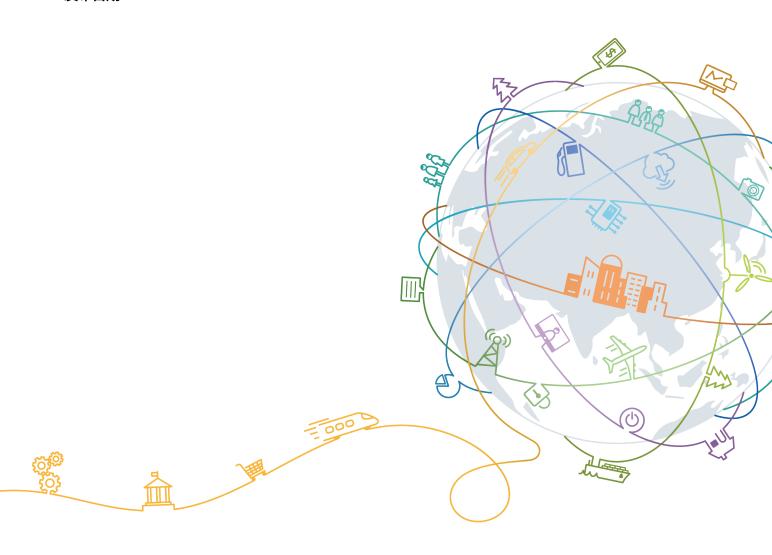
鲲鹏性能优化十板斧

文档版本 04

发布日期 2020-07-17





版权所有 © 华为技术有限公司 2020。 保留一切权利。

非经本公司书面许可,任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部,并不得以任何形式传播。

商标声明



nuawe和其他华为商标均为华为技术有限公司的商标。 本文档提及的其他所有商标或注册商标,由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束,本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定,华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因,本文档内容会不定期进行更新。除非另有约定,本文档仅作为使用指导,本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目录

1 简介	1
1.1 鲲鹏处理器 NUMA 简介	1
1.2 性能调优五步法	2
2 CPU 与内存子系统性能调优	4
- C C C C C C C C C C C C C C C C C C C	
2.2 常用性能监测工具	
13/15年15年15年15年15年15年15年15日 - 13/15年15日 - 13/15年	
2.2.2 Perf 工具	
2.2.3 numactl 工具	
2.3 优化方法	
2.3.2 定时器机制调整,减少不必要的时钟中断	12
2.3.3 调整线程并发数	
2.3.4 NUMA 优化,减少跨 NUMA 访问内存	13
2.3.5 调整内存页的大小	14
3 网络子系统性能调优	16
3.1 调优简介	16
3.2 常用性能监测工具	17
3.2.1 ethtool 工具	17
3.2.2 strace 工具	19
3.3 优化方法	19
3.3.1 PCIE Max Payload Size 大小配置	20
3.3.2 网络 NUMA 绑核	20
3.3.3 中断聚合参数调整	23
3.3.4 开启 TSO	23
3.3.5 使用 epoll 代替 select	24
3.3.6 单队列网卡中断散列	25
3.3.7 TCP checksum 优化	26
4 磁盘 IO 子系统性能调优	27
4.1 调优简介	27
4.2 常用性能监测工具	28
4.2.1 iostat 工具	28

鲲鹏性能优化十板斧	目录
4.2.2 blktrace 工具	30
4.3 优化方法	
4.3.1 调整脏数据刷新策略,减小磁盘的 IO 压力	31
4.3.2 调整磁盘文件预读参数	
4.3.3 优化磁盘 IO 调度方式	33
4.3.4 文件系统参数优化	33
4.3.5 使用异步文件操作 libaio 提升系统性能	34
5 应用程序调优	35
5.1 调优简介	35
5.2 优化方法	
5.2.1 优化编译选项,提升程序性能	35
5.2.2 文件缓冲机制选择	36
5.2.3 执行结果缓存	36
5.2.4 减少内存拷贝	37
5.2.5 锁优化	
5.2.6 使用 jemalloc 优化内存分配	39
5.2.7 Cacheline 优化	
5.2.8 NEON 指令加速	41
6 JVM 性能调优	44
6.1 调优简介	44
6.2 常用性能监测工具	45
6.2.1 jstat 工具	45
6.2.2 jmap 工具	47
6.3 JVM 原理及配置建议	48
6.3.1 尽量使用高版本 JDK	48
6.3.2 设置 JVM 堆空间大小	49
5.3.3 选择合适的垃圾回收器	
5.4 优化调优方法	53
5.4.1 GC 分析优化	
6.4.2 JVM 线程优化	
6.4.3 调整线程堆栈大小	
A libaio 实现参考	57
B 进入 BIOS 界面	59
C NEON lib 库应用加速	61
D 常用操作系统内存参数说明	62

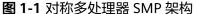
1 简介

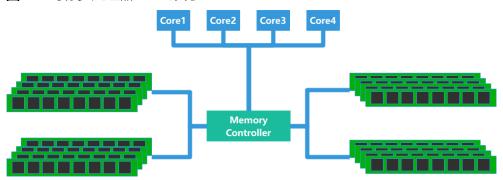
- 1.1 鲲鹏处理器NUMA简介
- 1.2 性能调优五步法

1.1 鲲鹏处理器 NUMA 简介

随着现代社会信息化、智能化的飞速发展,越来越多的设备接入互联网、物联网、车 联网,从而催生了庞大的计算需求。但是功耗墙问题以功耗和冷却两大限制极大的影响了单核算力的发展。为了满足智能世界快速增长的算力需求,多核架构成为最重要 的演进方向。

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<mark>图1-1</mark>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。



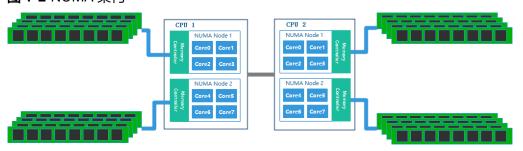


鲲鹏处理器支持NUMA(Non-uniform memory access, 非统一内存访问)架构,能够很好的解决SMP技术对CPU核数的制约。NUMA架构将多个核结成一个节点(Node),每一个节点相当于是一个对称多处理机(SMP),一块CPU的节点之间通过On-chip Network通讯,不同的CPU之间采用Hydra Interface实现高带宽低时延的片间通讯,如图1-2所示。在NUMA架构下,整个内存空间在物理上是分布式的,所有

鲲鹏性能优化十板斧 1 简介

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,现在的操作系统也提供了丰富的工具和接口,帮助我们完成就近访问内存的优化和配置。所以,使用鲲鹏处理器所实现的计算机系统,通过适当的性能调优,既能够达成很好的性能,又能够解决SMP架构下的总线瓶颈问题,提供更强的多核扩展能力,以及更好更灵活的计算能力。

图 1-2 NUMA 架构



1.2 性能调优五步法

性能优化通常可以通过如表1-1五个步骤完成。

表 1-1 性能优化的通用步骤

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。

鲲鹏性能优化十板斧 1 简介

序号	步骤	说明
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。

在性能调优经验比较少或者对系统的软硬件并不是非常了解时,可以参考使用五步法 的模式逐步展开性能调优的工作。对于有丰富调优经验的工程师,或者对系统的性能 瓶颈已经有深入洞察的专家,也可以采用其他方法或过程展开优化工作。

本文档描述鲲鹏芯片常用的性能优化方法和分析工具。文档分别从CPU与内存子系统,网络子系统,磁盘IO子系统和应用程序优化4个方面阐述了常用的性能优化方法和分析工具,主要适用于执行性能优化的研发工程师和技术支持工程师。

2 CPU 与内存子系统性能调优

- 2.1 调优简介
- 2.2 常用性能监测工具
- 2.3 优化方法

2.1 调优简介

调优思路

性能优化的思路如下:

- 如果CPU的利用率不高,说明资源没有充分利用,可以通过工具(如strace)查看 应用程序阻塞在哪里,一般为磁盘,网络或应用程序自己的业务逻辑有休眠或信 号等待,这些优化措施在其它章节描述。
- 如果CPU利用率高,可以选择更好的硬件,优化硬件的配置参数来适配业务场景,或者通过优化软件来降低CPU占用率。

根据CPU的能力配置合适的内存条,建议内存满通道配置,发挥内存最大带宽:一颗 鲲鹏920处理器的内存通道数为8,两颗鲲鹏920处理器的内存通道数为16;建议选择 高频率的内存条,提升内存带宽: 鲲鹏920在1DPC配置时,支持的内存最高频率为 2933MHz。

主要优化参数

优化项	优化项简介	默认值	生效范 围	鲲鹏 916	鲲鹏 920
优化应用程 序的NUMA 配置	在NUMA架构下,CPU core访问临近的内存时访 问延迟更低。将应用程序 绑在一个NUMA节点,可 减少因访问远端内存带来 的性能下降。	默认不绑定 核	立即生 效	yes	yes

优化项	优化项简介	默认值	生效范围	鲲鹏 916	鲲鹏 920
修改CPU预 取开关	内存预取在数据集中场景下可以提前将要访问的数据读到CPU cache 中,提升性能;若数据不集中,导致预取命中率低,则浪费内存带宽。	on	重启生 效	no	yes
调整定时器 机制	nohz机制可减少不必要的 时钟中断,减少CPU调度 开销。	不同OS默认 配置不同 Euler: nohz=off	重启生效	yes	yes
调整内存的 页大小为 64K	内存的页大小越大,TLB 中每行管理的内存越多, TLB命中率就越高,从而 减少内存访问次数。	不同OS默认 配置不同: 4KB或64K	重新编 译内 核、更 新内核 后生效	yes	yes
优化应用程 序的线程并 发数	适当调整应用的线程并发数,使得充分利用多核能力和资源争抢之间达到平 衡。	由应用本身 决定	立即生 效或重 启生效 (由决 用决 定)	yes	yes

2.2 常用性能监测工具

2.2.1 top 工具

介绍

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:

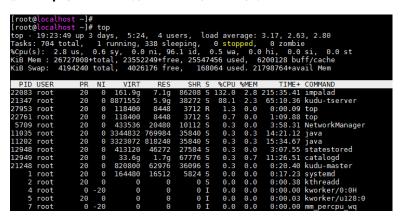
命令	说明
top	查看系统整体的CPU、内存资源消耗。
top执行后输入1	查看每个CPU core资源使用情况。
top执行后输入F,并选 择P选项	查看线程执行过程中是否调度到其它CPU core。
top -p \$PID -H	查看某个进程内所有线程的CPU资源占用。

