性能调优

- 基础知识
- Tiptop
- uptime
- vmstat
- info proc mappings
- pmap
- ArchLinux上清理磁盘空间
- 如何度量代码运行了多少时钟周期（X86平台）
- strace
- netstat
- systemtap
- ebpf
- ss
- neofetch
- lscpu
- Nmon
- 如何计算进程占用的物理内存
- dmidecode
- uftrace
- 度量命令的时间
- 查看内存
- 查看进程所运行的CPU
- AddressSanitizer

开源调优软件

- UB Sanitizer 和 Savior模糊漏洞测试
- address Sanitizer
- intel vtune implifier
- openresty
- perf trace
- perf-cpu
- packetdrill

系统架构优化几步法

- 基础知识
- 优化五步法
- CPU与内存子系统调优
  - top
  - perf
  - numactl
  - 调整并发线程数
  - numa优化
- 网络子系统调优
  - ethtool
  - strace
  - 优化方式
- 磁盘io子系统调优
  - iostat
  - blktrace
- 应用程序调优
    - 优化编译
    - 锁优化
    - jemalloc
    - cacheline优化
    - 热点函数优化

DPDK优化专题

性能调优

基础知识

在`Linux kernel`中，`0`号进程是`scheduler`，`1`号进程是`init/systemd`（所有`user thread`的祖先），`2`号进程是`[kthreadd]`（所有`kernel thread`的父进程）。

守护进程(daemon)是一类在后台运行的特殊进程，用于执行特定的系统任务。 很多守护进程在系统引导的时候启动，并且一直运行直到系统关闭。 另一些只在需要的时候才启动，完成任务后就自动结束。

Tiptop

Tiptop是一个`Linux`系统性能工具，它通过读取`CPU`硬件计数器的信息（比如`cahche miss`，`executed instructions per cycle`等等），使我们对程序的执行效率有了更清晰的认识：

uptime

`uptime`可以用来检查`Linux`系统的负载状况：

```
$ uptime
 22:53:34 up 169 days,  6:47, 19 users,  load average: 1.99, 2.00, 2.03
```

`load average`后面的`3`个值分别是系统在过去`1`，`5`和`15`分钟负载的平均值（这里的负载包含`3`种进程：当前正在被`CPU`执行的，一切条件就绪等待`CPU`调度的，和等待`I/O`操作结果的）。

对于`OpenBSD`来说，由于其没有`/proc`文件系统。它的`uptime`实现则是通过`sysctl`系统调用读取`vm.loadavg`的值。

vmstat

`vmstat`也可以用来检查`Linux`系统的负载状况：

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 32312576 1716732 74998032    0    0     0     8    0    0  2  0 98  0  0
 2  0      0 32312264 1716732 74998032    0    0     0     0 3059  139  5  0 95  0  0
 ......
```

其中`r`这一栏表示当前正在被`CPU`执行的，和一切条件就绪等待`CPU`调度的进程。由于其不包含等待`I/O`操作结果的进程，所以`vmstat`比`uptime`更准确地表示系统是否“饱和”：如果`r`的值大于`CPU`数量（这里的`CPU`指一个“逻辑`CPU`”，即需要考虑物理`CPU`有多个`core`，每个`core`支持`hyper-thread`的情况），那么系统就处于饱和状态。

info proc mappings

在`Unix`平台，如果要查看某个进程的内存分布，可以使用`gdb`附着在该进程，再使用“`info proc mappings`”命令：

```
$ sudo gdb -p 1
......
(gdb) info proc mappings
process 1
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0
/usr/bin/runit
            0x401000           0x480000    0x7f000     0x1000
/usr/bin/runit
            0x480000           0x4aa000    0x2a000    0x80000
/usr/bin/runit
            0x4ab000           0x4ae000     0x3000    0xaa000
/usr/bin/runit
            0x4ae000           0x4b0000     0x2000        0x0
            0x62d000           0x650000    0x23000        0x0 [heap]
      0x7ffe5e3f3000     0x7ffe5e414000    0x21000        0x0 [stack]
      0x7ffe5e4a4000     0x7ffe5e4a7000     0x3000        0x0 [vvar]
      0x7ffe5e4a7000     0x7ffe5e4a8000     0x1000        0x0 [vdso]
```

pmap

可以使用`pmap`命令：

```
$ sudo pmap -x 1
1:   runit
Address           Kbytes     RSS   Dirty Mode  Mapping
0000000000400000       4       4       0 r---- runit
0000000000401000     508     440       0 r-x-- runit
0000000000480000     168     124       0 r---- runit
00000000004ab000      12      12      12 rw--- runit
00000000004ae000       8       8       8 rw---   [ anon ]
000000000062d000     140       8       8 rw---   [ anon ]
00007ffe5e3f3000     132      12      12 rw---   [ stack ]
00007ffe5e4a4000      12       0       0 r----   [ anon ]
00007ffe5e4a7000       4       4       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB             988     612      40
```

通过查看进程的内存分布，可以了解哪些地址是有效的，可写的；这对于调试有一定帮助。

ArchLinux上清理磁盘空间

Arch Linux: Find Disk Space by Cleaning Filesystem一文介绍如何清理`ArchLinux`上的磁盘空间：

（1）删除不用的`package`：

```
$ sudo pacman -Rns $(pacman -Qtdq)
```

（2）清理`package cache`：

```
$ sudo pacman -Sc
```

如何度量代码运行了多少时钟周期（X86平台）

Daniel Lemire的benchmark代码展示了在`X86`平台上，如何度量一段代码运行了多少时钟周期：

```
......
#define RDTSC_START(cycles)
    \
  do {
     \
    unsigned cyc_high, cyc_low;
   \
    __asm volatile("cpuid\n\t"
    \
                   "rdtsc\n\t"
    \
                   "mov %%edx, %0\n\t"
    \
                   "mov %%eax, %1\n\t"
    \
                   : "=r"(cyc_high), "=r"(cyc_low)::"%rax", "%rbx", "%rcx",
   \
                     "%rdx");
   \
    (cycles) = ((uint64_t)cyc_high << 32) | cyc_low;
    \
  } while (0)

#define RDTSC_FINAL(cycles)
    \
  do {
     \
    unsigned cyc_high, cyc_low;
   \
    __asm volatile("rdtscp\n\t"
   \
                   "mov %%edx, %0\n\t"
    \
                   "mov %%eax, %1\n\t"
    \
                   "cpuid\n\t"
    \
                   : "=r"(cyc_high), "=r"(cyc_low)::"%rax", "%rbx", "%rcx",
   \
                     "%rdx");
   \
```

```
    (cycles) = ((uint64_t)cyc_high << 32) | cyc_low;
    \
  } while (0)
......
RDTSC_START(cycles_start);
......
RDTSC_FINAL(cycles_final);
cycles_diff = (cycles_final - cycles_start);
......
```

这段代码其实参考自How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures，原理如下：

（1）测量开始和结束时的`cpuid`指令用来防止代码的乱序执行（`out-of-order`），即保证`cpuid`之前的指令不会调度到`cpuid`之后执行，因此两个`cpuid`指令之间只包含要度量的代码，没有掺杂其它的。

（2）`rdtsc`和`rdtscp`都是读取系统自启动以来的时钟周期数（`cycles`，高`32`位保存在`edx`寄存器，低`32`位保存在`eax`寄存器），并且`rdtscp`保证其之前的代码都已经完成。两次采样值相减就是我们需要的时钟周期数。

综上所述，通过`cpuid`，`rdtsc`和`rdtscp`这`3`条汇编指令，我们就可以计算出一段代码到底消耗了多少时钟周期。

P.S.，stackoverflow也有相关的讨论。

strace

strace常用来跟踪进程执行时的系统调用和所接收的信号。在Linux世界，进程不能直接访问硬件设备，当进程需要访问硬件设备(比如读取磁盘文件，接收网络数据等等)时，必须由用户态模式切换至内核态模式，通过系统调用访问硬件设备。strace可以跟踪到一个进程产生的系统调用,包括参数，返回值，执行消耗的时间。

netstat

Netstat命令用于显示本机网络连接、运行端口、路由表等信息.

systemtap

systemtap 是利用Kprobe 提供的API来实现动态地监控和跟踪运行中的Linux内核的工具，相比Kprobe，systemtap更加简单，提供给用户简单的命令行接口，以及编写内核指令的脚本语言。具体实践可以参考文末的链接。

ebpf

eBPF 全称 extended Berkeley Packet Filter，中文意思是`扩展的伯克利包过滤器`。 ebpf 能在内核中运行沙箱程序（sandbox programs）， 而无需修改内核源码或者加载内核模块。

XDP全称eXpress Data Path，即快速数据路径，XDP是Linux网络处理流程中的一个eBPF钩子，能够挂载eBPF程序，它能够在网络数据包到达网卡驱动层时对其进行处理，具有非常优秀的数据面处理性能，打通了Linux网络处理的高速公路。

ss

ss可以显示跟netstat类似的信息，但是速度却比netstat快很多，netstat是基于/proc/net/目录下查询相关信息输出的，用strace跟踪一下netstat查询tcp的连接，会看到他open的是/proc/net/tcp的信息,但是同样跟踪一下ss，你会发现他调用了cache, libc 和slabinfo：

ss快的秘密就在于它利用的是TCP协议的tcp_diag模块，而且是从内核直接读取信息。感兴趣的朋友可以去读一下他的源码： https://github.com/shemminger/iproute2/blob/master/misc/ss.c 。

常用命令：

```
ss -h //帮助信息

root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -t
State    Recv-Q    Send-Q       Local Address:Port           Peer Address:Port     Process
ESTAB    0         0            172.23.105.55:57616          100.100.30.25:http
ESTAB    0         0               127.0.0.1:43364           127.0.0.1:2379
ESTAB    0         0               127.0.0.1:2379            127.0.0.1:43364
ESTAB    0         0            172.23.105.55:ssh            36.152.119.226:2162


//展示带收发的进程信息
root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -ltp
State        Recv-Q       Send-Q             Local Address:Port               Peer Address:Port       Process
LISTEN       0            4096               127.0.0.53%lo:domain                  0.0.0.0:*           users:(("systemd-resolve",pid=2947253,fd=13))
LISTEN       0            128                      0.0.0.0:ssh                   0.0.0.0:*           users:(("sshd",pid=527,fd=3))
LISTEN       0            128                    127.0.0.1:6011                  0.0.0.0:*           users:(("sshd",pid=2938424,fd=9))
LISTEN       0            4096                   127.0.0.1:2379                  0.0.0.0:*           users:(("etcd",pid=2947216,fd=8))
LISTEN       0            4096                   127.0.0.1:2380                  0.0.0.0:*           users:(("etcd",pid=2947216,fd=7))

