The BIRQ user manual

Sergey Kalichev <serj.kalichev@gmail.com>

2014

Overview

BIRQ stands for Balance IRQs. This is software to balance interrupts between CPUs on Linux while high load. The birq project is written in C. It has no external dependencies. The author of birq project is Sergey Kalichev <serj.kalichev(at)gmail.com>. The project development is sponsored by "Factor-TS" company http://www.factor-ts.ru/.

The IRQ balancing is important task for a systems with high loaded I/O. The balancer gathers the system statistics and then set the IRQ affinity to free overloaded CPUs. The statistics can contain information about CPU utilization, number of interrupts, system topology etc.

There are two well known balancer projects:

- irqbalance https://github.com/Irqbalance/irqbalance
- irqd https://github.com/vaesoo/irqd

The both projects have an advantages and disadvantages. I have used an irqbalance for a long time. It's good project but now the accumulated problems make me to start new balancer project. I will consider some irqd and irqbalace problems later but firstly I want to note two important problems with IRQ balancing that can't be solved now by any balancer including birq.

BIRQ related links

There are the BIRQ related links:

- GIT repository https://src.libcode.org/birq
- Downloads http://birq.libcode.org/files
- Issue tracker http://birq.libcode.org/issues/new
- BIRQ discussion mailing list http://groups.google.com/group/birq

Useless statistics

The one of the most important problems for IRQ balancing is a useless kernel statistics about CPU utilization by IRQs. The Linux kernel shows the following information:

- Time that CPU spent in IRQ handlers and in softirq's. See the /proc/stat. This time is a total for all IRQs and softirq's.
- Number of interrupts for each IRQ. See the /proc/stat or /proc/interrupts.

The most critical source of interrupts while IRQ balancing is a networking. Earlier the network drivers were fully interrupt driven. The greater traffic leads to greater number of interrupts from network card. In general the interfaces with greater number of interrupts lead to greater load of CPU. But now this is not a case.

Network drivers in modern Linux kernel use NAPI mechanism to increase performance for high bandwidth. The driver can disable hardware interrupts and schedule softirq to poll for new data later. The softirq will poll for the new data, handle appropriate actions and then schedule softirq again. Note the interrupts are still disabled. The only case driver enables interrupts is when all the data were received and there is nothing to receive. Often NAPI leads to the inverse logic for the number of interrupts and CPU load. The greater traffic leads to the lesser number of interrupts. So the lesser number of interrupts in a case of NAPI means greater CPU load. But if the traffic is not very high the number of interrupts will be still large bacause the interrups is enabled almost all the time (nothing to receive -> enable interrupts).

This is a mess. It's no way to determine the CPU load originated by specified IRQ using the number of interrupts for this IRQ. Moreover the scheduled softirq's don't have any association with original IRQ. Because of softirq polling the number of interrupts is useless information to determine CPU load originated by IRQ. So the balancer know nothing about IRQ weight while choosing the IRQ to move away from overloaded CPU.

IRQ sticking

The next unsolved problem for IRQ balancing is IRQ sticking. When CPU has a really high load the polling mechanism is always on and there is no interrupts at all. The balancer can change the IRQ affinity but can't change CPU for scheduled softirq. So the polling can be executed on the current CPU indefinitely until driver receives all the data and enables interrupts.

For balancer it means the wrong statistics about CPU load and IRQ affinity. The balancer supposes the IRQ has a new affinity but actually softirq use old

CPU. The old CPU has a high load but new CPU has no additional load at all. The deceived balancer will move IRQs away from old CPU again and again. But nothing happens. Sometimes the experiments can show the empty (no IRQs) CPU with 100% load. After a while (can be a minutes) the IRQs will be really moved to another CPU. The old CPU load will become 0% but some another CPU load probably will become 100% because balancer moves IRQs to minimally loaded CPU. Often the minimally loaded CPU is the same for the series of CPU movements because moving of sticking IRQs has no effect on target CPU load. The CPU with 100% load has a great chance to make its IRQs sticking. The situation will repeat again and again.

It's no way to formally determine IRQ sticking. And I don't know the way to unstick IRQs.

Another problems and peculiarities

I'll try to show some balancing problems by example of existent balancers.

The irqd project uses Receive Packet Steering (RPS) to balance CPU load. In a several words RPS is a mechanism to share network packet processing between several CPUs. The RPS share the stream originated from one source. The source is a network interface or network receive queue. The whole stream will be processed on the single CPU without RPS. Search for RPS details in the Internet or Linux kernel documentation. The first problem with RPS - the RPS is network specific. Actually it's not a serious problem I think. The second problem is the RPS is good for one large stream but not good for many smaller streams. So it can't be universal. The experiments show the server can receive more network packets from one large stream using RPS but the load of CPUs is very high for total amount of received data. And the third problem is the name of network receive queues is not standardized. So the irqd have to parse /proc/interrupts for the IRQ source names and use fuzzy logic to find out the corresponding RPS-controlling filesystem entries. Each network driver uses its own naming scheme.