安装方式

系统自带,无需安装。

使用方法

步骤1 使用top命令统计整体CPU、内存资源消耗。



- 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。

步骤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。

```
top - 20:12:52 up 3 days, 6:13, 4 users, load average: 1.20, 2.27, 2.61

Tasks: 704 total, 1 running, 339 sleeping, 0 stopped, 0 zombie

%Cpu0 : 0.0 us, 0.6 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu1 : 0.0 us, 0.6 sy, 0.0 ni, 99.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu2 : 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu3 : 0.0 us, 0.6 sy, 0.0 ni, 99.4 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu4 : 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu5 : 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu6 : 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
```

步骤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。

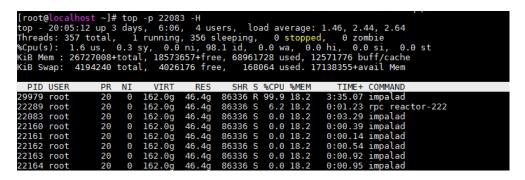
```
Fields Management for window FiDeM, whose current sort field is %CPU
Navigate with Up/Dn, Right selects for move then <Enter> or Left commits,
'd' or <Space* toggles display, 's' sets sort. Use 'q' or <Esc> to end!

*PID = Process Id

*USER = Effective User Name
PR = Priority
*NI = Nice Value
*VIRT = Virtual Image (KiB)
*RES = Resident Size (KiB)
*SES = Resident Size (KiB)
*S = Process Status
*VCPU = CPU Usage
*WCPU = CPU Usage
*WMEM = Memory Usage (RES)
*TIME+ CPU Time, bundredths
*COMMAND = Command Name/Line
PPID = Parent Process pid
UID = Effective User Id
*RUSER = Real User Name
SUID = Saved User Name
SUID = Saved User Name
GID = Group Id
*GROUP = Group Name
GID = Group Id
*GROUP = Frocess Group Id
*TTY = Controlling Tty
*PIGID = Tty Process for Id
*SID = Session Id
*TTY = CNUTAGE
*PIN = Number of Threads
*PIN = CPU Time
*IN = CPU
```

步骤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。



----结束

2.2.2 Perf 工具

介绍

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。

命令	说明
perf sched latency sort max	查看上一步记录的结果,以调度延迟排序。

安装方式

以CentOS为例,使用如下命令安装:

yum -y install perf

使用方法

步骤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。

```
Samples: 14K of event 'cycles:ppp', Event count (approx.): 4104207681

Overhead Shared Object Symbol

6.41% [kernel] [k] arch_cpu_idle

2.38% [kernel] [k] finish_task_switch

2.07% perf [.] _symbols__insert

1.91% [kernel] [k] module_get_kallsym

1.53% libpython2.7.so.1.0 [.] PyEval_EvalFrameEx

1.50% libc-2.17.so [.] strlen

1.48% libc-2.17.so [.] strchr

1.42% perf [.] rb_next

1.33% [kernel] [k] __ll_sc_atomic_sub_return_release

1.23% [kernel] [k] __ll_sc_atomic_sub_return_release

1.23% [kernel] [k] __ll_sc_atloc

1.17% [kernel] [k] __ll_sc_atloc

1.17% [kernel] [k] __ll_sc_atloc

1.11% [kernel] [k] __ll_sc_atloc

1.12% [kernel] [k] __ll_sc_atloc

1.13% [kernel] [k] __ll_sc_atloc

1.14% [kernel] [k] __ll_sc_atloc

1.15% [kernel] [k] __ll_sc_atloc

1.16% [k] __ll_sc_atloc

1.17% [kernel] [k] __ll_sc_atloc

1.11% [kernel] [k] __ll_sc_atloc

1.12% [kernel] [k] __ll_sc_atloc

1.13% [kernel] [k] __ll_sc_atloc

1.14% [kernel] [k] __ll_sc_atloc

1.15% [kernel] [k] __ll_sc_atloc

1.16% [k] __ll_sc_atloc

1.17% [kernel] [k] __ll_sc_atloc

1.18% [kernel] [k] __ll_sc_atloc

1.19% [kernel] [k] __ll_sc_atloc

1.11% [ke
```

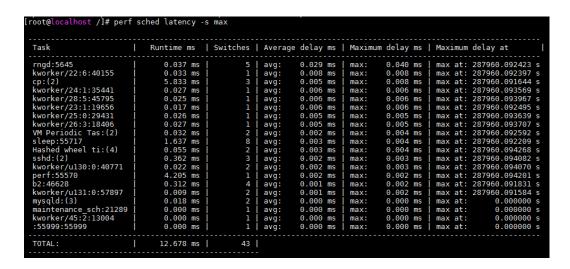
步骤2 收集一段时间内的线程调用。

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) ]
```

步骤3 解析收集到的线程调度信息。

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。



----结束

2.2.3 numactl 工具

介绍

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。

安装方式

以CentOS为例,使用如下命令安装:

yum -y install numactl numastat

使用方法

步骤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 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
```