//统计当前系统各个状态的连接数的两种办法
netstat -an | awk '/^tcp/ {++s[$NF]} END{for (i in s) print i, s[i]}'

root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -s
Total: 122
TCP:   10 (estab 4, closed 1, orphaned 0, timewait 1)

Transport Total     IP        IPv6
RAW       1         0         1
UDP       4         3         1
TCP       9         9         0
INET      14        12        2
FRAG      0         0         0

//展示时间信息
root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -ant -o
State      Recv-Q  Send-Q   Local Address:Port       Peer Address:Port     Process
```

```
LISTEN     0       4096     127.0.0.53%lo:53             0.0.0.0:*
LISTEN     0       128            0.0.0.0:22             0.0.0.0:*
LISTEN     0       128          127.0.0.1:6011           0.0.0.0:*
LISTEN     0       4096         127.0.0.1:2379           0.0.0.0:*
LISTEN     0       4096         127.0.0.1:2380           0.0.0.0:*
ESTAB      0       0        172.23.105.55:57616    100.100.30.25:80
ESTAB      0       0            127.0.0.1:43364        127.0.0.1:2379
 timer:(keepalive,12sec,0)
ESTAB      0       0            127.0.0.1:2379         127.0.0.1:43364
timer:(keepalive,1.700ms,0)

//展示IPv4、IPv6的socket连接信息
root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -lt6
State      Recv-Q      Send-Q           Local Address:Port           Peer
Address:Port     Process
root@iZ8vbbi54mgzr5g88hd3alZ:~/vtune# ss -lt4
State      Recv-Q     Send-Q          Local Address:Port           Peer
Address:Port     Process
LISTEN     0          4096            127.0.0.53%lo:domain
 0.0.0.0:*
LISTEN     0          128                   0.0.0.0:ssh
0.0.0.0:*
LISTEN     0          128                 127.0.0.1:6011
 0.0.0.0:*
LISTEN     0          4096                127.0.0.1:2379
 0.0.0.0:*
LISTEN     0          4096                127.0.0.1:2380
 0.0.0.0:*

//查看不同state的信息，state包括：
/*1. established
2. syn-sent
3. syn-recv
4. fin-wait-1
5. fin-wait-2
6. time-wait
7. closed
8. close-wait
9. last-ack
10. closing
11. all - All of the above states
12. connected - All the states except for listen and closed
13. synchronized - All the connected states except for syn-sent
14. bucket - Show states, which are maintained as minisockets, i.e. time-
wait and syn-recv.
15. big - Opposite to bucket state.
*/
# ss -ant4 state syn-sent
Recv-Q       Send-Q             Local Address:Port               Peer
Address:Port       Process


//ss的高级用法，过滤地址和端口号，有点类似于tcpdump的用法
过滤目标端口是80的或者源端口是1723的连接，dst后面要跟空格然后加“：”：
# ss -ant dst :80 or src :1723
# ss -ant dport = :80 or sport = :1723
目标地址是111.161.68.235的连接
# ss -ant dst 111.161.68.235
```

```
源端口大于1024的端口：
# ss sport gt 1024

查看mss cwnd大小
#ss -tai
```

neofetch

Neofetch是一个显示系统信息工具，本质上就是一个`bash`脚本，支持`150`多个操作系统。

```
root@R740-3240:~/polo# neofetch
            .-/+oossssoo+/-.               root@R740-3240
        `:+ssssssssssssssssss+:`           --------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 20.04.2 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: PowerEdge R740
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 5.13.0-30-generic
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: 60 days, 7 hours, 23 mins
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 2285 (dpkg), 7 (snap)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: bash 5.0.17
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1024x768
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   Terminal: /dev/pts/2
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   CPU: Intel Xeon Gold 5218R (80) @ 4.000GHz
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   GPU: 03:00.0 Matrox Electronics Systems Ltd. Integrated Matrox G200eW3 Graphics Contr
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Memory: 16868MiB / 257553MiB
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/
  +sssssssssdmydMMMMMMMNddddyssssssss+
   /sssssssssshdmNNNNmyNMMMMhssssss/
    .ossssssssssssssssssdMMMNysssso.
      -+sssssssssssssssssyyyssss+-
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.

```

lscpu

查看时钟频率与其他等信息

```
root@R740-3240:~/polo# lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          80
On-line CPU(s) list:             0-79
Thread(s) per core:              2
Core(s) per socket:              20
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
```

```
Model:                           85
Model name:                      Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
Stepping:                        7
CPU MHz:                         4000.000
CPU max MHz:                     4000.0000
CPU min MHz:                     800.0000
BogoMIPS:                        4200.00
Virtualization:                  VT-x
```

Nmon

Nmon是`Linux`系统下一个简单但强大的性能监控工具。

（1）获取性能数据。`/proc`文件系统是个宝库，你想要的信息几乎都可以从这里得到：
a）`CPU`利用率：`/proc/stat`；
b）内存利用率：`/proc/meminfo`和`/proc/vmstat`；
c）磁盘利用率：`/proc/diskstats`；
d）网络利用率：`/proc/net/dev`；
e）单独进程的状态：`/proc/[pid]`目录；
f）其余感兴趣的信息：比如关于系统的负载状况，可以读取`/proc/loadavg`。

（2）理解和解析数据。参考`man`手册，了解每一项数据的含义，必要时候可以阅读内核代码和学习相关的硬件知识。

（3）展示数据。`Nmon`使用的是“原始”的`ncurses`库，当然也可以使用“现代化”的`GUI`工具以达到更好的用户体验。

如果想进一步了解`nmon`内部的原理，也可参考我写的这本剖析`nmon`代码的小册子。

如何计算进程占用的物理内存

How to get the process resident set size一文不仅介绍了在`Windows`和`Unix`（包括`Linux`，`BSD`，`macOS`等等）操作系统上如何获取进程所使用的峰值和实时物理内存，并提供了现成的代码。在这里我只分析一下`Linux`上相关功能的实现：

（1）获取峰值内存：

```
size_t getPeakRSS() {
    struct rusage rusage;
    getrusage(RUSAGE_SELF, &rusage);
    return (size_t)(rusage.ru_maxrss * 1024L);
}
```

getrusage函数获取进程资源的使用情况。当第一个参数是`RUSAGE_SELF`时，表示得到当前进程的统计数据。`struct rusage`中的`ru_maxrss`即表示该进程物理内存的使用峰值，因为度量单位是`kilobytes`，故而需要乘以`1024`。

（2）获得实时内存：

```
size_t getCurrentRSS() {
    long rss = 0L;
    FILE* fp = NULL;
    if ( (fp = fopen( "/proc/self/statm", "r" )) == NULL )
        return (size_t)0L;      /* Can't open? */
    if ( fscanf( fp, "%*s%ld", &rss ) != 1 )
    {
        fclose( fp );
        return (size_t)0L;      /* Can't read? */
    }
    fclose( fp );
    return (size_t)rss * (size_t)sysconf( _SC_PAGESIZE);
}
```

`/proc/self/statm`文件共包含`7`个字段，第二个即是进程当前时刻占用的物理内存，单位是页面大小。要获得精确的字节数，还需通过`sysconf`系统调用获得页面所占据的空间，通常为`4096`。

dmidecode

dmidecode命令来获取CPU和cache信息：

```
root@R740-3240:~/polo# dmidecode -t processor -t cache
# dmidecode 3.2
Getting SMBIOS data from sysfs.
SMBIOS 3.2 present.

Handle 0x0400, DMI type 4, 48 bytes
Processor Information
        Socket Designation: CPU1
        Type: Central Processor
        Family: Xeon
        Manufacturer: Intel
        ID: 57 06 05 00 FF FB EB BF
        Signature: Type 0, Family 6, Model 85, Stepping 7
``````
        Version: Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
        Voltage: 1.8 V
        External Clock: 10400 MHz
        Max Speed: 4000 MHz
        Current Speed: 2100 MHz
        Status: Populated, Enabled
        Upgrade: Socket LGA2011
        L1 Cache Handle: 0x0703
        L2 Cache Handle: 0x0704
        L3 Cache Handle: 0x0705
        Serial Number: Not Specified
        Asset Tag: Not Specified
        Part Number: Not Specified
        Core Count: 20
        Core Enabled: 20
        Thread Count: 40
        Characteristics:
                64-bit capable
                Multi-Core
                Hardware Thread
                Execute Protection
```
```

```
                Enhanced Virtualization
                Power/Performance Control
```
        Handle 0x0703, DMI type 7, 27 bytes
Cache Information
        Socket Designation: Not Specified
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 0 kB
        Maximum Size: 0 kB
        Supported SRAM Types:
                Unknown
        Installed SRAM Type: Unknown
        Speed: Unknown
        Error Correction Type: Parity
        System Type: Unified
        Associativity: 8-way Set-associative

Handle 0x0704, DMI type 7, 27 bytes
Cache Information
        Socket Designation: Not Specified
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 0 kB
        Maximum Size: 0 kB
        Supported SRAM Types:
                Unknown
        Installed SRAM Type: Unknown
        Speed: Unknown
        Error Correction Type: Single-bit ECC
        System Type: Unified
        Associativity: 16-way Set-associative
```

查看cacheline大小

```
# cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

uftrace

`uftrace`是一个追踪和分析`C/C++`程序的工具，其灵感来自于`Linux kernel`的`ftrace`框架（项目主页：https://github.com/namhyung/uftrace）。

（1）安装。
`uftrace`依赖于`elfutils`项目中的`libelf`，所以要首先安装`libelf`，而`uftrace`的安装则很简单：

```
# git clone https://github.com/namhyung/uftrace.git
# cd uftrace
# make
# make install
```
```

（2）使用。
以这个简单程序（`test.cpp`）为例：

```
#include <cstdio>

class A {
public:
        A() {printf("A is created\n");}
        ~A() {printf("A is destroyed\n");}
};

int main() {
        A a;
        return 0;
}
```

`uftrace`要求编译时指定`-pg`或`-finstrument-functions`选项：

```
# g++ -pg test.cpp
```

编译成功后，通过`uftrace`工具可以对程序进行分析：

```
# uftrace a.out
A is created
A is destroyed
# DURATION    TID     FUNCTION
   4.051 us [ 8083] | __cxa_atexit();
            [ 8083] | main() {
            [ 8083] |   A::A() {
  13.340 us [ 8083] |     puts();
  17.321 us [ 8083] |   } /* A::A */
            [ 8083] |   A::~A() {
   1.815 us [ 8083] |     puts();
   4.679 us [ 8083] |   } /* A::~A */
  26.051 us [ 8083] | } /* main */
```

