接收到数据帧，然后将其存储至网卡的缓冲队列中。
2. 硬件中断：网卡向CPU发送硬件中断告诉CPU网卡里有数据快来取。CPU响应该中断并调用中断处理程序进行处理（主要就是将网卡中的数据拷贝到内存中）。
3. 软中断：CPU把内核中的原始数据交给网络协议栈进行处理。
4. 应用接受：应用层程序通过调用相关系统调用（如recvfrom()）从网络协议栈中获取数据。

注：系统把硬件中断和软中断进行分开处理的原因：

1. 由于硬件中断仅仅只是响应一下网卡，速度非常快，同时其必须要快，因为硬件中断不容许内核抢占，如果速度不快的话，稍微有点网络负载的话，用户空间的程序将有巨大延迟，这是用户无法接受的。
2. 同时，软中断也不一定就是在硬件中断发生后就立即发生，而是会选择一个适当的时间点开始执行。

## 2.2 硬件中断的优化

如上所述，网卡的接受数据大致可以分为上述四个过程，那么针对网卡的优化就可以对上述四个过程分别进行优化：（irqbalance服务就是针对硬件中断进行优化）

1. 硬件接受：
2. 网卡的接受/发送队列的buffer是可以进行调节的，可以通过命令“ethtool -g XXX”进行调节。
3. 同时ethtool还可以调节rx-usecs来控制何时向CPU发送硬件中断。
4. 硬件中断：
5. 在网卡单队列时，一个网卡同一时刻只能由一个CPU进行响应硬件中断。（需要注意的是，由于中断处理程序速度很快，所以一般硬件中断不会导致性能瓶颈。）
6. 为了提高性能，现代网卡都是多队列的（即一个网卡可以有多个中断号），那么一个网卡可以同时有多个中断响应（即可由多个CPU同时进行处理）。

3）软件中断：

1. 在网卡单队列时，软件中断将和硬件中断采用相同的CPU，于是软件中断很有可能成为性能瓶颈。Tom Herbert为了解决这个问题，搞了个RPS（Receive Packet Steering，相关设置在/sys/class/net/eth0/queues/目录下），可以把软件中断分散到多个CPU上，避免单个CPU负载过大导致性能瓶颈，从而实现达到负载均衡。
2. 网卡多队列时，软件中断也是和硬件中断采用相关的CPU，硬件中断可以配置文件smp\_affinty\_list来设置不同的CPU亲属性，这样软中断也就均匀分配到不同的CPU上，同样达到负载均衡的效果。
3. 应用程序接受：
4. 不管是网卡单队列还是多队列，都会出现软中断处理的CPU和应用程序处理的CPU不是同一个CPU的情况，这样CPU之间就会发生IPI(处理器间中断），CPU Cache的利用率就会变低，从而影响程序性能。
5. 为了解决CPU Cache利用率低的问题，Tom Herbert又搞了个RFS（Receive Flow Streering），尽可能的让软中断和应用程序使用相同的CPU进行处理，从而提高CPU的Cache利用率，达到性能优化的目的。

注：

1. 在2.6.21版本内核之前，linux仅支持单队列的网卡，这样一块网卡只能有一个中断号，不过一个中断号可以绑定到多个CPU上，即网卡中断可以由不同的CPU响应，但某一个时刻只有一个CPU在响应。
2. 在2.6.21版本内核之后，linux开始支持多队列的网卡了，这样一块网卡可以有多个中断号，那么多个CPU就可以同时响应一个网卡上的不同中断号。
3. 网卡的中断分配是通过四元组（源IP，源Port，目的IP，目的Port）进行相关计算被分配到某一个中断号上，即相同的四元组将始终触发相同的网卡中断号。
4. Message Signaled Interrupts(MSI)是PCI规范的一个实现，可以突破CPU 256条interrupt的限制，使每个设备具有多个中断线变成可能，多队列网卡驱动给每个queue申请了MSI。

# 3 实现原理

通过研究irqbalance源代码，可以知道其处理过程，大致如下图所示：

图3-1：irqbalance的处理流程



由上图可知irqbalance服务的完整处理流程，接下来，将针对各个流程的具体实现方法进行详细说明。

## 3.1 构建CPU树和irq\_db

1. CPU树构建是通过读取两个目录（/sys/devices/system/node/和sys/devices/system/cpu/）下的文件来完成的。
2. irq\_db的构建是通过读取目录（/sys/bus/pci/devices）下的文件，以及/proc/interrupts文件来完成的。

构建完成后的CPU树和irq\_db结构，如下图所示：

图3-2 CPU树结构

图3-3 irq\_db结构



说明：

方括号表示的都是一个Glist结构体，椭圆号表示的是CPU和irq相关的信息，具体结构定义如下：

|  |  |  |
| --- | --- | --- |
| - | - | - |
| struct\_GList {  void \*data;  GList \*next;  GList \*prev;  }; | struct topo\_obj {  uint64\_t load;  uint64\_t last\_load;  uint64\_t irq\_count;  enum obj\_type\_e obj\_type;  int number;  int powersave\_mode;  cpumask\_t mask;  GList \*interrupts;  struct topo\_obj \*parent;  GList \*children;  GList \*\*obj\_type\_list; | struct irq\_info {  int irq;  int class;  int type;  int level;  int flags;  struct topo\_obj \*numa\_node;  cpumask\_t cpumask;  cpumask\_t affinity\_hint;  int hint\_policy;  uint64\_t irq\_count;  uint64\_t last\_irq\_count;  uint64\_t load;  int moved;  struct topo\_obj \*assigned\_obj;  unsigned int warned;  char \*name;  }; |