步骤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
```

步骤3 通过numastat查看当前NUMA节点的内存访问命中率。

[root@localhost to	est]# numastat			
	node0	node1	node2	node3
numa_hit	68641597	59333134	50159172	55076006
numa_miss	1	1005288	188567	2132
numa_foreign	1171587	2133	Θ	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。

----结束

2.3 优化方法

2.3.1 修改 CPU 的预取开关

原理

局部性原理分为时间局部性原理和空间局部性原理:

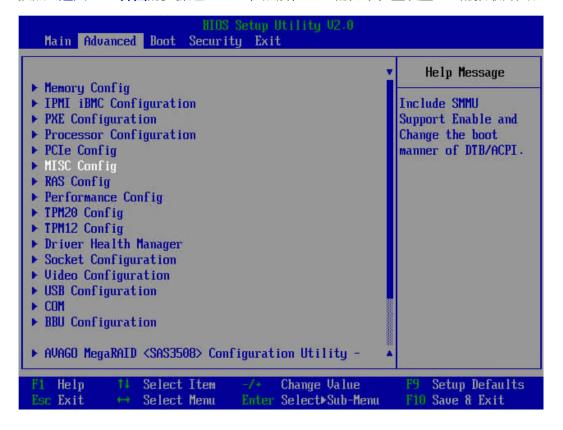
- 时间局部性原理(temporal locality):如果某个数据项被访问,那么在不久的将来它可能再次被访问。
- 空间局部性原理(spatial locality): 如果某个数据项被访问,那么与其地址相邻的数据项可能很快也会被访问。

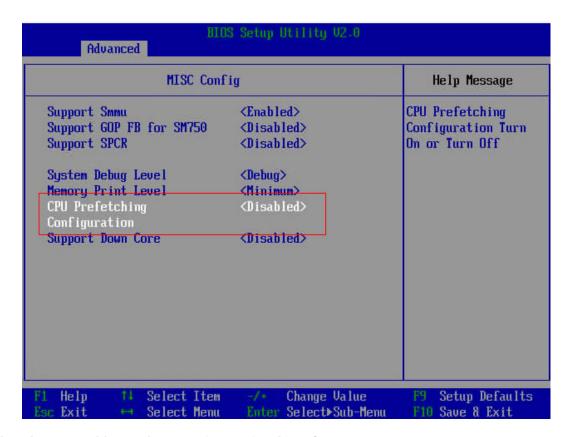
CPU将内存中的数据读到CPU的高速缓冲Cache时,会根据局部性原理,除了读取本次要访问的数据,还会预取本次数据的周边数据到Cache里面,如果预取的数据是下次要访问的数据,那么性能会提升,如果预取的数据不是下次要取的数据,那么会浪费内存带宽。

对于数据比较集中的场景,预取的命中率高,适合打开CPU预取,反之需要关闭CPU 预取。目前发现speccpu和X265软件场景适合打开CPU预取,STREAM测试工具、 Nginx和数据库场景需要关闭CPU预取。

修改方式

按照**B进入BIOS界面**的步骤进入BIOS,然后在BIOS的如下位置设置CPU的预取开关。





2.3.2 定时器机制调整,减少不必要的时钟中断

原理

在Linux内核2.6.17版本之前,Linux内核为每个CPU设置一个周期性的时钟中断, Linux内核利用这个中断处理一些定时任务,如线程调度等。这样导致就算CPU不需要 定时器的时候,也会有很多时钟中断,导致资源的浪费。Linux 内核2.6.17版本引入了 nohz机制,实际就是让时钟中断的时间可编程,减少不必要的时钟中断。

修改方式

执行**cat /proc/cmdline**查看Linux 内核的启动参数,如果有nohz=off关键字,说明 nohz机制被关闭,需要打开。修改方法如下:

□ 说明

修改前后,可以通过如下命令观察timer_tick的调度次数,其中\$PID为要观察的进程ID,可以选择CPU占用高的进程进行观察:

perf sched record -- sleep 1 -p \$PID

perf sched latency -s max

输出信息中有如下信息,其中591字段表示统计时间内的调度次数,数字变小说明修改生效。 timer_tick:(97) | 7.364 ms | 591 | avg: 0.012 ms | max: 1.268 ms

步骤1 在 "/boot" 目录下通过find -name grub.cfg找到启动参数的配置文件。

步骤2 在配置文件中将nohz=off去掉。

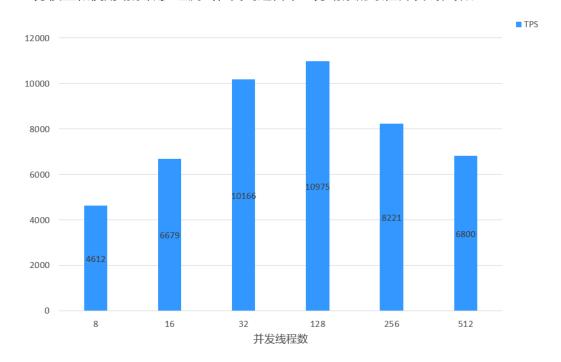
步骤3 重启服务器。

----结束

2.3.3 调整线程并发数

原理

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。



修改方式

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。

2.3.4 NUMA 优化,减少跨 NUMA 访问内存

原理

通过**1.1 鲲鹏处理器NUMA简介**章节可以看到不同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。

修改方式

网络可以通过如下方式绑定运行的CPU core,其中\$cpuMask是16进制的数,最右边的bit表示core0;\$irq为网卡队列中断号。

echo \$cpuMask > /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。

2.3.5 调整内存页的大小

原理

TLB(Translation lookaside buffer)为页表(存放虚拟地址的页地址和物理地址的页地址的映射关系)在CPU内部的高速缓存。TLB的命中率越高,页表查询性能就越好。

TLB的一行为一个页的映射关系,也就是管理了一个页大小的内存:

TLB管理的内存大小 = TLB行数 x 内存的页大小

同一个CPU的TLB行数固定,因此内存页越大,管理的内存越大,相同业务场景下的 TLB命中率就越高。

□ 说明

修改前后可以通过如下命令观察TLB的命中率(\$PID为进程ID):

perf stat -p \$PID -d -d -d

输出结果包含如下信息,其中和0.59%分别表示数据的miss率和指令的miss率。

 1,090,788,717
 dTLB-loads
 # 520.592 M/sec

 13,213,603
 dTLB-load-misses
 # 1.21% of all dTLB cache hits

 669,485,765
 iTLB-loads
 # 319.520 M/sec

 3,979,246
 iTLB-load-misses
 # 0.59% of all iTLB cache hits

修改方式

- 调整内核内存页大小
- 设置内存大页
- 设置透明大页

设置内核内存页大小

修改linux内核的内存页大小,需要在修改内核编译选项后重新编译内核(详情可参考内核源码编译安装),简要步骤如下所示:

- 执行make menuconfig
- 选择PAGESIZE大小为64K
 Kernel Features-->Page size(64KB)
- 编译和安装内核

设置内存大页

在内核启动阶段设置

可参考kernel-parameters设置如下内核启动参数:

参数	说明
hugepages	定义启动时内核中配置的内存大页的数量。
hugepagesz	定义启动时在内核中配置的内存大页的大小。
default_hugepagesz	定义启动时在内核中配置的内存大页的默认大小。

• 在运行时设置

设置node中2 MB大页的数量为20。

echo 20 > /sys/devices/system/node/node0/hugepages/ hugepages-2048kB/nr_hugepages

设置透明大页

透明大页是以上内存大页的升级版,使用透明大页,内核会自动为进程分配大页,因此无需手动保留大页。

设置/sys/kernel/mm/transparent_hugepage/enabled的值为:

参数	说明
always	在整个系统范围内启用THP,如果有进程大量使用连续的虚 拟内存,则内核会尝试将大页分配给该进程。
madvise	内核仅将大页分配给madvise()系统调用指定的单个进程的内存区域。
never	禁用透明大页。

例如关闭透明大页的设置方法:

echo never > /sys/kernel/mm/transparent_hugepage/enabled

操作系统还有很多内存参数可以调整,具体参考D常用操作系统内存参数说明。

鲲鹏性能优化十板斧 3 网络子系统性能调优

3 网络子系统性能调优

- 3.1 调优简介
- 3.2 常用性能监测工具
- 3.3 优化方法

3.1 调优简介

调优思路

本章主要是围绕优化网卡性能和利用网卡的能力分担CPU的压力来提升性能。在高并发的业务场景下,推荐使用两块网卡,减少跨片内存访问的次数。即将两块网卡分别绑定在服务器的不同CPU上,每个CPU只处理对应的网卡数据。高并发场景还可以为网卡选择x16的PCIE卡。

□ 说明

高并发场景是指:在同时或极短时间内,有大量的请求到达服务端,每个请求都需要服务端耗费 资源进行处理,并做出相应的反馈。

主要优化参数

优化项	优化项简介	默认值	生效范 围	鲲鹏 916	鲲鹏 920
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	重启生 效	Y	Y
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	立即生效	Υ	Υ
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	立即生 效	Υ	Y

优化项	优化项简介	默认值	生效范 围	鲲鹏 916	鲲鹏 920
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	立即生效	Υ	Υ
开启TCP分段 Offload(卸载)	将TCP的分片处理 交给网卡处理	关闭	立即生 效	Υ	Υ

3.2 常用性能监测工具

3.2.1 ethtool 工具

介绍

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。

安装方式

以CentOS为例,使用如下命令安装:

yum -y install ethtool net-tools

使用方式

命令格式: ethtool [参数]

常用参数如下:

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。
-с	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。

输出格式:

ethtool -k eth0

Features for eth0: rx-checksumming: on

鲲鹏性能优化十板斧 3 网络子系统性能调优

> tx-checksumming: on scatter-gather: on

tcp-segmentation-offload: on

ethtool -l eth0

Channel parameters for eth0:

Pre-set maximums:

Current hardware settings:

Combined: 8

ethtool -c eth0

Coalesce parameters for eth0: Adaptive RX: off TX: off

rx-usecs: 30 rx-frames: 50

tx-usecs: 30 tx-frames: 1

参数含义如下:

参数	说明
rx- checksumming	接收包校验和。
tx- checksumming	发送包校验和。
scatter-gather	分散-聚集功能,是网卡支持TSO的必要条件之一。
tcp- segmentation- offload	简称为TSO,利用网卡对TCP数据包分片。
Combined	网卡队列数。
adaptive-rx	接收队列的动态聚合执行开关。
adaptive-tx	发送队列的动态聚合执行开关。
tx-usecs	产生一个中断之前至少有一个数据包被发送之后的微秒数。
tx-frames	产生中断之前发送的数据包数量。
rx-usecs	产生一个中断之前至少有一个数据包被接收之后的微秒数。
rx-frames	产生中断之前接收的数据包数量。

3.2.2 strace 工具

介绍

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。

安装方式

以CentOS为例,使用如下命令安装:

yum -y install strace

使用方式

命令格式: strace [参数]

常用参数如下:

-Т	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。
-р	跟踪指定的线程ID。

输出格式:

18:25:47.902439 epoll_pwait(716, [{EPOLLIN, {u32=1052576880, u64=281463144385648}}, {EPOLLIN, {u32=1052693569, u64=281463144502337}}, {EPOLLOUT, {u32=1052638657, u64=28146314447425}}, {EPOLLIN|EPOLLOUT|EPOLLRDHUP, {u32=1052673241, u64=281463144482009}}, {EPOLLIN|EPOLLOUT|EPOLLERP|EPOLLHUP|EPOLLRDHUP, {u32=1052636016, u64=281463144444784}}], 512, 1, NULL, 8) = 5 <0.000038>

参数含义如下:

参数	说明
18:25:47.902439	为系统调用发生的时间。
epoll_pwait	为系统调用的函数名。
(716…)	括号内的值为函数参数。
=5	为系统调用的返回值。
<0.000038>	为系统调用的执行时间。

3.3 优化方法

鲲鹏性能优化十板斧 3 网络子系统性能调优

3.3.1 PCIE Max Payload Size 大小配置

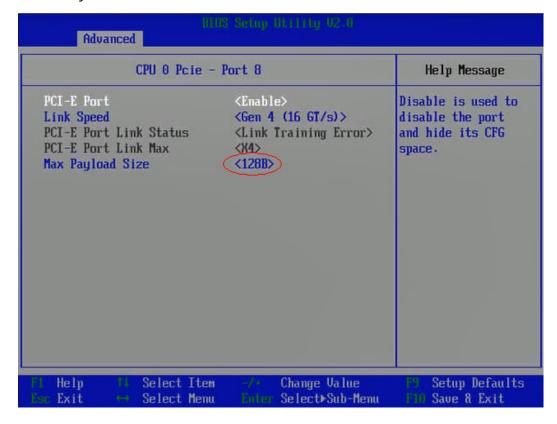
原理

网卡自带的内存和CPU使用的内存进行数据传递时,是通过PCIE总线进行数据搬运的。Max Payload Size为每次传输数据的最大单位(以字节为单位),它的大小与PCIE链路的传送效率成正比,该参数越大,PCIE链路带宽的利用率越高。

Transaction Layer Packet		
Header	Data Payload	ECRC

修改方式

按照**B 进入BIOS界面**的步骤进入BIOS,选择"Advanced > Max Payload Size",将 "Max Payload Size"的值设置为"512B"。



3.3.2 网络 NUMA 绑核

原理

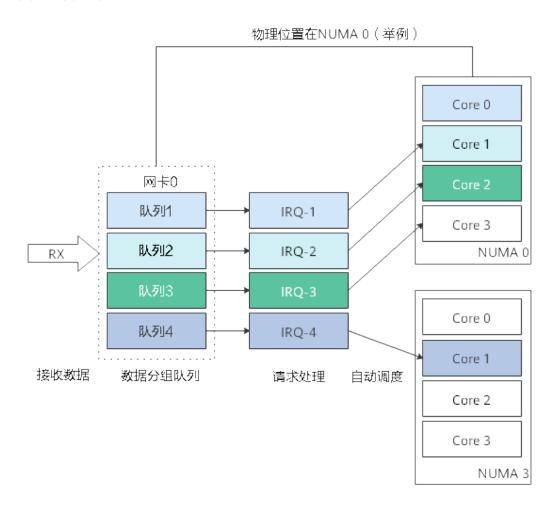
当网卡收到大量请求时,会产生大量的中断,通知内核有新的数据包,然后内核调用中断处理程序响应,把数据包从网卡拷贝到内存。当网卡只存在一个队列时,同一时间数据包的拷贝只能由某一个core处理,无法发挥多核优势,因此引入了网卡多队列机制,这样同一时间不同core可以分别从不同网卡队列中取数据包。