可以看到输出结果包含了程序的运行流程以及各个函数的执行时间。另外也可以使用`-k`选项追踪内核的相关函数：

```
# uftrace -k a.out
A is created
A is destroyed
# DURATION    TID     FUNCTION
   1.048 us [ 8091] | __cxa_atexit();
   0.978 us [ 8091] | sys_clock_gettime();
   0.768 us [ 8091] | main();
            [ 8091] | sys_clock_gettime() {
            [ 8091] |   A::A() {
   0.699 us [ 8091] |   } /* sys_clock_gettime */
            [ 8091] |   sys_clock_gettime() {
            [ 8091] |     puts() {
   0.768 us [ 8091] |     } /* sys_clock_gettime */
            [ 8091] |     sys_newfstat() {
   1.466 us [ 8091] |       smp_irq_work_interrupt();
   4.819 us [ 8091] |     } /* sys_newfstat */
```

```
   3.422 us [ 8091] |     __do_page_fault();
            [ 8091] |     sys_clock_gettime() {
   1.327 us [ 8091] |       smp_irq_work_interrupt();
   3.701 us [ 8091] |     } /* puts */
......
```

通常我们需要把运行结果保存下来，便于以后分析，这时可以使用`uftrace`的`record`功能：

```
# uftrace record a.out
A is created
A is destroyed
# ls
a.out  test.cpp  uftrace.data
```

可以看到在当前目录下多了一个`uftrace.data`的文件夹，里面记录了关于这次程序运行的信息，随后就可以对程序进行分析了。举个例子，可以使用`uftrace`的`replay`功能对程序的运行进行一遍“回看”：

```
# uftrace replay
# DURATION    TID     FUNCTION
   3.980 us [ 8104] | __cxa_atexit();
            [ 8104] | main() {
            [ 8104] |   A::A() {
  30.660 us [ 8104] |     puts();
  34.781 us [ 8104] |   } /* A::A */
            [ 8104] |   A::~A() {
  27.378 us [ 8104] |     puts();
  30.591 us [ 8104] |   } /* A::~A */
  69.632 us [ 8104] | } /* main */
```

综上所述，`uftrace`在下面这两个方面可以给我们很大帮助：
（1）了解程序的执行流程；
（2）度量函数的运行时间，确定热点。

度量命令的时间

```
root@R740-3240:~/polo# /usr/bin/time -v echo

        Command being timed: "echo"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: ?%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 2136
```

查看内存

`free`命令查看`Linux`系统使用内存时，`used`一项会把当前`cache`的大小也会加进去，这样会造成`free`这一栏显示的内存特别少：

```
root@R740-3240:~/polo# free -m
              total        used        free      shared  buff/cache
available
Mem:         257553       16681       51857         154      189014
238925
Swap:          2047           0        2047
```

可以使用 `-w` 命令行选项得到 `buff` 和 `cache` 各自使用的数量：

```
root@R740-3240:~/polo# free -wm
              total        used        free      shared     buffers
cache   available
Mem:         257553       16679       51859         154         640
 188373      238927
Swap:          2047           0        2047
```

查看进程所运行的CPU

本文介绍如何在 `Linux` 系统上查看某个进程（线程）所运行的 `CPU`，但在此之前我们需要弄清楚两个基本概念：

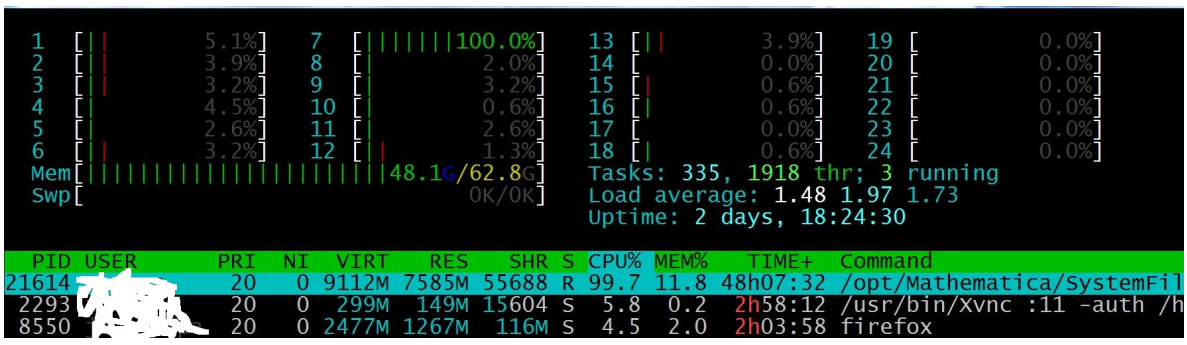
（1）`Linux` 操作系统上的进程和线程没有本质区别，在内核看来都是一个 `task`。属于同一个进程的各个线程共享某些资源，每一个线程都有一个 `ID`，而“主线程”的线程 `ID` 同进程 `ID`，也就是我们常说的 `PID` 是一样的。

（2）使用 `lscpu` 命令，可以得到当前系统 `CPU` 的数量：

```
$ lscpu
......
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
......
```

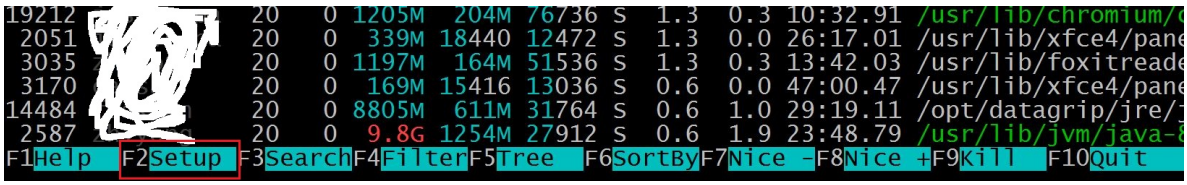
系统有 `2` 个物理 `CPU`（`Socket(s): 2`），每个 `CPU` 有 `6` 个 `core`（`Core(s) per socket: 6`），而每个 `core` 又有 `2` 个 `hardware thread`（`Thread(s) per core: 2`）。所以整个系统上一共有 `2X6X2=24`（`CPU(s):24`）个逻辑 `CPU`，也就是实际运行程序的 `CPU`。

使用 `htop` 命令可以得到进程（线程）所运行的 `CPU` 信息，但是 `htop` 默认情况下不会显示这一信息：

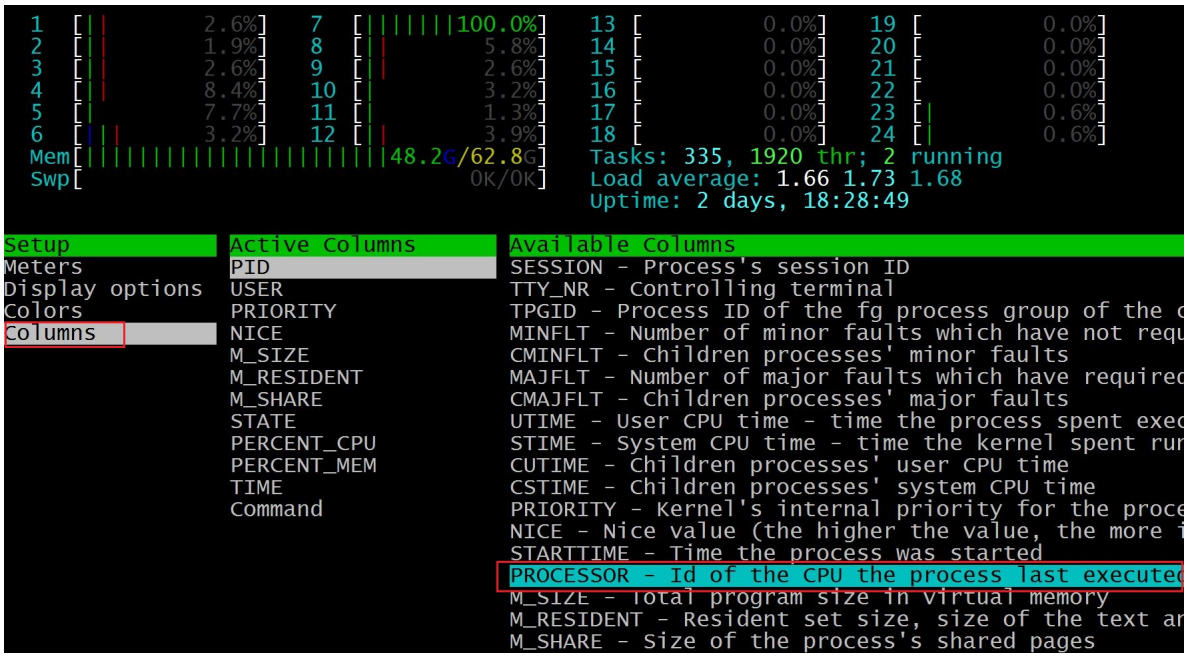


开启方法如下：

（1）启动 `htop` 后，按 `F2`（`Setup`）：



（2） 在`Setup`中选择`Columns`，然后在`Available Columns`中选择`PROCESSOR - ID of the CPU the process last executed`，接下来按`F5`（`Add`）和`F10`（`Done`）即可：



现在`htop`就会显示`CPU`的相关信息了。需要注意的是，其实`htop`显示的只是“进程（线程）之前所运行的`CPU`”，而不是“进程（线程）当前所运行的`CPU`”，因为有可能在`htop`显示的同时，操作系统已经把进程（线程）调度到其它`CPU`上运行了。

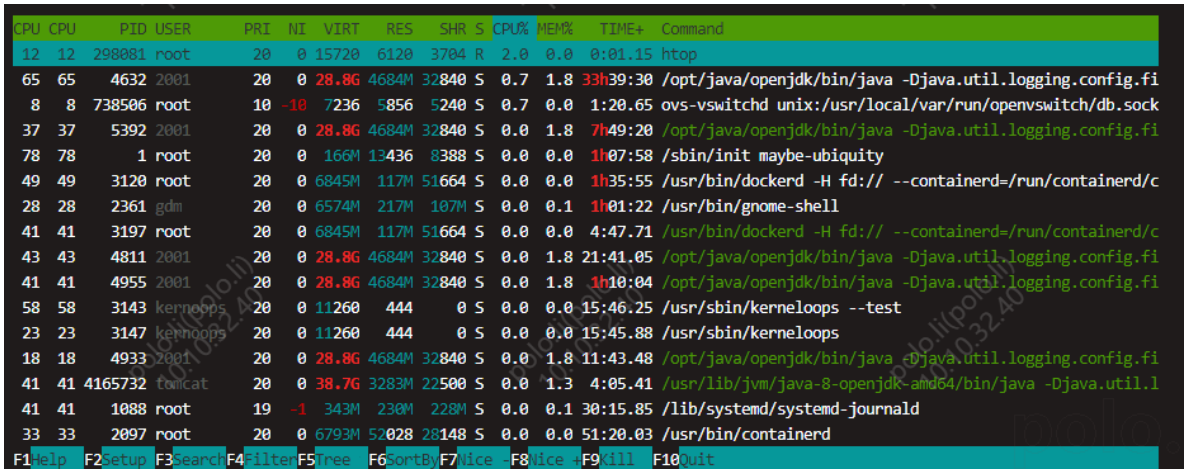
下面是一个运行时会包含`4`个线程的程序：

```
#include <omp.h>

int main(void){

        #pragma omp parallel num_threads(4)
        for(;;)
        {
        }

        return 0;
}
```