## 3.2 解析/proc/interrupts文件

1. 读取/proc/interrupts文件来统计各个irq的中断数。
2. 修改CPU树中各个irq的irq\_info结构体中的irq\_count和last\_irq\_count参数，为后面找出需要移植的irq做准备。

## 3.3 解析/proc/stat文件

1. 读取/proc/stat文件来统计各个CPU的负载，
2. 修改CPU树中各个CPU的topo\_obj结构体中的last和last\_load参数，为后面找出需要移植的irq做准备。

## 3.4 找出需要移植的irq

在CPU树中找出所有需要移植的irq，这个过程是从CPU树的底层向顶层逐层进行寻找，即从CPU->Cache\_domain->Package->Node这样一层一层向上的方法进行寻找，整个寻找过程较为复杂，详细如下：（以CPU层次为例）

1. 先找出CPU层次中，所有irq中load（irq\_info结构体的参数load）的最小值min\_load。
2. 针对CPU层次中的每个CPU进行如下判断：
   1. 只有在，当前CPU的load（topo\_obj的参数load）大于min\_load，且该CPU连接的irq数大于1时，才继续进行下一步，否则换CPU层次中的下一个CPU继续。
   2. 针对当前CPU下连接的每个irq进行如下判断操作：

|  |
| --- |
| if (cpu.load – min\_load > 2 \* irq.load )  {  cpu.load=cpu.load – irq.load  min\_load=min\_load + irq.load  }  else  return;  migrate\_irq(); |

1. 通过对CPU层次中的所有CPU进行扫描判断，最后可以得出CPU层次中所有需要进行移植的irq。

通过同样的方法，再对Cache\_domain、Package和Node层进行寻找，最终将找到整个CPU树中所有需要进行移植的irq（即rebalance\_irq\_list链表）。

## 3.5 在CPU树中移植irq

在CPU树中找出需要移植的irq（即rebalance\_irq\_list）是通过从底层向顶层一层一层寻找的，而在CPU树中移植irq则是通过顶层向底层一层一层移植的，具体过程如下：

1. 扫描rebalance\_irq\_list中的每个irq，根据irq所对应的node号（由irq\_info结构中的numa\_node参数决定），把该irq连接到CPU数中对应的node下。
2. 接下来，就是从Node->Package->cache\_domain 这样一层一层向下移植irq，具体移植过程如下：（以Package层为例针对Packages层次中的每个Package所连接的irq进行如下判断）
3. 判断当前irq是否需要进行移植（检查irq\_info结构的moved标志），如果不需要移植的话，换下一个irq进行判断。
4. 判断当前irq的移植平衡范围是否为BALANCE\_PACKAGE，如果是的话，表明不需要移动到下一层（即cache\_domain层），换下一个irq从a）开始。
5. 在当前Package层次中所连接的cache\_domain层次中，寻找一个负载最小的cache\_domain，移植该irq到负载最小的cache\_domain上。

通过同样的方法，再对Cache\_domain进行移植，最终将移植完所有的irq到CPU树中。

## 3.6 修改移植irq的CPU亲属性

修改移植irq的CPU亲属性，其过程如下：

1. 针对CPU树中的每个irq进行判断，是否是移植过的irq，如果是的话继续下一步操作。
2. 针对当前irq的hint\_policy不同进行不同处理:
3. 如果是HINT\_POLICY\_EXACT，那么该irq的CPU亲属性就是irq\_info结构的affinity\_hint参数值。
4. 如果是HINT\_POLICY\_SUBSET,那么irq的亲属性就是irq所连接的CPU子码cpu.mask和irq的irq\_info结构affinity\_hint参数之间的“与”值(即irq的亲属性 = cpu.mask & irq.affinity\_hint)。
5. 如果不是上面两种情况，表明hint\_policy为HINT\_POLICY\_IGNORE,此时irq的新属性将为irq所连接的CPU子码cpu.mask。
6. 最后，通过修改/proc/irq/N/ smp\_affinity文件中的值来完成改变irq的CPU亲属性。

# 4 使用方法

关于irqbalance的使用方法，可以通过命令行指定相关参数，也可以通过修改irqbalance的配置文件（/etc/sysconfig/irqbalance）来完成，主要参数介绍如下：

|  |  |  |
| --- | --- | --- |
| 序号 | 参数 | 说明 |
| 1 | --oneshot | irqbalance只运行一次就退出。 |
| 2 | --debug | debug模式，可以向终端打印出irqbalance运行过程中irq的终端数和负载等信息。 |
| 3 | --foreground | 前台模式运行。 |
| 4 | --hintpolicy=  [exact | subset | ignore] | 设置irq的affinity hinting如何被处理，默认值为ignore。 |
| 5 | -i | 设置禁止被irqbalance改变亲属性的irq号。 |
| 6 | --deepestcache | 这个参数将会影响到CPU树的构建架构，默认情况下该参数为2，表明cache\_domain层管理的是系统的逻辑CPU（非超线程开启的CPU）。 |
| 7 | --interval=<time> | 设置irqbalance多久针对irq进行一次balance，默认值为10秒。 |
| 8 | IRQBALANCE\_ONESHOT | 同参数 --oneshot 一样。 |
| 9 | IRQBALANCE\_DEBUG | 同参数 --debug 一样。 |
| 10 | IRQBALANCE\_BANNED\_CPUS | 同参数 --i 一样。 |

# 5 参考资料

<https://github.com/Irqbalance/irqbalance>