在网卡开启多队列时,操作系统通过Irqbalance服务来确定网卡队列中的网络数据包交由哪个CPU core处理,但是当处理中断的CPU core和网卡不在一个NUMA时,会触发

鲲鹏性能优化十板斧 3 网络子系统性能调优

跨NUMA访问内存。因此,我们可以将处理网卡中断的CPU core设置在网卡所在的 NUMA上,从而减少跨NUMA的内存访问所带来的额外开销,提升网络处理性能。

图 3-1 自动绑定: 中断绑定随机,出现跨 NUMA 访问内存



鲲鹏性能优化十板斧 3 网络子系统性能调优

物理位置在NUMA 0(举例) Core 0 Core 1 网卡() Core 2 队列1 IRQ-1 Core 3 队列2 IRQ-2 NUMA 0 RX 队列3 IRQ-3 Core 0 队列4 IRQ-4 Core 1 接收数据 数据分组队列 请求处理 绑定到NUMA Core 2 Core 3 NUMA 3

图 3-2 NUMA 绑定:中断绑定到指定核,避免跨 NUMA 访问内存

修改方式

步骤1 停止irqbalance。

systemctl stop irqbalance.service

systemctl disable irqbalance.service

步骤2 设置网卡队列个数为CPU的核数。

ethtool -L ethx combined 48

步骤3 查询中断号。

cat /proc/interrupts | grep \$eth | awk -F ':' '{print \$1}'

步骤4 根据中断号,将每个中断分别绑定在一个核上,其中cpuMask是16进制的数,最右边的bit表示core0。

echo \$cpuMask > /proc/irq/\$irq/smp_affinity_list

----结束

鲲鹏性能优化十板斧 3 网络子系统性能调优

3.3.3 中断聚合参数调整

原理

中断聚合特性允许网卡收到报文之后不立即产生中断,而是等待一小段时间有更多的报文到达之后再产生中断,这样就能让CPU一次中断处理多个报文,减少开销。

修改方式

使用ethtool -C \$eth方法调整中断聚合参数。其中参数"\$eth"为待调整配置的网卡设备名称,如eth0,eth1等。

ethtool -C eth3 adaptive-rx off adaptive-tx off rx-usecs N rx-frames N tx-usecs N tx-frames N

为了确保使用静态值,需禁用自适应调节,关闭Adaptive RX和Adaptive TX。

- rx-usecs:设置接收中断延时的时间。
- tx-usecs:设置发送中断延时的时间。
- rx-frames: 产生中断之前接收的数据包数量。
- tx-frames:产生中断之前发送的数据包数量。这4个参数设置N的数值越大,中断越少。

□ 说明

增大聚合度,单个数据包的延时会有微秒级别的增加。

3.3.4 开启 TSO

原理

当一个系统需要通过网络发送一大段数据时,计算机需要将这段数据拆分为多个长度较短的数据,以便这些数据能够通过网络中所有的网络设备,这个过程被称作分段。 TCP分段卸载将TCP的分片运算(如将要发送的1M字节的数据拆分为MTU大小的包) 交给网卡处理,无需协议栈参与,从而降低CPU的计算量和中断频率。

修改方式

使用ethtool工具打开网卡和驱动对TSO(TCP Segmentation Offload)的支持。如下命令中的参数"\$eth"为待调整配置的网卡设备名称,如eth0,eth1等。

ethtool -K \$eth tso on

□ 说明

要使用TSO功能,物理网卡需同时支持TCP校验计算和分散-聚集 (Scatter Gather) 功能。 查看网卡是否支持TSO:

ethtool -k \$eth

rx-checksumming: on tx-checksumming: on scatter-gather: on

tcp-segmentation-offload: on

鲲鹏性能优化十板斧 3 网络子系统性能调优

3.3.5 使用 epoll 代替 select

原理

epoll机制是Linux内核中的一种可扩展IO事件处理机制,可被用于代替POSIX select系统调用,在高并发场景下获得较好的性能提升。

select有如下缺点:

- 内核默认最多支持1024个文件句柄
- select采用轮询的方式扫描文件句柄,性能差

epoll改进后的机制:

- 没有最大并发连接的限制,能打开的文件句柄上限远大于1024,方便在一个线程 里面处理更多请求
- 采用事件通知的方式,减少了轮询的开销

修改方式

使用epoll函数代替select,epoll函数有:epoll_create,epoll_ctl和epoll_wait。 Linux-2.6.19又引入了可以屏蔽指定信号的epoll_wait: epoll_pwait。

函数简介:

步骤1 创建一个epoll句柄。

int epoll_create(int size);

其中size表示监听的文件句柄的最大个数。

步骤2 注册要监听的事件类型。

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

参数说明如下:

- epfd: epoll_create返回的文件句柄。
- op: 要进行的操作,有EPOLL_CTL_ADD、EPOLL_CTL_MOD、EPOLL_CTL_DEL 等。
- fd: 要操作的文件句柄。
- event:事件描述,常用的事件有:
 - EPOLLIN: 文件描述符上有可读数据。
 - EPOLLOUT: 文件描述符上可以写数据。
 - EPOLLERR: 表示对应的文件描述符发生错误。

步骤3 等待事件的产生。

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);

int epoll_pwait(int epfd, struct epoll_event *events,

int maxevents, int timeout,

const sigset_t *sigmask);

鲲鹏性能优化十板斧 3 网络子系统性能调优

参数说明如下:

• epfd: epoll_create返回的文件句柄。

• events:返回待处理事件的数组。

maxevents: events数组长度。

• timeout: 超时时间,单位为毫秒。

● sigmask: 屏蔽的信号。

----结束

□ 说明

在网络高并发场景下,select调用可使用epoll进行替换。

使用如下命令确认是否调用epoll函数:

查看某个进程的系统调用信息

strace -p \$TID -T -tt

18:25:47.902439 epoll_pwait(716, [{EPOLLIN, {u32=1052576880, u64=281463144385648}}, {EPOLLIN, {u32=1052693569, u64=281463144502337}}, {EPOLLOUT, {u32=1052638657, u64=281463144447425}}, {EPOLLIN|EPOLLOUT| EPOLLRDHUP, {u32=1052673241, u64=281463144482009}}, {EPOLLIN|EPOLLOUT| EPOLLERR|EPOLLHUP|EPOLLRDHUP, {u32=1052636016, u64=28146314444784}}], 512, 1. NULL. 8) = 5 < 0.000038>

3.3.6 单队列网卡中断散列

原理

RPS 全称是 Receive Packet Steering, 在2.6.35进入Linux内核。RPS采用软件模拟的方式,实现了多队列网卡所提供的功能,分散了在多CPU系统上数据接收时的负载, 把软中断分到各个CPU处理,而不需要硬件支持,大大提高了网络性能。

对单队列网卡可以使用RPS将中断分散到各个core处理,避免软中断集中到一个core导致该core软中断过高形成性能瓶颈。

修改方式

通过直接修改网卡队列参数设置RPS,能够立即生效,无需重启服务器。

修改前:

/sys/class/net/eth0/queues/rx-0/rps_cpus 0 /sys/class/net/eth0/queues/rx-0/rps_flow_cnt 0 /proc/sys/net/core/rps_sock_flow_entries 0

修改方法:

#echo ff > /sys/class/net/eth0/queues/rx-0/rps_cpus
echo 4096 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt
echo 32768 > /proc/sys/net/core/rps_sock_flow_entries

□□说明

这里以单个网卡为例说明,ff对应的是core 0-7,意思是将软中断散列到0-7个core上。如果是多个网卡,对应的ff需要修改。将并发活动连接的最大预期数目设置为32768,是因为这个是linux 官方内核推荐值。

鲲鹏性能优化十板斧 3 网络子系统性能调优

3.3.7 TCP checksum 优化

原理

TCP校验和(checksum)是一个端到端的校验和,由发送端计算,然后由接收端验证。其目的是为了发现TCP首部和数据在发送端到接收端之间发生的任何改动。如果接收方检测到校验和有差错,则TCP段会被直接丢弃。

通常TCP checksum是由内核网络驱动来实现的,而且在5.7版本之前的内核是使用普通的算法实现。

部分网卡驱动是支持TCP校验和功能的,可以通过ethtool -k **网卡名称 | grep** checksumming查看是否支持。

如果网卡驱动不支持,可以通过修改内核合入checksum优化算法来提升性能。

修改方式

5.7之前的内核可以通过合入**checksum优化算法补丁**提升性能,修改后需要重新编译内核(详情可参考**内核源码编译安装**)。

4 磁盘 IO 子系统性能调优

- 4.1 调优简介
- 4.2 常用性能监测工具
- 4.3 优化方法

4.1 调优简介

调优思路

CPU的Cache、内存和磁盘之间的访问速度差异很大,当CPU计算所需要的数据并没有及时加载到内存或Cache中时,CPU将会浪费很多时间等待磁盘的读取。计算机系统通过cache、RAM、固态盘、磁盘等多级存储结构,并配合多种调度算法,来消除或缓解这种速度不对等的影响。但是缓存空间总是有限的,我们可以利用局部性原理,尽可能的将热点数据提前从磁盘中读取出来,降低CPU等待磁盘的时间浪费。因此我们的部分优化手段其实是围绕着如何更充分的利用Cache获得更好的IO性能。

另外,本章节也会介绍从文件系统层面的优化手段。

主要优化参数

优化项	优化项简介	默认值	生效范围	鲲 鹏 91 6	鲲 鹏 92 0
脏数据缓存到期时 间	调整脏数据缓存到期时间, 分散磁盘的压力。	3000(单 位为1/100 秒)	立即生效	Υ	Υ
脏页面占用总内存 最大的比例	调整脏页面占用总内存最大 的比例(以memfree +Cached-Mapped为基 准),增加PageCache命中 率。	10%	立即 生效	Υ	Υ

优化项	优化项简介	默认值	生效 范围	鲲 鹏 91 6	鲲 鹏 92 0
脏页面缓存占用总 内存最大的比例	调整脏页面占用总存最大的 比例,避免磁盘写操作变为 O_DIRECT同步,导致缓冲 机制失效。	40%	立即 生效	Y	Y
调整磁盘文件预读 参数	根据局性原理,在读取磁盘 数据时,额外地多读一定量 的数据缓存到内存。	128KB	立即 生效	Y	Υ
磁盘IO调度方式	根据业务处理数据的特点, 选择合适的IO调度器。	cfq	立即 生效	Υ	Υ
文件系统	选用性能更好的文件系统以 及文件系统相关的选项。	N/A	立即 生效	Y	Υ

4.2 常用性能监测工具

4.2.1 iostat 工具

介绍

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。

安装方式

iostat一般会随系统安装。如果没有,以CentOS为例,可以使用以下命令安装:

yum -y install sysstat

使用方式

命令格式:直接使用命令+参数,例如

iostat -d -k -x 1 100

常用参数如下:

-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。

-р	显示磁盘单个的情况。
-t	显示时间戳。
-X	显示详细信息。

我们也可以在最后添加统计周期和统计时长,比如上面的样例中,是要求以1秒为周期统计,总共统计100s。

输出格式:

Device:	rrqm/s	wrqm	n/s r/s	s w/s	rkB/	s wkB,	s avgrq-	sz avgo	qu-sz a	wait s	vctm %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:

每秒合并放入请求队列的读操作数。
每秒合并放入请求队列的写操作数。
每秒磁盘实际完成的读I/O设备次数。
每秒磁盘实际完成的写I/O设备次数。
每秒从磁盘读取KB数。
每秒写入磁盘的KB数。
平均请求数据大小,单为为扇区(512B)。
平均I/O队列长度(操作请求数)。
平均每次设备I/O操作的等待时间(毫秒)。
平均每次设备I/O操作的响应时间(毫秒)。
用于I/O操作时间的百分比,即使用率。

重要参数详解:

- 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。

4.2.2 blktrace 工具

介绍

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。

安装方式

以CentOS为例,可以使用以下命令安装:

yum -y install blktrace

使用方式

步骤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。

步骤2 使用blkparse命令解析blktrace生成的数据,如:

blkparse -i sda -d blkparse.out

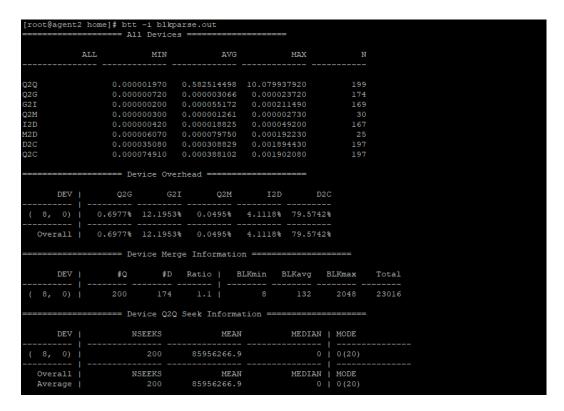
"-i"后接的参数指定输入文件名,由于blktrace输出的文件名默认都是 *device*.blktrace.*cpu*格式,所以只需输入前面的"device"即可;"-d"后接的参数指定二进制输出文件。