编译并运行代码：

```
$ gcc -fopenmp thread.c
$ ./a.out &
[1] 17235
```

使用`htop`命令可以得到各个线程`ID`，以及在哪个`CPU`上运行：



AddressSanitizer

AddressSanitizer是google推出用于各种内存检测的工具，它是sanitizers中的一个子功能模块，可以用于检测：

- 内存释放后又被使用
- 内存重复释放
- 释放未申请的内存
- 使用栈内存作为函数返回值
- 使用了超出作用域的栈内
- 内存越界访问

sanitizers还有ThreadSanitizer、MemorySanitizer功能，分别用于检测线程、和内存等异常，需要开启不同的gcc选项。

开源调优软件

UB Sanitizer 和 Savior模糊漏洞测试

address Sanitizer

Address Sanitizer（ASan）是一个快速的内存错误检测。它包括一个编译器instrumentation模块和一个提供malloc()/free()替代项的运行时库。 DPDK官方指导编程手册第68条已经支持如何通过meson配置使用AddressSanitizer。结合我们自己的BM场景PMD驱动调试手册中的编译步骤为

cd /home/ubuntu/workspace/gerrit/dpdk (目录自己的dpdk源码目录)

sudo meson build -Dbuildtype=debug -Db_sanitize=address

cd build

sudo ninja

sudo ninja install

intel vtune implifier

可以看网上搜索

openresty

https://openresty.com.cn/cn/xray/

perf trace

perf-cpu

参考了vic写的perf操作文档：http://wiki.dpu.tech/pages/viewpage.action?pageId=7907953

Linux性能分析工具Perf简介

perf.tar.gz

安装步骤：

下列步骤我是在arm上直接操作

1、sudo apt-get install linux-source

2、 cd /usr/src/linux-source-5.4.0

3、tar xvf linux-source-5.4.0.tar.bz2

4、cd /usr/src/linux-source-5.4.0/linux-source-5.4.0/tools/perf

5、make && make install

火焰图：

具体操作步骤如下：

Linux下用火焰图进行性能分析

首先用 perf script 工具对 perf.data 进行解析

```
# 生成折叠后的调用栈
perf script -i perf.data &> perf.unfold
将解析出来的信息存下来, 供生成火焰图
```

首先用 stackcollapse-perf.pl 将 perf 解析出的内容 perf.unfold 中的符号进行折叠：

```
# 生成火焰图
./stackcollapse-perf.pl perf.unfold &> perf.folded
```

最后生成 svg 图

./flamegraph.pl perf.folded > perf.svg

我们可以使用管道将上面的流程简化为一条命令

perf script | FlameGraph/stackcollapse-perf.pl | FlameGraph/flamegraph.pl > process.svg

火焰图是基于 stack 信息生成的 SVG 图片, 用来展示 CPU 的调用栈。

y 轴表示调用栈, 每一层都是一个函数. 调用栈越深, 火焰就越高, 顶部就是正在执行的函数, 下方都是它的父函数.

x 轴表示抽样数, 如果一个函数在 x 轴占据的宽度越宽, 就表示它被抽到的次数多, 即执行的时间长. 注意, x 轴不代表时间, 而是所有的调用栈合并后, 按字母顺序排列的.

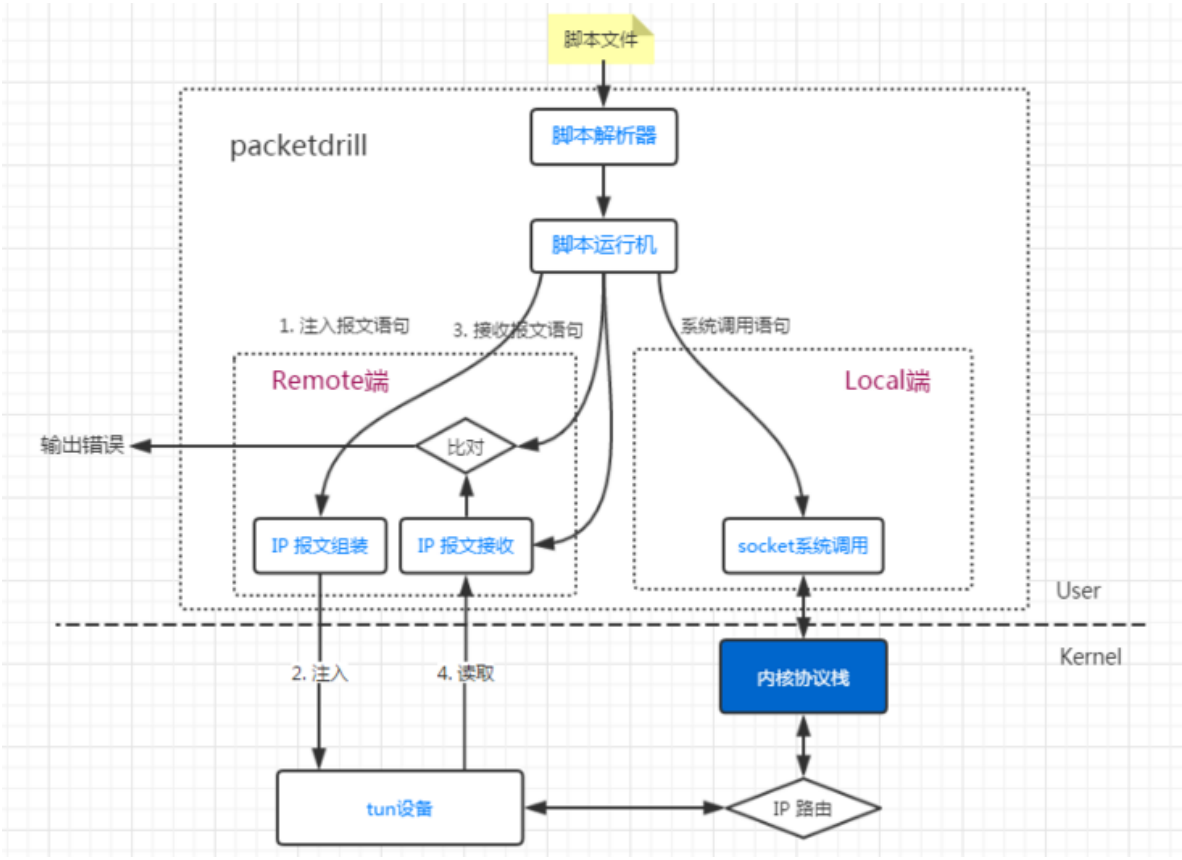
火焰图就是看顶层的哪个函数占据的宽度最大. 只要有 “平顶”(plateaus), 就表示该函数可能存在性能问题。

颜色没有特殊含义, 因为火焰图表示的是 CPU 的繁忙程度, 所以一般选择暖色调.

packetdrill

（一般用来测试内核协议栈的，没看到过有人用来测试用户态协议栈，暂不深入研究）

`packetdrill`是一个非常有用的用于测试网络协议栈的工具，由`Google`开发，它常用于对网络协议栈进行回归测试，确保新的功能不会影响原有功能 `packetdrill`的整体框架如下图所示



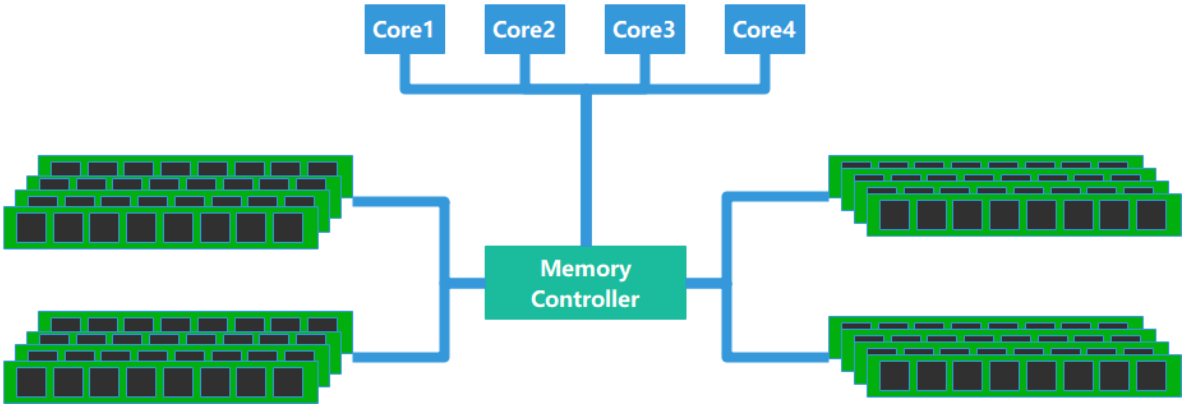
`packetdrill` 应用内部模拟了一个连接的 `Remote端` 和 `Local端`。其中 `Remote端` 用作远端发送到本机报文的通道，我们可以在 `packetdrill` 应用内向 `tun` 设备写入 `IP` 报文，对内核协议栈来来，这相当于从远端收到了这个 `IP` 报文，再经过路由，这个报文会上送协议栈。反过来说，内核协议栈的向 `Remote端` 发送的报文会通过这个 `tun` 设备回到 `packetdrill` 应用，这时，我们可以通过比对其输入，验证协议栈的功能正确性。

脚本文件是以 `.pkt` 为后缀的文件，`packetdrill` 启动后读取该文件，`脚本解析器` 将每一行脚本语句其解析为运行时 `event`，`脚本运行机` 依次执行每个 `event`。