The irqbalance rely on interrupt statistics, suppose the IRQs with greater number of interrupts produce greater CPU load. Then it calculates IRQ weights to set more optimal affinities. Unfortunately it not works due to NAPI. The weights are wrong for a high loaded system. The irqbalance is too intellectual for current kernel statistics state.

The irqbalance classifies IRQs (devices) and use different balance level for different device classes. As a result some devices have affinity to several CPUs at the same time. It's not good because most of interrupt controllers actually use a single CPU anyway. It leads to wrong weight calculations.

The irqbalance consider Hyper Threading as a real processor. Two threads of one CPU core have a single execution core so the threads have a great influence

to each other. The threads can't be considered as independent CPUs.

Some BIRQ features

By default BIRQ doesn't use the second threads of Hyper Threading. These logical CPUs are ignored. May be some day birq will use Hyper Threading to share load between threads within single CPU core. But I think second thread ignore is better than using threads independently.

The birq gathers statistics of CPU utilization and choose the most overloaded one. Then it choose the IRQ to move away from overloaded CPU. The balancer can use three different strategies to choose IRQ. The strategies are:

- Choose the IRQ with maximum number of interrupts.
- Choose the IRQ with minimum number of interrupts.
- Random choose.

The experiments show the most effective strategy is random choose. Now it's default. The user can choose strategy using command line arguments for birq executable. In a case of minimal/maximal choose the problem is with periodic processes. The more intellectual IRQ placing is useless due to useless kernel statistics.

The birq doesn't use device classification. All IRQs are equals.

Actually the birq balancing is not perfect of cause. But I think the perfect balancing is not possible because of useless kernel statistics and IRQ sticking.

Usage

The current version of birq is 1.1.2.

```
$ birq [options]
```

Options :

- **-h, –help** - Print help.
- **-d, –debug** - Debug mode. Don't daemonize.
- **-v, –verbose** - Be verbose.
- **-r, –ht** - Enable Hyper Threading support. The second threads will be considered as a real CPU. Not recommended.
- **-p <path>, –pid=<path>** - File to save daemon's PID to.

- **-O <facility>, –facility=<facility>** - Syslog facility. Default is DAEMON.
- **-t <float>, –threshold=<float>** - Threshold to consider CPU is overloaded, in percents. Float value.
- **-i <sec>, –short-interval=<sec>** - Short iteration interval in seconds. It will be used when the overloaded CPU is found. Default is 2 seconds.
- **-I <sec>, –long-interval=<sec>** - Long iteration interval in seconds. It will be used when there is no overloaded CPUs. Default is 5 seconds.
- **-s <strategy>, –strategy=<strategy>** - Strategy for choosing IRQ to move. The possible values are “min”, “max”, “rnd”. The default is “rnd”. Note the birq-1.0.0 uses **-c, –choose** option name for the same functionality.
- **-x <PATH>, –pxm=<PATH>** - Specify proximity config file. Implemented since birq-1.1.0.

Proximity

The NUMA node proximity is very important characteristic for IRQ balancing. Often the PCI buses have different distance to the CPUs from different NUMA nodes. You can see the block schemes of large servers motherboards - the PCI bridges are connected to specific NUMA node (CPU package). So the path from PCI device to non-local CPU (CPU from another NUMA node) is not direct. The IRQ handling on non-local CPUs decreases performance. For example the network IRQ handling on non-local CPU can half the performance and traffic bandwidth.

The most hardware platforms have an information about PCI devices proximity. The balancer can get this information from sysfs. But some platforms have broken proximity information or don’t have it at all. In this case the local NUMA node is set to -1 and the local CPU mask includes all the system CPUs. It’s sub-optimal.

The birq allows to set proximity manually to repair system settings. Use “-x” or “–pxm” command line option to specify proximity config file. The proximity config file looks like this:

```
0000: node 1
0000:08:00.0 node 0
0000:00:1c.6 cpumask ff
# it's comment

# empty strings are ignored
```

The first field is a PCI address. You can find out the PCI addresses using following command:

```
$ lspci -D
```

Formally the PCI address contain the following parts with ":" delimiter.

<Domain>:<Bus>:<Device>.<Func>

- Domain - 4 characters. Example "0000"
- Bus - 2 characters. Example "08"
- Device - 2 characters. Example "1c"
- Func - 1 character. Example "6". Note the delimiter between and is "."

The proximity config file can contain incomplete PCI address. The first line of example says the all PCI devices with address started with "0000:" (i.e. Domain address equal to "0000") will be binded to NUMA node 1. The second line instructs birq to bind PCI device with address "0000:08:00.0" to the NUMA node 0. The third line instructs to bind PCI device "0000:00:1c.6" to CPUs with numbers 0, 1, 2, 3, 4, 5, 6, 7. The "ff" is a CPU mask to bind to. It allows to bind PCI devices in more specific way than the NUMA node definition.

If PCI device address matches the several lines within config file then the more specific (longer) line will be used. The PCI device "0000:08:00.0" matches the first and second lines. The second line will be used because the "0000:08:00.0" is more specific than "0000:".

Note you don't need proximity config file if your platform shows right values for PCI device proximity.