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。

步骤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。



- 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。

----结束

4.3 优化方法

4.3.1 调整脏数据刷新策略,减小磁盘的 IO 压力

原理

PageCache中需要回写到磁盘的数据为脏数据。在应用程序通知系统保存脏数据时,应用可以选择直接将数据写入磁盘(O_DIRECT),或者先写到PageCache(非

O_DIRECT模式)。非O_DIRECT模式,对于缓存在PageCache中的数据的操作,都在内存中进行,减少了对磁盘的操作。

修改方式

系统中提供了以下参数来调整策略:

1. /proc/sys/vm/dirty_expire_centisecs此参数用于表示脏数据在缓存中允许保留的时长,即时间到后需要被写入到磁盘中。此参数的默认值为30s(3000 个1/100秒)。如果业务的数据是连续性的写,可以适当调小此参数,这样可以避免IO集中,导致突发的IO等待。可以通过echo命令修改:

echo 2000 > /proc/sys/vm/dirty_expire_centisecs

2. /proc/sys/vm/dirty_background_ratio 脏页面占用总内存最大的比例(以memfree+Cached-Mapped为基准),超过这个值,pdflush线程会刷新脏页面到磁盘。增加这个值,系统会分配更多的内存用于写缓冲,因而可以提升写磁盘性能。但对于磁盘写入操作为主的的业务,可以调小这个值,避免数据积压太多最后成为瓶颈,可以结合业务并通过观察await的时间波动范围来识别。此值的默认值是10,可以通过echo来调整:

echo 8 > /proc/sys/vm/dirty_background_ratio

3. /proc/sys/vm/dirty_ratio 为脏页面占用总内存最大的比例,超过这个值,系统不会新增加脏页面,文件读写也变为同步模式。文件读写变为同步模式后,应用程序的文件读写操作的阻塞时间变长,会导致系统性能变慢。此参数的默认值为40,对于写入为主的业务,可以增加此参数,避免磁盘过早的进入到同步写状态。

□ 说明

如果加大了脏数据的缓存大小和时间,在意外断电情况下,丢失数据的概率会变多。因此对于需要立即存盘的数据,应用应该采用O_DIRECT模式避免关键数据的丢失。

4.3.2 调整磁盘文件预读参数

原理

文件预取的原理,就是根据局部性原理,在读取数据时,会多读一定量的相邻数据缓存到内存。如果预读的数据是后续会使用的数据,那么系统性能会提升,如果后续不使用,就浪费了磁盘带宽。在磁盘顺序读的场景下,调大预取值效果会尤其明显。

修改方式

文件预取参数由文件read_ahead_kb指定,CentOS中为"/sys/block/\$DEVICE-NAME/queue/read_ahead_kb"(\$DEVICE-NAME为磁盘名称),如果不确定,则通过命令以下命令来查找。

find / -name read_ahead_kb

此参数的默认值128KB,可使用echo来调整,仍以CentOS为例,将预取值调整为4096KB:

echo 4096 > /sys/block/\$DEVICE-NAME /queue/read ahead kb

□ 说明

这个值实际和读模型相关,要根据实际业务调整。

4.3.3 优化磁盘 IO 调度方式

原理

文件系统在通过驱动读写磁盘时,不会立即将读写请求发送给驱动,而是延迟执行,这样Linux内核的I/O调度器可以将多个读写请求合并为一个请求或者排序(减少机械磁盘的寻址)发送给驱动,提升性能。我们在前文介绍工具iostat时,也提到了合并的统计,这个值就是由这个过程统计获得。

目前Linux版本主要支持3种调度机制:

1. CFQ,完全公平队列调度

早期Linux 内核的默认调度算法,它给每个进程分配一个调度队列,默认以时间片和请求数限定的方式分配IO资源,以此保证每个进程的 IO 资源占用是公平的。这个算法在IO压力大,且IO主要集中在某几个进程的时候,性能不太友好。

2. DeadLine, 最终期限调度

这个调度算法维护了4个队列,读队列,写队列,超时读队列和超时写队列。当内核收到一个新请求时,如果能合并就合并,如果不能合并,就会尝试排序。如果既不能合并,也没有合适的位置插入,就放到读或写队列的最后。一定时间后, I/O 调度器会将读或写队列的请求分别放到超时读队列或者超时写队列。这个算法并不限制每个进程的IO资源,适合IO压力大且IO集中在某几个进程的场景,比如大数据、数据库使用HDD磁盘的场景。

3. NOOP,也叫NONE,是一种简单的FIFO调度策略 因为固态硬盘支持随机读写,所以固态硬盘可以选择这种最简单的调度策略,性 能最好。

修改方式

查看当前的调度方式:

cat /sys/block/\$DEVICE-NAME/queue/scheduler
noop deadline [cfq]

[]中即为当前使用的磁盘IO调度模式。

□说明

操作系统不同,返回的值或默认值可能不同。

如果需要修改,可以采用echo来修改,比如要将sda修改为deadline:

echo deadline > /sys/block/sda/queue/scheduler

4.3.4 文件系统参数优化

Linux支持多种文件系统,不同的文件系统性能上也存在着差异,因此如果可以选择,可以选用性能更好的文件系统,比如XFS。在创建文件系统时,又可以通过增加一些参数进行优化。

另外Linux在挂载文件分区时,也可以增加参数来达到性能提升的目的。

磁盘挂载方式优化 nobarrier

原理

当前Linux文件系统,基本上采用了日志文件系统,确保在系统出错时,可以通过日志进行恢复,保证文件系统的可靠性。Barrier(栅栏),即先加一个栅栏,保证日志总是先写入,然后对应数据才刷新到磁盘,这种方式保证了系统崩溃后磁盘恢复的正确性,但对写入性能有影响。

服务器如果采用了RAID卡,并且RAID本身有电池,或者采用其它保护方案,那么就可以避免异常断电后日志的丢失,我们就可以关闭这个栅栏,可以达到提高性的目的。

修改方式

假如sda挂载在"/home/disk0"目录下,默认的fstab条目是:

mount -o nobarrier -o remount /home/disk0

□ 说明

nobarrier参数使得系统在异常断电时无法确保文件系统日志已经写到磁盘介质,因此只适用于使用了带有保护的RAID卡的情况。

选用性能更优的文件系统 XFS

原理

XFS是一种高性能的日志文件系统,XFS极具伸缩性,非常健壮,特别擅长处理大文件,同时提供平滑的数据传输。因此如果可以选择,我们可以优先选择XFS文件系统。

XFS文件系统在创建时,可先选择加大文件系统的block,更加适用于大文件的操作场景。

修改方式

步骤1 格式化磁盘。假设我们要对sda1进行格式化:

mkfs.xfs /dev/sda1

步骤2 指定blocksize,默认情况下为4KB(4096B),我们假设在格式化时指定为变更为8192B:

mkfs.xfs /dev/sda1 -b size=8192

----结束

4.3.5 使用异步文件操作 libaio 提升系统性能

原理

对于磁盘文件,文件的读取是同步的,导致线程读取文件时,属于阻塞状态。程序为了提升性能和磁盘的吞吐,程序会创建几个单独的磁盘读写线程,并通过信号量等机制进行线程间通信(同时带有锁);显然线程多,锁多,会导致更多的资源抢占,从而导致系统整体性能下降。

libaio提供了磁盘文件读写的异步机制,使得文件读写不用阻塞,结合epoll机制,实现一个线程可以无阻塞的运行,同时处理多个文件读写请求,提升服务器整体性能。

修改方式

参考附录中A libaio实现参考。

鲲鹏性能优化十板斧 5 应用程序调优

5 应用程序调优

- 5.1 调优简介
- 5.2 优化方法

5.1 调优简介

应用程序部署到鲲鹏服务器上以后,需要结合芯片和服务器的特点优化代码性能,使硬件能力得到充分发挥。本章列举几个典型场景,涉及锁、编译器配置、Cacheline、缓冲机制等优化。

5.2 优化方法

5.2.1 优化编译选项,提升程序性能

原理

C/C++代码在编译时,gcc编译器将源码翻译成CPU可识别的指令序列,写入可执行程序的二进制文件中。CPU在执行指令时,通常采用流水线的方式并行执行指令,以提高性能,因此指令执行顺序的编排将对流水线执行效率有很大影响。通常在指令流水线中要考虑:执行指令计算的硬件资源数量、不同指令的执行周期、指令间的数据依赖等等因素。我们可以通过通知编译器,程序所运行的目标平台(CPU)指令集、流水线,来获取更好的指令序列编排。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。

修改方式

- 在Euler系统中使用HCC编译器,可以在CFLAGS和CPPFLAGS里面增加编译选项:
 -mtune=tsv110 -march=armv8-a
- 在其它操作系统中,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

5.2.2 文件缓冲机制选择

原理

内存访问速度要高于磁盘,应用程序在读写磁盘时,通常会经过一些缓存,以减少对磁盘的直接访问,如下图所示:



clib buffer: clib buffer是用户态的一种数据缓冲机制,在启用clib buffer的情况下,数据从应用程序的buffer拷贝至clib buffer后,并不会立即将数据同步到内核,而是缓冲到一定规模或者主动触发的情况下,才会同步到内核;当查询数据时,会优先从clib buffer查询数据。这个机制能减少用户态和内核态的切换(用户态切换内核态占用一定资源)。

PageCache: PageCache是内核态的一种文件缓存机制,用户在读写文件时,先操作 PageCache,内核根据调度机制或者被应用程序主动触发时,会将数据同步到磁盘。 PageCache机制能减少磁盘访问。

修改方式

应用程序根据自己的业务特点选择合适的文件读写方式:

- fread/fwrite函数使用了clib buffer缓存机制,而read/write并没有使用,因此 fread/fwrite比read/write多一层内存拷贝,即从应用程序buffer至clib buffer的拷 贝,但是fread/fwrite比read/write有更少的系统调用。因此对于每次读写字节数 较大的操作,内存拷贝比系统调用占用更多资源,可以使用read/write来减少内存 拷贝;对于每次读写字节数较少的操作,系统调用比内存拷贝占用更多资源,建 议使用fread/ fwrite来减少系统调用次数。
- O_DIRECT模式没有使用PageCache,因此少了一层内存拷贝,但是因为没有缓冲导致每次都是从磁盘里面读取数据。
 - O_DIRECT主要适用场景为:应用程序有自己的缓冲机制;数据读写一次后,后面不再从磁盘读这个数据。

5.2.3 执行结果缓存

原理

对于相同的输入,应用软件经过计算后,有相同的输出,可以将运算结果保存,在下次有相同的输入时,返回上次执行的结果。

修改方式

目前部分开源软件已经实现这种机制,举例如下:

1. Nginx缓冲

基于局部性原理,Nginx使用proxy_cache_path等参数将请求过的内容在本地内存建立一个副本,这样对于缓存中的文件不用去后端服务器去取。

2. JIT编译

JIT(Just-In-Time)编译,将输入文件转为机器码。为了提升效率,转换后的机器码被缓存在内存,这样相同的输入(如JAVA程序的字节码)不用重新翻译,直接返回缓存中的内容。如果发现JAVA虚拟机的C1 Compiler/C2 Compiler线程的CPU占用比较多,可能是JIT缓冲不够,可以增加JAVA虚拟机ReservedCodeCacheSize参数。

3. MySql查询缓冲

MySQL的SQL语句缓存在内存中保存查询返回的结果。当查询命中时,直接返回解析后的SQL,跳过了查询操作的解析,优化和执行阶段。如果缓存的表被修改了,对应表的缓存就会失效。

MySql查询缓冲状态可以通过如下语句查询:

show status like '%Qcache%';

MySql可以通过query_cache_size和query_cache_type来设置查询缓存的属性。

5.2.4 减少内存拷贝

原理

基于数据流分析,发现并减少内存拷贝次数,能降低CPU使用率,并减少内存带宽占用。

修改方式

减少内存拷贝要基于业务逻辑进行分析,这里例举几种减少内存拷贝的实现机制:

样例一: 使用sendfile代替send/sendto/write等函数将文件发送给对端
 如下两条语句将文件发送给对端,一般会有4次内存拷贝

ssize t read (int fd, void *buf, size t count);

ssize_t send (int s, const void * buf, size_t len, int flags);

- read函数一般有2次内存拷贝: DMA将数据搬运到内核的PageCache;内核将数据搬运到应用态的buf。
- send函数一般有2次内存拷贝: write函数将应用态的buf的拷贝到内核; DMA 将数据搬运到网卡。

使用如下函数实现只需要两次内存拷贝:

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);

内核通过 DMA将文件搬运到缓存(一次内存拷贝),然后把缓存的描述信息 (位置和长度)传递给TCP/IP协议栈,内核在通过DMA将缓冲搬运到网卡 (第二次内存拷贝)。

除了修改代码,部分开源软件已经支持这个特性,如Nginx可以通过sendfile on参数打开这个功能。

● **样例二**:进程间通信使用共享内存代替socket/pipe通信

内存共享方式可以让多个进程操作同样的内存区域,相比socket通信的的方式, 内存拷贝少。应用程序可以使用shmget等函数实现进程间通信。

5.2.5 锁优化

原理

自旋锁和CAS指令都是基于原子操作指令实现,当应用程序在执行原子操作失败后,并不会释放CPU资源,而是一直循环运行直到原子操作执行成功为止,导致CPU资源浪费。如下图代码的黄色部分是一个循环等待过程:

```
int NoBarrier_CompareAndSwap(volatile int* ptr, int old_value, int new_value)
int prev;
int temp;
  _asm__ _volatile__ (
   'a:
   "ldaxr %w[prev], %[ptr]
   "cmp %w[prev], %w[old_value]
   "bne 1f
   "stlxr %w[temp], %w[new_value], %[ptr] \n\t" // Try to store the new value. stxr
   cbnz ‰[temp], 0b
   : [prev] "=&r" (prev),
    [temp] "=&r" (temp),
    [ptr]"+Q" (*ptr)
   : [old_value]"IJr" (old_value),
     [new_value]"r" (new_value)
   : "cc", "memory"
return prev;
```

修改方式

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。
- 减少线程并发数:参考2.3.3 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**5.2.7 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 ldaxn+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*/
```

5.2.6 使用 jemalloc 优化内存分配

原理

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。

修改方式

步骤1 下载jemalloc,参考INSTALL.md编译安装。

源码下载地址https://github.com/jemalloc/jemalloc

步骤2 修改应用软件的链接库的方式,在编译选项中添加如下编译选项:

-I'jemalloc-config --includedir'-L'jemalloc-config --libdir' -Wl,-rpath,'jemalloc-config --libdir' -ljemalloc 'jemalloc-config --libs'

具体参考https://github.com/jemalloc/jemalloc/wiki/Getting-Started

步骤3 部分开源软件可以修改配置参数来指定内存分配库,如MySql可以配置my.cnf文件: malloc-lib=/usr/local/lib/libjemalloc.so

----结束

5.2.7 Cacheline 优化

原理

CPU标识Cache中的数据是否为有效数据不是以内存位宽为单位,而是以Cacheline为单位。这个机制可能会导致伪共享(false sharing)现象,从而使得CPU的Cache命中率变低。出现伪共享的常见原因是高频访问的数据未按照Cacheline大小对齐。

Cache空间大小划分成不同的Cacheline,示意图如<mark>图5-1</mark>所示。readHighFreq虽然没有被改写,且在Cache中,在发生伪共享时,也是从内存中读。

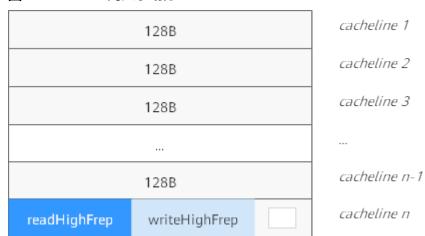


图 5-1 Cache 空间大小划分

例如以下代码定义两个变量,会在同一个Cacheline中,Cache会同时读入:

int readHighFreq, writeHighFreq

其中readHighFreq是读频率高的变量,writeHighFreq为写频率高的变量。 writeHighFreq在一个CPU core里面被改写后,这个cache 中对应的Cacheline长度的 数据被标识为无效,也就是readHighFreq被CPU core标识为无效数据,虽然 readHighFreq并没有被修改,但是CPU在访问readHighFreq时,依然会从内存重新导入,出现伪共享导致性能降低。

鲲鹏920和x86的Cacheline大小不一致,可能会出现在X86上优化好的程序在鲲鹏 920上运行时的性能偏低的情况,需要重新修改业务代码数据内存对齐大小。X86 L3 cache的Cacheline大小为64字节,鲲鹏920的Cacheline为128字节。

修改方式

- 修改业务代码,使得读写频繁的数据以Cacheline大小对齐,修改方法可参考:
 - a. 使用动态申请内存的对齐方法:
 int posix_memalign(void **memptr, size_t alignment, size_t size)
 调用posix_memalign函数成功时会返回size字节的动态内存,并且这块内存的起始地址是alignment的倍数。
 - b. 局部变量可以采用填充的方式:
 int writeHighFreq;
 char pad[CACHE_LINE_SIZE sizeof(int)];

代码中CACHE_LINE_SIZE是服务器Cacheline的大小,pad变量没有用处,用于填充writeHighFreg变量余下的空间,两者之和是CacheLine大小。

2. 部分开源软件代码中有Cacheline的宏定义,修改宏的值即可。如在impala使用 CACHE LINE SIZE宏来表示目标平台的Cacheline大小。

5.2.8 NEON 指令加速

NEON是一种基于SIMD思想的技术,能够基于单条指令对多个数据同时进行操作,其使用的NEON指令类似于Intel CPU下的MMX/SSE/AVX指令,通过向量化的计算方式优化应用程序性能,通常应用于图像处理、音视频处理、数据并行处理等需要大量计算场景。

NEON技术依赖于128位NEON寄存器的硬件支持,NEON寄存器是一种向量寄存器,一个寄存器中可存储多个数据元素,但要求其具有相同的数据类型。

以下是ARMv8-A中aarch64架构下的寄存器:

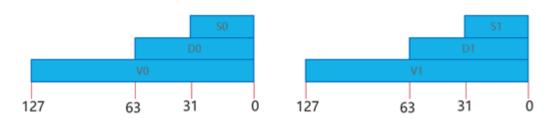
● 64位寄存器: (通用寄存器)

ARMv8 有31 个64位通用寄存器,1个特殊寄存器。因此可以看成31个64位的X寄存器或者31个32位的W寄存器(X寄存器的低32位)。



● 128位寄存器: (向量寄存器)

ARMv8有32个128位的V寄存器,同样也可以看成是32个32位的S寄存器或者32个64位的D寄存器。



使用编译器能力自动向量化加速

原理:

编译器支持自动向量化功能,其会自动利用NEON属性,编译时将代码向量化。启用自动向量化功能前需要打开相应的编译选项,且并非所有代码均可向量化,其需要符合一定的编码方式和规律,以提供更多的提示信息给编译器,进一步触发编译器进行代码的向量化。

支持该特性的编译器有: gcc、LLVM、适用于嵌入式和Linux项目的Arm编译器。

修改方式:

- 自动向量化编译选项使能
 - gcc编译器使用-O3会自动使能-ftree-vectorize选项,在-O1和-O2下需要添加-ftree-vectorize选项才能进行向量化。在-O0模式下,即使添加-ftree-vectorize也无法进行向量化。
 - armcc编译器使用-vectorize选项来使能向量化编译,一般选择更高的优化等级如-O2或者-O3就能使能-vectorize选项。在-O1模式下需要使用-vectorize选项使能向量化编译,在-O0模式下,即使添加-vectorize选项编译器同样无法进行向量化。