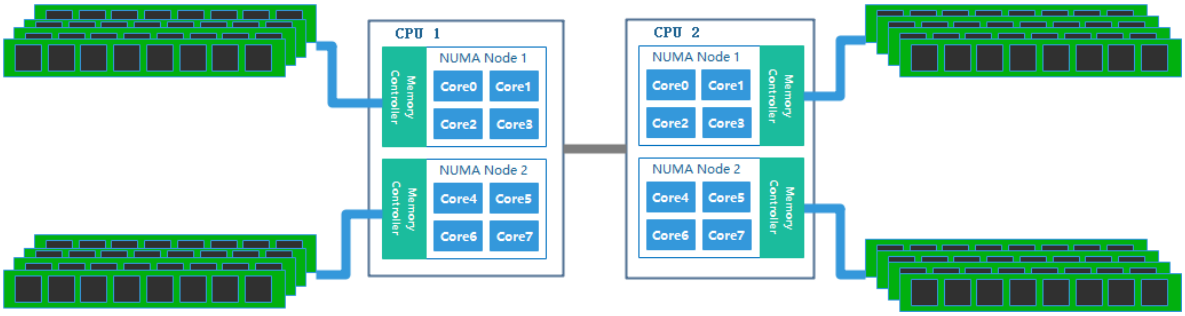
系统架构优化几步法

基础知识

传统的多核方案采用的是SMP（Symmetric Multi-Processing）技术，即对称多处理器结构，如图1所示。在对称多处理器架构下，每个处理器的地位都是平等的，对内存的使用权限也相同。任何一个程序或进程、线程都可以分配到任何一个处理器上运行，在操作系统的支持下，可以达到非常好的负载均衡，让整个系统的性能、吞吐量有较大提升。但是，由于多个核使用相同的总线访问内存，随着核数的增长，总线将成为瓶颈，制约系统的扩展性和性能。



NUMA（Non-uniform memory access, 非统一内存访问）架构，能够很好的解决SMP技术对CPU核数的制约。NUMA架构将多个核结成一个节点（Node），每一个节点相当于是一个对称多处理机（SMP），一块CPU的节点之间通过On-chip Network通讯，不同的CPU之间采用Hydra Interface实现高带宽低时延的片间通讯，如图2所示。在NUMA架构下，整个内存空间在物理上是分布式的，所有这些内存的集合就是整个系统的全局内存。每个核访问内存的时间取决于内存相对于处理器的位置，访问本地内存（本节点内）会更快一些。Linux内核从2.5版本开始支持NUMA架构 。



优化五步法

序号	步骤	说明
1	建立基准	在进行优化或者开始进行监视之前，首先要建立一个基准数据和优化目标。这个基准包括硬件配置、组网、测试模型、系统运行数据（CPU/内存/IO/网络吞吐/响应延时等）。我们需要对系统做全面的评估和监控，才能更好的分析系统性能瓶颈，以及实施优化措施后系统的性能变化。优化目标即是基于当前的软硬件架构所期望系统达成的性能目标。性能调优是一个长期的过程，在优化工作的初期，很容易识别瓶颈并实施有效的优化措施，优化成果往往也很显著，但是越到后期优化的难度就越大，优化措施更难寻找，效果也将越来越弱。因此我们建议有一个合理的平衡点。
2	压力测试与监视瓶颈	使用峰值工作负载或专业的压力测试工具，对系统进行压力测试。使用一些性能监视工具观察系统状态。在压力测试期间，建议详细记录系统和程序的运行状态，精确的历史记录将更有助于分析瓶颈和确认优化措施是否有效。
3	确定瓶颈	压力测试和监视系统的目的是为了确定瓶颈。系统的瓶颈通常会在CPU过于繁忙、IO等待、网络等待等方面出现。需要注意的是，识别瓶颈是分析整个测试系统，包括测试工具、测试工具与被测系统之间的组网、网络带宽等。有很多“性能危机”的项目其实是由于测试工具、测试组网等这些很容易被忽视的环节所导致的，在性能优化时应该首先花一点时间排查这些环节。
4	实施优化	确定了瓶颈之后，接着应该对其进行优化。本文总结了笔者所在团队在项目中所遇到的常见系统瓶颈和优化措施。我们需要注意的是，系统调优的过程是在曲折中前进，并不是所有的优化措施都会起到正面效果，负优化也是经常遇到的。所以我们在准备好优化措施的同时，也应该准备好将优化措施回滚的操作指导。避免因为实施了一些不可逆的优化措施导致重新恢复环境而浪费大量的时间和精力。
5	确认优化效果	实施优化措施后，重新启动压力测试，准备好相关的工具监视系统，确认优化效果。产生负优化效果的措施要及时回滚，调整优化方案。如果有正优化效果，但未达到优化目标，则重复步骤2“压力测试与监视瓶颈”，如达成优化目标，则需要将所有有效的优化措施和参数总结、归档，进入后续生产系统的版本发布准备等工作中。

CPU与内存子系统调优

top

top是最常用的Linux性能监测工具之一。通过top工具可以监视进程和系统整体性能。

命令参考举例：

命令	说明
top	查看系统整体的CPU、内存资源消耗。
top执行后输入1	查看每个CPU core资源使用情况。
top执行后输入F，并选择P选项	查看线程执行过程中是否调度到其它CPU core。
top -p $PID -H	查看某个进程内所有线程的CPU资源占用。

1、使用**top**命令统计整体CPU、内存资源消耗。

```
top - 09:44:27 up 61 days, 30 min, 14 users,  load average: 0.35, 0.18, 0.12
Tasks: 1024 total,   1 running, 1004 sleeping,  19 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 257553.3 total,  51851.4 free,  16704.5 used, 188997.4 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used. 238902.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4632 2001      20   0   28.8g   4.6g  32840 S   1.0   1.8   2038:36 java
    594 root      20   0       0      0      0 I   0.3   0.0   2:16.07 kworker/42:1-events
 997829 root      20   0       0      0      0 I   0.3   0.0   0:01.11 kworker/10:1-events
1017417 root      20   0       0      0      0 I   0.3   0.0   0:00.64 kworker/74:2-events
1018766 root      20   0       0      0      0 I   0.3   0.0   0:00.68 kworker/1:0-events
1025294 root      20   0       0      0      0 I   0.3   0.0   0:00.51 kworker/20:1-events
1028837 root      20   0       0      0      0 I   0.3   0.0   0:00.45 kworker/35:2-events
```

- CPU项：显示当前总的CPU时间使用分布。
  - us表示用户态程序占用的CPU时间百分比。
  - sy表示内核态程序所占用的CPU时间百分比。
  - wa表示等待IO等待占用的CPU时间百分比。
  - hi表示硬中断所占用的CPU时间百分比。
  - si表示软中断所占用的CPU时间百分比。

    通过这些参数我们可以分析CPU时间的分布，是否有较多的IO等待。在执行完调优步骤后，我们也可以对CPU使用时间进行前后对比。如果在运行相同程序、业务情况下CPU使用时间降低，说明性能有提升。
- KiB Mem：表示服务器的总内存大小以及使用情况。
- KiB Swap：表示当前所使用的Swap空间的大小。Swap空间即当内存不足的时候，把一部分硬盘空间虚拟成内存使用。如果当前所使用的Swap空间大于0，可以考虑优化应用的内存占用或增加物理内存。

2、在**top**命令执行后按1，查看每个CPU core的使用情况。通过该命令可以查看单个CPU core的使用情况，如果CPU占用集中在某几个CPU core上，可以结合业务分析触发原因，从而找到优化思路。

3、选中**top**命令的P选项，查看线程运行在哪些 CPU core上。

在**top**命令执行后按F，可以进入top命令管理界面。在该界面通过上下键移动光标到P选项，通过空格键选中后按Esc退出，即可显示出线程运行的CPU核。观察一段时间，若业务线程在不同NUMA节点内的CPU core上运行，则说明存在较多的跨NUMA访问，可通过NUMA绑核进行优化。

4、使用**top -p $PID -H**命令观察进程中每个线程的CPU资源使用。

“-p”后接的参数为待观察的进程ID。通过该命令可以找出消耗资源多的线程，随后可根据线程号分析线程中的热点函数、调用过程等情况。

perf

Perf工具是非常强大的Linux性能分析工具，可以通过该工具获得进程内的调用情况、资源消耗情况并查找分析热点函数。

命令参考举例：

命令	说明
perf top	查看当前系统中的热点函数。
perf sched record -- sleep 1 -p $PID	记录进程在1s内的系统调用。
perf sched latency --sort max	查看上一步记录的结果，以调度延迟排序。

以CentOS为例，使用如下命令安装：

```
# yum -y install perf
```

1、通过**perf top**命令查找热点函数。

该命令统计各个函数在某个性能事件上的热度，默认显示CPU占用率，可以通过“-e”监控其它事件。

- Overhead表示当前事件在全部事件中占的比例。
- Shared Object表示当前事件生产者，如kernel、perf命令、C语言库函数等。
- Symbol则表示热点事件对应的函数名称。

通过热点函数，我们可以找到消耗资源较多的行为，从而有针对性的进行优化。

2、收集一段时间内的线程调用。

perf sched record命令用于记录一段时间内，进程的调用情况。“-p”后接进程号，“sleep”后接统计时长，单位为秒。收集到的信息自动存放在当前目录下，文件名为perf.data。

```
[root@localhost ~]# perf sched record -p 11202 -- sleep 1
Warning:
PID/TID switch overriding SYSTEM[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 9.912 MB perf.data (76934 samples) ]
```

3、解析收集到的线程调度信息。

```
[root@localhost /]# perf sched latency -s max

 -----------------------------------------------------------------------------------------------------------------
  Task                  |   Runtime ms  | Switches | Average delay ms | Maximum delay ms | Maximum delay at       |
 -----------------------------------------------------------------------------------------------------------------
  rngd:5645             |      0.037 ms |        5 | avg:    0.029 ms | max:    0.040 ms | max at: 287960.092423 s
  kworker/22:6:40155    |      0.033 ms |        1 | avg:    0.008 ms | max:    0.008 ms | max at: 287960.092397 s
  cp:(2)                |      5.833 ms |        3 | avg:    0.005 ms | max:    0.008 ms | max at: 287960.091644 s
  kworker/24:1:35441    |      0.027 ms |        1 | avg:    0.006 ms | max:    0.006 ms | max at: 287960.093569 s
  kworker/28:5:45795    |      0.025 ms |        1 | avg:    0.006 ms | max:    0.006 ms | max at: 287960.093967 s
  kworker/23:1:19656    |      0.017 ms |        1 | avg:    0.006 ms | max:    0.006 ms | max at: 287960.092495 s
  kworker/25:0:29431    |      0.026 ms |        1 | avg:    0.005 ms | max:    0.005 ms | max at: 287960.093639 s
  kworker/26:3:18406    |      0.027 ms |        1 | avg:    0.005 ms | max:    0.005 ms | max at: 287960.093707 s
  VM Periodic Tas:(2)   |      0.032 ms |        2 | avg:    0.002 ms | max:    0.004 ms | max at: 287960.092592 s
  sleep:55717           |      1.637 ms |        8 | avg:    0.003 ms | max:    0.004 ms | max at: 287960.092209 s
  Hashed wheel ti:(4)   |      0.055 ms |        2 | avg:    0.003 ms | max:    0.004 ms | max at: 287960.094268 s
  sshd:(2)              |      0.362 ms |        3 | avg:    0.002 ms | max:    0.003 ms | max at: 287960.094082 s
  kworker/u130:0:40771  |      0.022 ms |        2 | avg:    0.002 ms | max:    0.003 ms | max at: 287960.094070 s
  perf:55570            |      4.205 ms |        1 | avg:    0.002 ms | max:    0.002 ms | max at: 287960.094201 s
  b2:46628              |      0.312 ms |        4 | avg:    0.001 ms | max:    0.002 ms | max at: 287960.091831 s
  kworker/u131:0:57897  |      0.009 ms |        2 | avg:    0.001 ms | max:    0.002 ms | max at: 287960.091584 s
  mysqld:(3)            |      0.018 ms |        2 | avg:    0.000 ms | max:    0.000 ms | max at:       0.000000 s
  maintenance_sch:21289 |      0.000 ms |        1 | avg:    0.000 ms | max:    0.000 ms | max at:       0.000000 s
  kworker/45:2:13004    |      0.000 ms |        1 | avg:    0.000 ms | max:    0.000 ms | max at:       0.000000 s
  :55999:55999          |      0.000 ms |        1 | avg:    0.000 ms | max:    0.000 ms | max at:       0.000000 s
 -----------------------------------------------------------------------------------------------------------------
  TOTAL:                |     12.678 ms |       43 |
 ---------------------------------------------------
```

perf sched latency命令可以解析当前目录下的perf.data文件。“-s”表示进行排序，后接参数“max”表示按照最大延迟时间大小排序。

numactl

numactl工具可用于查看当前服务器的NUMA节点配置、状态，可通过该工具将进程绑定到指定CPU core，由指定CPU core来运行对应进程。

命令参考举例：

命令	说明
numactl -H	查看当前服务器的NUMA配置。
numactl -C 0-7 ./test	将应用程序test绑定到0~7核运行。
numastat	查看当前的NUMA运行状态。

以CentOS为例，使用如下命令安装：

```
# yum -y install numactl numastat
```

1. 通过**numactl**查看当前服务器的NUMA配置。

   从**numactl**执行结果可以看到，示例服务器共划分为4个NUMA节点。每个节点包含16个CPU core，每个节点的内存大小约为64GB。同时，该命令还给出了不同节点间的距离，距离越远，跨NUMA内存访问的延时越大。应用程序运行时应减少跨NUMA访问内存。

   ```
   [root@localhost ~]# numactl -H
   available: 4 nodes (0-3)
   node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   node 0 size: 64794 MB
   node 0 free: 55404 MB
   node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
   node 1 size: 65404 MB
   node 1 free: 58642 MB
   node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
   node 2 size: 65404 MB
   node 2 free: 61181 MB
   node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
   node 3 size: 65402 MB
   node 3 free: 55592 MB
   node distances:
   node   0   1   2   3
     0:  10  15  20  20
     1:  15  10  20  20
     2:  20  20  10  15
     3:  20  20  15  10
   ```

2. 通过**numactl**将进程绑定到指定CPU core。

   通过 numactl -C 0-15 top 命令即是将进程“top”绑定到0~15 CPU core上执行。

   ```
   [root@localhost test]# numactl -C 0-15 top
   top - 20:34:41 up 3 days,  6:35,  4 users,  load average: 2.13, 2.32, 2.47
   Tasks: 706 total,   1 running, 339 sleeping,   0 stopped,   0 zombie
   %Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.6 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
   KiB Mem : 26727008+total, 23926508+free, 26837568 used,  1167424 buff/cache
   KiB Swap:  4194240 total,  4026176 free,   168064 used. 21921017+avail Mem

     PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
   22761 root      20   0  118400   8448   3712 S   1.3  0.0   1:51.20 top
   22083 root      20   0  161.9g   8.4g  86400 S   1.0  3.3 332:53.26 impalad
   ```

3. 通过**numastat**查看当前NUMA节点的内存访问命中率。

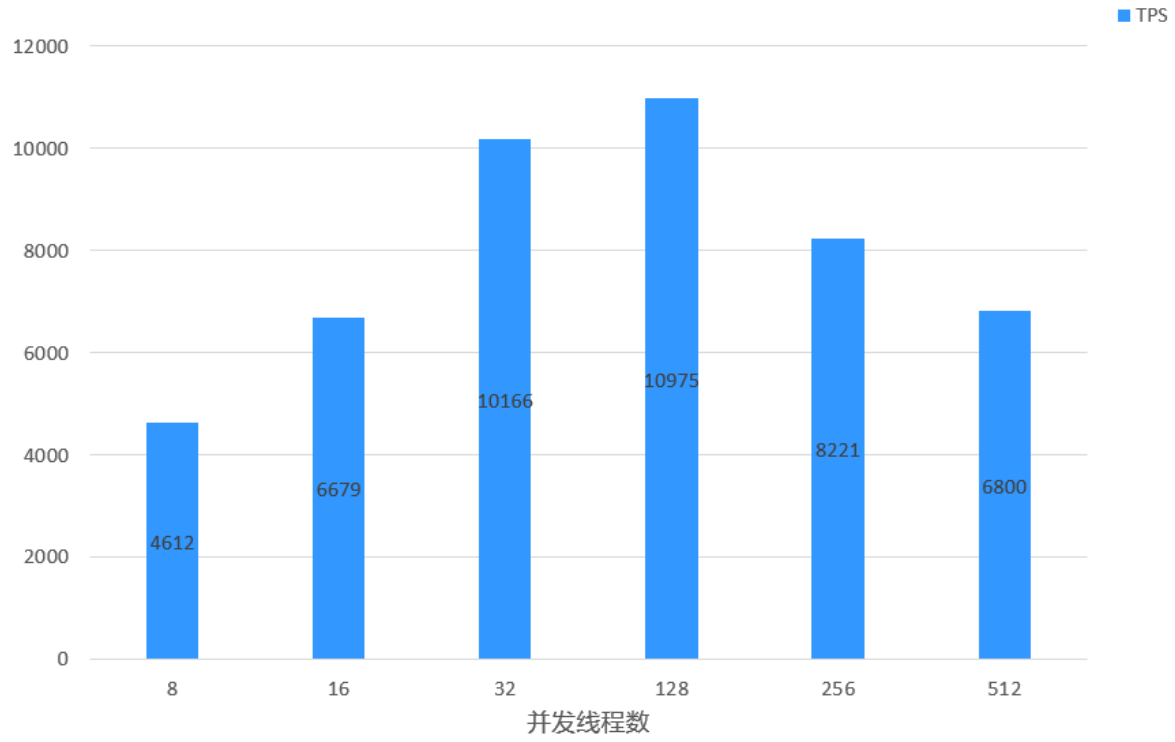
```
[root@localhost test]# numastat
                           node0           node1           node2           node3
numa_hit                68641597        59333134        50159172        55076006
numa_miss                   1           1005288          188567           2132
numa_foreign             1171587            2133               0          22268
interleave_hit              1476            1463            1477            1410
local_node              68640860        59318730        50157010        55073766
other_node                   738           1019692          190729           4372
```

可以通过**numastat**命令观察各个NUMA节点的状态。

- numa_hit表示节点内CPU核访问本地内存的次数。
- numa_miss表示节点内核访问其他节点内存的次数。跨节点的内存访问会存在高延迟从而降低性能，因此，numa_miss的值应当越低越好，如果过高，则应当考虑绑核。

调整并发线程数

程序从单线程变为多线程时，CPU和内存资源得到充分利用，性能得到提升。但是系统的性能并不会随着线程数的增长而线性提升，因为随着线程数量的增加，线程之间的调度、上下文切换、关键资源和锁的竞争也会带来很大开销。当资源的争抢比较严重时，甚至会导致性能明显降。下面数据为某业务场景下，不同并发线程数下的TPS，可以看到并发线程数达到128后，性能达到高峰，随后开始下降。我们需要针对不同的业务模型和使用场景做多组测试，找到适合本业务场景的最佳并发线程数。



不同的软件有不同的配置，需要根据代码实现来修改，这里举例几个常用开源软件的修改方法：

- MySQL可以通过innodb_thread_concurrency设置工作线程的最大并发数。
- Nginx可以通过worker_processes参数设置并发的进程个数。

numa优化

不同NUMA内的CPU core访问同一个位置的内存，性能不同。内存访问延时从高到低为：跨CPU > 跨NUMA不跨CPU > NUMA内。

因此在应用程序运行时要尽可能的避免跨NUMA访问内存，我们可以通过设置线程的CPU亲和性来实现。

- 网络可以通过如下方式绑定运行的CPU core，其中$cpuNumber是core的编号，从0开始；$irq为网卡队列中断号。

```
echo $cpuNumber > /proc/irq/$irq/smp_affinity_list
```

- 通过numactl启动程序，如下面的启动命令表示启动test程序，只能在CPU core 28到core31运行(-C控制)。

```
numactl -C 28-31 ./test
```

- 在C/C++代码中通过sched_setaffinity函数来设置线程亲和性。
- 很多开源软件已经支持在自带的配置文件中修改线程的亲和性，例如Nginx可以修改nginx.conf文件中的worker_cpu_affinity参数来设置Nginx线程亲和性。

网络子系统调优

优化项	优化项简介	默认值
调整TLP（Transaction Layer Packet）的最大有效负载	调整PCIE总线每次数据传输的最大值	128B
设置网卡队列数	调整网卡队列数量	不同操作系统和网卡不同
将每个网卡中断分别绑定到距离最近的核上	减少跨NUMA访问内存	Irqbalance
聚合中断	调整合适的参数以减少中断处理次数	不同操作系统和网卡不同
开启TCP分段Offload（卸载）	将TCP的分片处理交给网卡处理	关闭

ethtool

ethtool是一个 Linux 下功能强大的网络管理工具，目前几乎所有的网卡驱动程序都有对 ethtool 的支持，可以用于网卡状态/驱动版本信息查询、收发数据信息查询及能力配置以及网卡工作模式/链路速度等查询配置。