在armv8-a的aarch64架构下才支持双浮点计算的向量化,其他架构下非必需时避免使用双浮点的数据类型,该类型会阻止编译器做向量化。各架构下支持的数据类型如下:

-	Armv7-A/R	Armv8-A/R	Armv8-A	
-	-	aarch32	aarch64	
Floating-point	32-bit	16-bit/32-bit	16-bit/32-bit/64- bit	
Integer	8-bit/16-bit/32- bit	8-bit/16-bit/32- bit/64-bit	8-bit/16-bit/32- bit/64-bit	

arch命令下可查看cpu硬件架构是aarch64还是aarch32。

- 编码方式上触发代码向量化
 - a. 循环次数在已知时要直接传递常数,而不使用变量,让编译器预先明确循环 迭代次数。循环次数是2的指数倍时,需告知编译器,以便尽可能的向量化。 在循环次数非2的指数倍时,也可将循环分解进行构造。

```
void vecAdd(int *vecA, int *vecB, int *vecC, int len)
{
    int i;
    // 告诉编译器len是4的整数倍
    for (i = 0; i < len * 4; i++) {
        vecC[i] = vecA[i] + vecB[i];
    }
}
```

- b. 在控制循环结束的条件中,尽量使用 "<"来进行条件判断,而不使用 "<="或"! =",使用 "<"能使编译器识别到在该变量值之前循环结束,这有助于编译器进行向量化。
- c. 使用restrict关键字

为指针添加_restrict或_restrict_关键字,提示编译器,对象已经被指针所引用,不能通过除该指针外所有其他直接或间接的方式修改该对象的内容,编译器以此获知当前对象无其他依赖,可并行操作和向量化。但使用前必须确保确实没有指针访问区域重叠的现象,否则计算结果可能会出错。

```
void vecAdd(int *_restrict__ vecA, int *_restrict__ vecB, int *_restrict__ vecC, int len)
{
   int i;
   for (i = 0; i < len *4; i++) {
      vecC[i] = vecA[i] + vecB[i];
   }
}</pre>
```

鲲鹏性能优化十板斧 5 应用程序调优

- 避免循环依赖(即某次循环的结果会被前一次循环的结果影响)。 d.
- 在满足需求情况下,使用尽可能小的数据类型,以便向量化后,NEON寄存 器一次能处理更多数据,提升向量化后代码性能。
- 避免在循环中出现条件判断,尽量少用break跳出循环。 f.
- 编写简单的代码,编译器更容易理解与自动向量优化。(向量化程度取决于编 译器所理解编码人员代码意图的程度)
- 用数组下标来替代指针访问元素。 h.
- 构造结构体时,可尽量保持结构体内变量的数据类型一致,便于数据加载时 向量化。

如下为像素点数据结构体做4字节对齐,采用以下方式可进行向量化:

```
struct aligned pixel {
  char r;
  char g;
  char b;
  char not_used; /* Padding used to keep r aligned to a 32-bit word */
}screen[10];
若只改变结构体内单个元素变量类型进行数据对齐,导致结构体内变量数据
```

类型不同,则无法进行自动向量化:

```
struct pixel {
  char r;
  short g; /* Green channel contains more information */
  char b;
}screen[10];
```

使用 NEON intrinsic 加速提升性能

原理:

NEON intrinsic函数是一系列C函数调用,编译器可将其替换为适当的NEON指令或 NEON指令序列。NEON intrinsic函数几乎提供与编写NEON汇编指令相同的功能,但 是将寄存器分配等工作留给编译器,以便开发人员可以专注于算法开发。与使用 NEON汇编指令编码相比,NEON intrinsic方式的代码有更好的可维护性。arm编译 器、gcc和LLVM编译器都支持NEON intrinsic。

修改方式:

在使用NEON intrinsic 函数时需要增加头文件#include <arm_neon.h>,详细的NEON intrinsic函数列表和使用方法,可参考NEON Intrinsic Reference: https:// developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics

鲲鹏性能优化十板斧 6 JVM 性能调优

6 JVM 性能调优

- 6.1 调优简介
- 6.2 常用性能监测工具
- 6.3 JVM原理及配置建议
- 6.4 优化调优方法

6.1 调优简介

调优思路

JVM是Java Virtual Machine (Java虚拟机)的缩写,Java代码在不同平台上运行时不需要重新编译,Java语言使用JVM屏蔽了与具体平台相关的硬件指令差异,使得Java语言编译程序只需生成在JVM上运行的字节码,实现在多种平台上不加修改地运行。JVM包括即时编译(JIT)、内存管理(垃圾回收GC技术)和Runtime技术,其中GC调优是性能调优中应用最为广泛,本章调优思路主要针对GC展开说明,首先优选尽可能高的JDK版本,高版本有更新的特性和优化,对Java程序性能有好处;其次根据实际业务场景和硬件资源给JVM选择合理的堆空间;最后要选择合理的GC算法。同时,Java自带很多工具,对程序运行的检测和性能分析都很有帮助,利用这些工具可以辅助Java性能调优。

主要优化参数

优化项	优化项简介	默认值	生效范围	鲲鹏 916	鲲鹏 920
-Xmx	设置JVM最大可用 堆内存大小。	根据系统资 源计算默认 值	Java进程重 启生效	Υ	Υ
-Xms	设置初始堆大小, 一般和Xmx保持一 致。	根据系统资 源计算默认 值	Java进程重 启生效	Υ	Υ

鲲鹏性能优化十板斧 6 JVM 性能调优

优化项	优化项简介	默认值	生效范围	鲲鹏 916	鲲鹏 920
-Xmn	设置年轻代堆大 小。	根据系统资 源计算默认 值	Java进程重 启生效	Υ	Υ
-Xss	设置每个线程的堆 大小。	JDK 1.5以后 每个线程堆 栈大小默认 为1MB,1.5 以前为 256KB。	Java进程重 启生效	Y	Υ

6.2 常用性能监测工具

6.2.1 jstat 工具

介绍

jstat是JDK自带的一个JVM统计监控工具,利用JVM内建的指令对Java应用程序的资源 和性能进行实时的命令行的监控,包括堆大小和应用程序GC状况的监控。

安装方式

完整安装JDK后自带jstat工具,无需单独安装,一般位于java的bin目录下。

使用方式

jstat是个非常强大的命令,可选项多,可以详细查看堆内各个部分的使用量,以及加载类的数量。使用时,需加上查看进程的进程pid和所选参数。

命令格式: jstat [参数] <pid>

一级参数主要功能如下:

jstat -class <pid></pid>	显示加载class的数量,及所占空间等信息。
jstat -compiler <pid></pid>	显示虚拟机实时编译的数量等信息。
jstat -gc <pid></pid>	显示GC的信息,查看GC的次数及时间。
jstat -gccapacity <pid></pid>	显示虚拟机内存中三代对象的使用和占用大小。
jstat -gcutil <pid></pid>	显示GC的统计信息。
jstat -gcnew <pid></pid>	显示年轻代对象的信息。
jstat -gcnewcapacity <pid></pid>	显示年轻代对象信息及其内存占用情况。
jstat -gcold <pid></pid>	显示老年代对象的信息。

鲲鹏性能优化十板斧 6 JVM 性能调优

jstat -gcpermcapacity <pid></pid>	显示永久代对象的信息及其占用量。
jstat -printcompilation <pid></pid>	显示当前虚拟机的执行信息。

GC优化中,利用jstat -gc <pid>较多,其输出参数及含义如下:

显示参数	含义
SOC	年轻代中第一个survivor(幸存区)的容量 (字节)
S1C	年轻代中第二个survivor(幸存区)的容量 (字节)
SOU	年轻代中第一个survivor(幸存区)目前已使用空间 (字节)
S1U	年轻代中第二个survivor(幸存区)目前已使用空间 (字节)
EC	年轻代中Eden(伊甸园)的容量 (字节)
EU	年轻代中Eden(伊甸园)目前已使用空间 (字节)
ОС	Old代的容量 (字节)
OU	Old代目前已使用空间 (字节)
PC	Perm(持久代)的容量 (字节)
PU	Perm(持久代)目前已使用空间 (字节)
YGC	从应用程序启动到采样时年轻代中gc次数
YGCT	从应用程序启动到采样时年轻代中gc所用时间(s)
FGC	从应用程序启动到采样时old代(全gc)gc次数
FGCT	从应用程序启动到采样时old代(全gc)gc所用时间(s)
GCT	从应用程序启动到采样时gc用的总时间(s)

输出格式:

jstat -gc 2342

sh-4.4# jstat -gc 159 1000 SOC S1C SOU S1U FGC FGCT GCT EC EU OC OU MC MU CCSC CCSU YGC 6144.0 13824.0 6128.8 0.0 702464.0 590176.1 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314 6144.0 13824.0 6128.8 0.0 702464.0 590237.9 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314 6144.0 13824.0 6128.8 0.0 702464.0 592583.5 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314 6144.0 13824.0 6128.8 0.0 702464.0 592715.7 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314 6144.0 13824.0 6128.8 0.0 702464.0 592715.7 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314 6144.0 13824.0 6128.8 0.0 702464.0 592715.7 497664.0 468611.6 139032.0 131745.2 15664.0 14404.5 151478 1673.004 1057 343.310 2016.314

鲲鹏性能优化十板斧 6 JVM 性能调优

6.2.2 jmap 工具

介绍

imap是JDK自带的堆信息查看和调试工具,可以将堆信息导出到文件分析,可以查看 堆空间分配等信息,是iava性能调优常用工具之一。

安装方式

完整安装JDK后自带imap工具,无需单独安装,一般位于java的bin目录下。

使用方式

命令格式: jmap [参数]

常用参数如下:

1. -dump:[live,]format=b,file=/you/path/filename.hprof <pid>

说明:输出jvm的堆对象内容到制定文件。推荐以.hprof后缀命名文件,可以用 MAT工具直接分析。 live选项是可选的,如果选live,那么只输出活的对象到文

举例: jmap -dump:format=b,file=/opt/log/java12123.hprof 12123

-finalizerinfo <pid>

说明:输出正等候回收的对象的信息

举例: jmap -finalizerinfo 12123

3. -heap <pid>

说明:输出当前java进程堆的概要统计信息,如GC算法,heap的配置空间等等

举例: jmap -heap 12123

-histo[:live] <pid>

说明:输出当前class的实例数目,内存占用,类全名信息。

举例: jmap -histo:live 12123 | head -n 20

-permstat <pid>

说明:打印classload和jvm heap中perm代的信息.包含每个classloader的名字,活 泼性,地址,父classloader和加载的class数量等信息。

举例: jmap -permstat 12123

jmap -heap <pid> 举例说明:

using parallel threads in the new generation. ##新生代采用的是并行线程处理方式

using thread-local object allocation.