以CentOS为例，使用如下命令安装：

yum -y install ethtool net-tools

参数	说明
ethX	查询ethx网口基本设置，其中x是对应网卡的编号，如eth0、eth1等。
-k	查询网卡的Offload信息。
-K	修改网卡的Offload信息。
-c	查询网卡聚合信息。
-C	修改网卡聚合信息。
-l	查看网卡队列数。
-L	设置网卡队列数。

strace

strace是Linux环境下的程序调试工具，用来跟踪应用程序的系统调用情况。strace命令执行的结果就是按照调用顺序打印出所有的系统调用，包括函数名、参数列表以及返回值等。

参数	说明
-T	显示每一调用所耗的时间。
-tt	在输出中的每一行前加上时间信息，微秒级。
-p	跟踪指定的线程ID。

优化方式

- 开启TSO：ethtool -K $eth tso on
- 开启LRO：ethtool -K $eth lro on
- epoll替换select

磁盘io子系统调优

iostat

iostat是调查磁盘IO问题使用最频繁的工具。它汇总了所有在线磁盘统计信息，为负载特征归纳，使用率和饱和度提供了指标。它可以由任何用户执行，统计信息直接来源于内核，因此这个工具的开销基本可以忽略不计。

```
# iostat -d -k -x 1 100
```

常用参数如下：

参数	说明
-c	显示CPU使用情况。
-d	显示磁盘使用情况。
-k	以KB为单位显示。
-m	以M为单位显示。
-p	显示磁盘单个的情况。
-t	显示时间戳。
-x	显示详细信息。

输出格式：

```
Device:         rrqm/s   wrqm/s    r/s     w/s    rkB/s    wkB/s avgrq-sz
avgqu-sz  await svctm  %util
sda               0.02     7.25    0.04    1.90     0.74    35.47    37.15
  0.04   19.13   5.58   1.09
dm-0              0.00     0.00    0.04    3.05     0.28    12.18     8.07
  0.65  209.01   1.11   0.34
dm-1              0.00     0.00    0.02    5.82     0.46    23.26     8.13
  0.43   74.33   1.30   0.76
dm-2              0.00     0.00    0.00    0.01     0.00     0.02     8.00
  0.00    5.41   3.28   0.00
```

参数含义如下：

参数	说明
rrqm/s	每秒合并放入请求队列的读操作数。
wrqm/s	每秒合并放入请求队列的写操作数。
r/s	每秒磁盘实际完成的读I/O设备次数。
w/s	每秒磁盘实际完成的写I/O设备次数。
rkB/s	每秒从磁盘读取KB数。
wkB/s	每秒写入磁盘的KB数。
avgrq-sz	平均请求数据大小，单位为扇区（512B）。
avgqu-sz	平均I/O队列长度（操作请求数）。
await	平均每次设备I/O操作的等待时间（毫秒）。
svctm	平均每次设备I/O操作的响应时间（毫秒）。
%util	用于I/O操作时间的百分比，即使用率。

重要参数详解：

1. rrqm/s和wrqm/s，每秒合并后的读或写的次数（合并请求后，可以增加对磁盘的批处理，对HDD还可以减少寻址时间）。如果值在统计周期内为非零，也可以看出数据的读或写操作的是连续的，

反之则是随机的。

2. 如果%util接近100%（即使用率为百分百），说明产生的I/O请求太多，I/O系统已经满负荷，相应的await也会增加，CPU的wait时间百分比也会增加（通过TOP命令查看），这时很明显就是磁盘成了瓶颈，拖累整个系统。这时可以考虑更换更高性的能磁盘，或优化软件以减少对磁盘的依赖。
3. await（读写请求的平均等待时长）需要结合svctm 参考。svctm的和磁盘性能直接有关，它是磁盘内部处理的时长。await的大小一般取决于svctm以及I/O队列的长度和。svctm一般会小于await，如果svctm比较接近await，说明I/O几乎没有等待时间（处理时间也会被算作等待的一部分时间）；如果wait大于svctm，差的过高的话一定是磁盘本身IO的问题；这时可以考虑更换更快的磁盘，或优化应用。
4. 队列长度（avgqu-sz）也可作为衡量系统I/O负荷的指标，但是要多统计一段时间后查看，因为有时候只是一个峰值过高。

blktrace

blktrace是一个用户态的工具，提供I/O子系统上时间如何消耗的详细信息，从中可以分析是IO调度慢还是硬件响应慢等。配套工具blkparse从blktrace读取原始输出，并产生人们可读的输入和输出操作摘要。btt作为blktrace软件包的一部分而被提供，它分析blktrace输出，并显示该数据用在每个I/O栈区域的时间量，使它更容易在I/O子系统中发现瓶颈。

使用方式

1. 使用blktrace命令抓取指定设备的IO信息，如：

```
# blktrace -w 120 -d /dev/sda
```

“-w”后接的参数指定抓取时间，以秒为单位；“-d”后接的参数指定设备。

命令执行完后会生成一系列以*device*.blktrace.*cpu*格式命令的二进制文件。

2. 使用blkparse命令解析blktrace生成的数据，如：

```
# blkparse -i sda -d blkparse.out
```

“-i”后接的参数指定输入文件名，由于blktrace输出的文件名默认都是*device*.blktrace.*cpu*格式，所以只需输入前面的“device”即可；“-d”后接的参数指定二进制输出文件。

这个命令会将分析结果输出到屏幕，并且将分析结果的二进制数据输出到blkparse.out文件中。

3. 使用btt命令查看IO的整体情况，如：

```
# btt -i blkparse.out
```

“-i”后接的参数指定输入文件名。

```
[root@agent2 home]# btt -i blkparse.out
==================== All Devices ====================

            ALL           MIN           AVG           MAX           N
--------------- ------------- ------------- ------------- -----------

Q2Q               0.000001970   0.582514498  10.079937920         199
Q2G               0.000000720   0.000003066   0.000023720         174
G2I               0.000000200   0.000055172   0.000211490         169
Q2M               0.000000300   0.000001261   0.000002730          30
I2D               0.000000420   0.000018825   0.000049200         167
M2D               0.000006070   0.000079750   0.000192230          25
D2C               0.000035080   0.000308829   0.001894430         197
Q2C               0.000074910   0.000388102   0.001902080         197

==================== Device Overhead ====================

       DEV |       Q2G       G2I       Q2M       I2D       D2C
---------- | --------- --------- --------- --------- ---------
 (  8,  0) |   0.6977%  12.1953%   0.0495%   4.1118%  79.5742%
---------- | --------- --------- --------- --------- ---------
   Overall |   0.6977%  12.1953%   0.0495%   4.1118%  79.5742%

==================== Device Merge Information ====================

       DEV |       #Q       #D   Ratio |   BLKmin   BLKavg   BLKmax    Total
---------- | -------- -------- ------- | -------- -------- -------- --------
 (  8,  0) |      200      174     1.1 |        8      132     2048    23016

==================== Device Q2Q Seek Information ====================

       DEV |          NSEEKS            MEAN          MEDIAN | MODE
---------- | --------------- --------------- --------------- | ----------------
 (  8,  0) |             200      85956266.9               0 | 0(20)
---------- | --------------- --------------- --------------- | ----------------
   Overall |          NSEEKS            MEAN          MEDIAN | MODE
   Average |             200      85956266.9               0 | 0(20)
```

- Q2Q：多个发送到块IO层请求的时间间隔。
- Q2G：生成IO请求所消耗的时间，包括remap（可能被DM（Device Mapper）或MD（Multiple Device, Software RAID）remap到其它设备）和split（可能会因为I/O请求与扇区边界未对齐、或者size太大而被分拆(split)成多个物理I/O）的时间。
- G2I：IO请求进入IO Scheduler所消耗的时间，包括merge（可能会因为与其它I/O请求的物理位置相邻而合并成一个I/O）的时间。
- I2D：IO请求在IO Scheduler中等待的时间。
- D2C：IO请求在driver和硬件上（IO请求被driver提交给硬件，经过HBA、电缆（光纤、网线等）、交换机（SAN或网络）、最后到达存储设备，设备完成IO请求之后再把结果发回）所消耗的时间。
- Q2C：整个IO请求所消耗的时间（Q2I + I2D + D2C = Q2C），相当于iostat的await。

正常情况D2C占比90%以上，若I2D占比较高，可以尝试调整IO调度策略。

应用程序调优

优化编译

在GCC 9.1.0版本，支持了鲲鹏处理器所兼容的Armv8指令集、tsv110流水线。

可以升级GCC版本到9.10，并在CFLAGS和CPPFLAGS里面增加编译选项：

-mtune=tsv110 -march=armv8-a

锁优化

可以通过perf top分析占用CPU资源靠前的函数，如果锁的申请和释放在5%以上，可以考虑优化锁的实现，修改思路如下：

1. 大锁变小锁：并发任务高的场景下，如果系统中存在唯一的全局变量，那么每个CPU core都会申请这个全局变量对应的锁，导致这个锁的争抢严重。可以基于业务逻辑，为每个CPU core或者线程分配对应的资源。

2. 使用ldaxr+stlxr两条指令实现原子操作时，可以同时保证内存一致性，而ldxr+stxr指令并不能保存内存一致性，从而需要内存屏障指令（dmb ish）配合来实现内存一致性。从测试情况看，ldaxr+stlxr指令比ldxr+stxr+dmb ish指令的性能高。
3. 减少线程并发数：参考调整线程并发数章节。
4. 对锁变量使用Cacheline对齐：对于高频访问的锁变量，实际是对锁变量进行高频的读写操作，容易发生伪共享问题。具体优化可以参考Cacheline优化章节。
5. 优化代码中原子操作的实现。下图代码为某软件的代码实现：

```
do {
    old = (u32)atomic_read(p_id);//原子读，从缓存从读取 p_id 的值
    new = old + delta + segs;//对 p_id 执行加操作
} while (atomic_cmpxchg(p_id, old, new) != old)
/*atomic_cmpxchg 为原子写操作，先从缓存中读取 p_id 的值是否与 old 相同
若不相同，函数退出，返回 p_id 当前缓存值，并重新执行 while 循环。*/
```

从函数调用逻辑上看，在while循环会重复执行原子读、变量加以及原子写入操作，代码语句多。优化思路：使用atomic_add_return指令替换这个代码流程，简化指令，提高性能。替换后的代码如下图：

```
return atomic_add_return（segs + delta, p_id） - segs;
/* atomic_add_return  使用了 ldaxr+stlxr 指令来保障原子操作的一致性函数会先从缓存中读取 p_id 后执行加
操作，再写入内存，若操作与其他线程冲突，循环执行直到写入成功*/
```

jemalloc

jemalloc是一款内存分配器，与其它内存分配器（glibc）相比，其最大优势在于多线程场景下内存分配性能高以及内存碎片减少。充分发挥鲲鹏芯片多核多并发优势，推荐业务应用代码使用jemalloc进行内存分配。

在内存分配过程中，锁会造成线程等待，对性能影响巨大。jemalloc采用如下措施避免线程竞争锁的发生：使用线程变量，每个线程有自己的内存管理器，分配在这个线程内完成，就不需要和其它线程竞争锁。

cacheline优化

arm是128位对齐

热点函数优化

1、neon指令优化，neon内联与neon汇编等

2、预取

3、数据布局

4、其他代码优化 如乘除换移位，likely，循环内减少计算量

DPDK优化专题

cache预取

主要分为硬件预取和软件预取，软件预取 软件开发人员在性能关键区域，把即将用到的数据从内存中加载到Cache，使得当前数据处理完毕后，即将用到的数据已经在Cache中，大大减小了从内存直接读取的开销，减少CPU的等待时间，提高性能。

指令	描述信息
PREFETCHT0	预取数据到所有级别的缓存，包括L0
PREFETCHT1	预取数据到除L0外所有级别的缓存
PREFETCHT2	预取数据到除L0和L1外所有级别的缓存
PREFETCHNTA	预取数据到非临时缓冲结构中，可以最小化对缓存的污染。和PREFETCHT0 功能类似，但是数据在使用完一次后，Cache认为数据是可以淘汰出去的
_mm_prefetch(intel)	从地址P处预取尺寸为cache line大小的数据缓存，参数i指示预取方式（_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, _MM_HINT_NTA，分别对应不同的预取指令0,1，2，A)

典型可参考代码是l3-fwd

```
/*从RX队列读取数据包 * /
     for（i = 0; i <qconf-> n_rx_queue; ++ i）{
          portid = qconf-> rx_queue_list [i] .port_id;
          queueid = qconf-> rx_queue_list [i] .queue_id;
          nb_rx = rte_eth_rx_burst（portid，queueid，pkts_burst，
MAX_PKT_BURST）;
                 / *预取第一个数据包* /
                for（j = 0; j <PREFETCH_OFFSET && j <nb_rx; j ++）{
                          rte_prefetch0（rte_pktmbuf_mtod（
                                    pkts_burst [j]，void *））;
                   }
                   / *预取并转发已经预取的数据包* /
                   for（j = 0; j <（nb_rx-PREFETCH_OFFSET）; j ++）{
                          rte_prefetch0（rte_pktmbuf_mtod（pkts_burst [
                                              j + PREFETCH_OFFSET]，void
*））;
                          l3fwd_simple_forward（pkts_burst [j]，portid，
qconf-> lookup_struct）;
                       }
                 / *转发剩余的预取包* /
                 for（; j <nb_rx; j ++）{
                 l3fwd_simple_forward（pkts_burst [j]，portid，qconf-
>lookup_struct）;

```

cache一致性

这个主要解决两个问题：

1）数据结构/数据缓冲区对应的Cache Line是否对齐？如果不是的话，即使数据区域小于Cache Line的话也会占用两个Cache Line；另外假如上一个CacheLine属于另一个数据结构且被另一个处理器核处理，数据如何同步呢？

2）假设数据结构/缓冲区的起始地址是CacheLine对齐的，但是有多个核同时对该内存进行读写，如何解决冲突?

对于第一个问题：

```
CacheLine对齐，DPDK中对很多结构体的定义是这样的：


struct lcore_conf {
uint16_t nb_rx_queue;
struct lcore_rx_queue rx_queue_list[MAX_RX_QUEUE_PER_LCORE];
uint16_t tx_queue_id[RTE_MAX_ETHPORTS];
struct buffer tx_mbufs[RTE_MAX_ETHPORTS];
struct ipsec_ctx inbound;
struct ipsec_ctx outbound;
struct rt_ctx *rt4_ctx;
struct rt_ctx *rt6_ctx;

} __rte_cache_aligned;

其中_rte_cache_aligned的定义是这样的

struct rte_mempool_ops_table __rte_cache_aligned
 __rte_aligned（RTE_CACHE_LINE_SIZE）
强制对齐缓存行。

定义在文件rte_memory.h的第62行。
#define RTE_CACHE_LINE_MIN_SIZE 64
#define __rte_cache_aligned __rte_aligned(RTE_CACHE_LINE_SIZE)
#define __rte_cache_min_aligned __rte_aligned(RTE_CACHE_LINE_MIN_SIZE)

PS:当前我们的pmd代码有很多结构设计并未考虑到这一点，还需要改进
```

对于第二个问题：

主要介绍总线窥探协议。即被X86，ARM，Power等架构广泛采用著名的MESI协议

MESI协议将cache line的状态分成modify、exclusive、shared、invalid，分别是修改、独占、共享和失效。

失效（Invalid）缓存段，要么已经不在缓存中，要么它的内容已经过时。为了达到缓存的目的，这种状态的段将会被忽略。一旦缓存段被标记为失效，那效果就等同于它从来没被加载到缓存中。

共享（Shared）缓存段，它是和主内存内容保持一致的一份拷贝，在这种状态下的缓存段只能被读取，不能被写入。多组缓存可以同时拥有针对同一内存地址的共享缓存段，这就是名称的由来。

独占（Exclusive）缓存段，和S状态一样，也是和主内存内容保持一致的一份拷贝。区别在于，如果一个处理器持有了某个E状态的缓存段，那其他处理器就不能同时持有它，所以叫“独占”。这意味着，如果其他处理器原本也持有同一缓存段，那么它会马上变成“失效”状态。

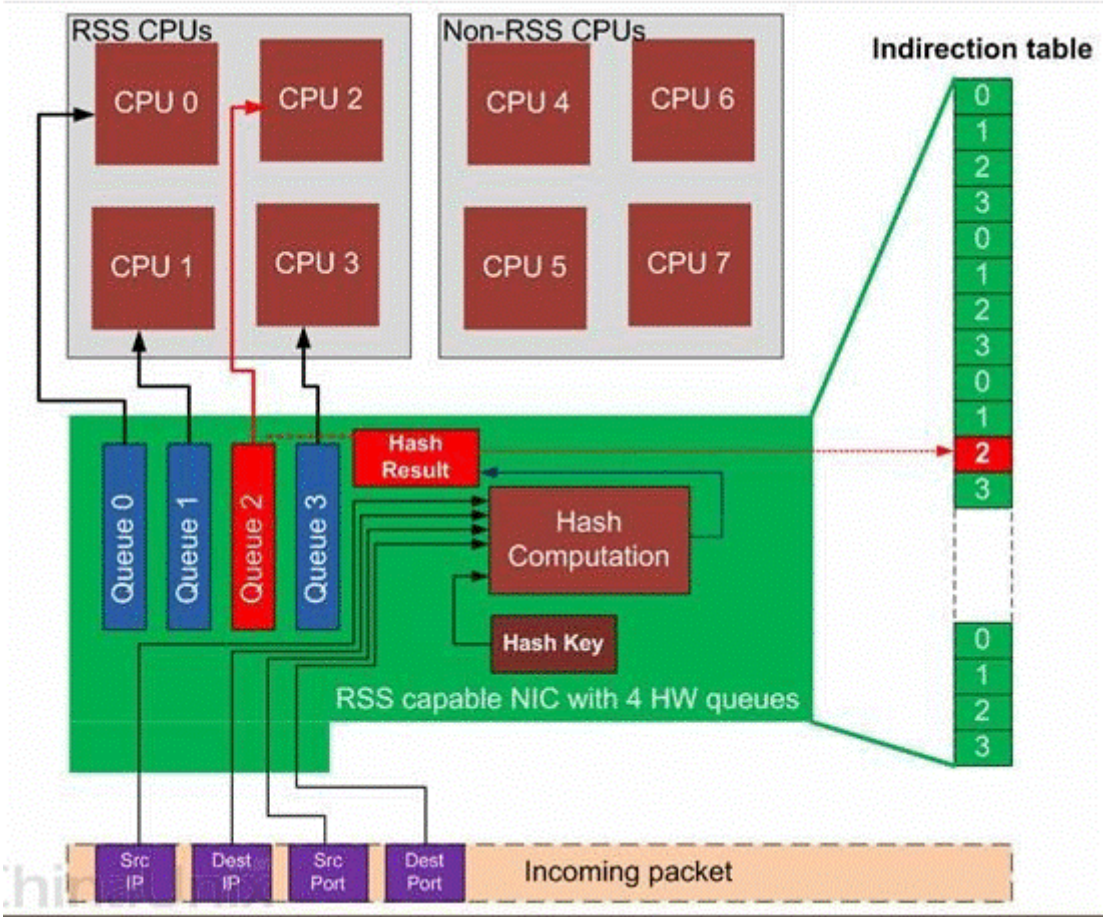
已修改（Modified）缓存段，属于脏段，它们已经被所属的处理器修改了。如果一个段处于已修改状态，那么它在其他处理器缓存中的拷贝马上会变成失效状态，这个规律和E状态一样。此外，已修改缓存段如果被丢弃或标记为失效，那么先要把它的内容回写到内存中——这和回写模式下常规的脏段处理方式一样。

DPDK解决方案很简单，首先避免多个核访问同一个内存地址或者数据结构。每个核尽量避免与其他核共享数据，从而减少因为错误的数据共享导致的Cache一致性开销。

举两个DPDK避免Cache一致性的例子：

例子1：`struct lcore_conf lcore[RTE_MAX_LCORE]_rte_cache_aligned;`

例子2：多核情况下，有可能多个核访问同一个网卡的收发队列，DPDK就会为每个核单独准备一个收发队列。该技术就是RSS (Receive Side Scaling)，是一种能够在多处理器系统下使接收报文在多个CPU之间高效分发的网卡驱动技术。



大页内存

一般64位机种都会配置1G的大页内存

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

热点函数优化

如：

```c
static inline uint16_t
nbl_virtqueue_fetch_flags_packed(const struct nbl_vring_packed_desc *dp)
{
        uint16_t flags;

        /* x86 prefers to using rte_smp_wmb over __atomic_store_n
         * as it reports a better perf(~1.5%), which comes from
         * the saved branch by the compiler.
         * The if and else branch are identical with the smp
         * and io barriers both defined as compiler barriers on x86.
         */
#ifdef RTE_ARCH_X86_64
        flags = dp->flags;
        rte_io_rmb();
#else
        flags = __atomic_load_n(&dp->flags, __ATOMIC_ACQUIRE);
#endif

        return flags;
}
```

OVS优化专题

参考链接