Concurrent Mark-Sweep GC ##同步并行垃圾回收

Heap Configuration: ##堆配置情况

MinHeapFreeRatio = 40 ##最小堆使用比例 MaxHeapFreeRatio = 70 ##最大堆可用比例

MaxHeapSize = 2147483648 (2048.0MB) ##最大堆空间大小

= 268435456 (256.0MB) ##新生代分配大小 NewSize

MaxNewSize = 268435456 (256.0MB) ##最大可新生代分配大小

OldSize = 5439488 (5.1875MB) ##老生代大小 NewRatio = 2 ##新生代比例

= 2 ##新生代比例 NewRatio

SurvivorRatio = 8 ##新生代与suvivor的比例

PermSize = 134217728 (128.0MB) ##perm区大小

MaxPermSize = 134217728 (128.0MB) ##最大可分配perm区大小

鲲鹏性能优化十板斧 6 JVM 性能调优

Heap Usage: ##堆使用情况 New Generation (Eden + 1 Survivor Space): ##新生代(伊甸区 + survior空间) capacity = 241631232 (230.4375MB) ##伊甸区容量 used = 77776272 (74.17323303222656MB) ##已经使用大小 = 163854960 (156.26426696777344MB) ##剩余容量 32.188004570534986% used ##使用比例 Eden Space: ##伊甸区 capacity = 214827008 (204.875MB) ##伊甸区容量 used = 74442288 (70.99369812011719MB) ##伊甸区使用 free = 140384720 (133.8813018798828MB) ##伊甸区当前剩余容量 34.65220164496263% used ##伊甸区使用情况 From Space: ##survior1区 capacity = 26804224 (25.5625MB) ##survior1区容量 used = 3333984 (3.179534912109375MB) ##surviror1区已使用情况 free = 23470240 (22.382965087890625MB) ##surviror1区剩余容量 12.43827838477995% used ##survior1区使用比例 To Space: ##survior2 区 capacity = 26804224 (25.5625MB) ##survior2区容量 used = 0 (0.0MB) ##survior2区已使用情况 free = 26804224 (25.5625MB) ##survior2区剩余容量 0.0% used ## survior2区使用比例 concurrent mark-sweep generation: ##老生代使用情况 capacity = 1879048192 (1792.0MB) ##老生代容量 used = 30847928 (29.41887664794922MB) ##老生代已使用容量 = 1848200264 (1762.5811233520508MB) ##老生代剩余容量 1.6416783843721663% used ##老生代使用比例 Perm Generation: ##perm区使用情况 capacity = 134217728 (128.0MB) ##perm区容量 used = 47303016 (45.111671447753906MB) ##perm区已使用容量 free = 86914712 (82.8883285522461MB) ##perm区剩余容量 35.24349331855774% used ##perm区使用比例

信息来源:https://blog.csdn.net/zhaozheng7758/article/details/8623530

6.3 JVM 原理及配置建议

6.3.1 尽量使用高版本 JDK

原理

当前主流使用的是Oracle官方的JDK版本,Open JDK是官方JDK的开源版本,两者核心源码和技术一致,Open JDK 更新发布更加频繁,目前使用用户数量正在明显增加。 Huawei JDK是Open JDK的一个发行版本,基于Open JDK对鲲鹏芯片和欧拉系统做了很多针对性优化。

新的JDK版本会增加新特性让编程更加方便简洁,同时对老版本的问题进行修复和改进,建议尽量使用新的版本。生产环境会关注程序运行稳定性,可选已经运行一段时间的成熟的高版本。如果运行在鲲鹏芯片上,推荐使用1.8及以上JDK版本。

- 使用开源版Open JDK版本,到如下网址获取安装:**http://openjdk.java.net/**
- 使用Huawei JDK版本,版本2020年底将随OpenEuler开源发布。

修改方式

建议使用高版本JDK,并重新安装。

查询JDK版本的方式,执行:

java -version

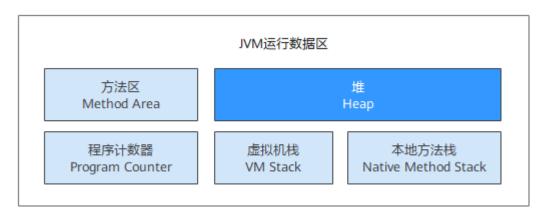
鲲鹏性能优化十板斧 6 JVM 性能调优

openjdkversion "1.8.0_232"
OpenJDKRuntime Environment (build 1.8.0_232-Huawei_JDK_V100R001C00SPC173B001-b09)
OpenJDK64-Bit Server VM (build 25.232-b09, mixed mode)

6.3.2 设置 JVM 堆空间大小

原理

JVM在执行Java程序时会把它所管理的内存划分为若干个不同的运行时数据区域,主要包括:程序计数器、方法区、虚拟机栈、本地方法栈和堆:



- 1. 程序计数器可以看作时当前线程所执行的字节码的行号指示器。
- 2. 方法区用于存储被JVM加载的类信息、常量、静态变量等数据。
- 3. 虚拟机栈存储的时Java方法执行的线程内存模型,每一个方法被调用到执行完毕的过程,就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。
- 4. 本地方法栈和虚拟机栈的功能相同,差别是本地方法栈只为本地方法调用服务。
- 5. 堆是JVM管理内存中占用比例最大的一块,用于存储Java程序对象实例,几乎所有的对象实例内存都在这里分配。从内存回收角度和经典垃圾收集器分带理论上看,堆内存空间一般被分为:新生代、老年代、永久代、Eden空间、From/ToSurvivor等区域。各代功能和划分说明可以参考:https://www.cnblogs.com/fangfuhai/p/7206944.html。

针对堆空间的优化是Java性能调优的重点之一。如果没有设置JVM堆空间大小,JVM会根据服务器物理内存大小设置默认堆大小的值。例如,在64位的服务器端,当物理内存小于192MB时,JVM堆大小默认选为物理内存的一半;当物理内存大192MB且小于128GB时,JVM堆大小默认选为物理内存的四分之一;当物理内存大于等于128GB时,都为32GB。通常情况下,Java应用程序的会通过参数指定堆大小,具体方法下文会有说明。

应用程序选用多大的堆空间大小及配比,一般要根据程序的GC情况和服务器内存资源进行综合评估,是个循序渐进不断优化的过程,如果垃圾回收(GC)频繁触发,可以尝试增加堆空间缓解。

推荐配置原则:

- 应用程序运行时,计算老年代存活对象的占用空间大小X。程序整个堆大小(Xmx和Xms)设置为X的3-4倍;永久代 PermSize和MaxPermSize设置为X的1.2-1.5倍。 年轻代Xmn的设置为X的1-1.5倍。老年代内存大小设置为X的2-3倍。
- 2. JDK官方建议年轻代占整个堆大小空间的3/8左右。
- 3. 完成一次Full GC后,应该释放出70%的堆空间(30%的空间仍然占用)。

鲲鹏性能优化十板斧 6 JVM 性能调优

修改方式

在Java应用程序启动时,添加如下参数并设置大小:

-Xmx	设置JVM最大可用堆内存大小。
-Xms	设置初始堆大小,一般和Xmx保持一致。
-Xmn	设置年轻代堆大小。
-Xss	设置每个线程的堆大小。JDK 1.5以后每个线程堆栈大小默认为 1MB,1.5以前为256K。
-XX:NewRatio=	设置年轻代(包括Eden和两个Survivor区)与年老代的比值 (不包括持久代)。如设置为4,则年轻代与年老代所占比值为 1:4,年轻代占整个堆栈的1/5。
- XX:SurvivorRatio=	设置年轻代中Eden区与Survivor区的大小比值。如设置为4,则两个Survivor区与一个Eden区的比值为2:4,一个Survivor区占整个年轻代的1/6。
- XX:MaxPermSize=	设置持久代堆大小。

举例:

java -Xmx3600m -Xms3600m -Xmn2g -Xss128k

设置最大堆空间为3600MB,初始化堆大小为3600MB,年轻代大小为2GB,线程堆大小为128KB。

6.3.3 选择合适的垃圾回收器

原理

垃圾回收器是内存回收的具体实现,JDK自带的垃圾回收器已经完成集成垃圾回收和清理算法,业务程序可以通过设置参数选择垃圾回收器,虚拟机用到的7种经典的垃圾回收器如下表。根据适用内存区域不同,JDK自带的垃圾回收器可分为新生代回收器和老年代回收器,两者可以配合使用。新生代回收器用于堆空间中新生代区域的垃圾回收,老年代回收器用于堆空间中老年代区域的垃圾回收。G1是一种新型的堆内垃圾收集器,既可以用于新生代也可以用于老年代垃圾回收。

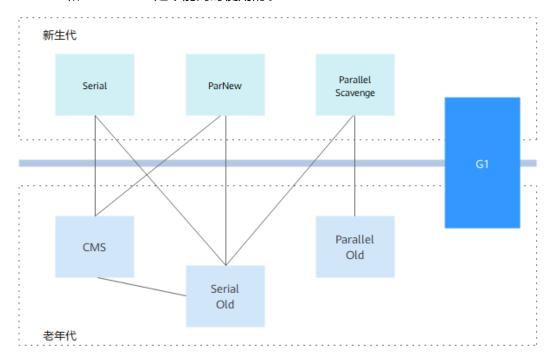
下表列举了经典的7种垃圾回收器的说明和功能分类:

名称	说明	收集模式	分代适用类型
Serial	单线程串行收集器	串行收集 器	新生代
ParNew	多线程并行Serial收集器	并行收集 器	新生代
Parallel Scavenge	并行吞吐量优先收集器	并行收集 器	新生代

鲲鹏性能优化十板斧 6 JVM 性能调优

名称	说明	收集模式	分代适用类型
Serial Old	Serial单线程收集器老年代 版本	串行收集 器	老年代
CMS(Concurrent Mark Sweep)	并行最短停顿时间收集器	并发收集 器	老年代
Parallel Old	Parallel Scavenge并行收 集器老年代版本	并行收集 器	老年代
G1	面向局部收集和基于 Region内存布局的新型低 延时收集器	并发/并行 收集器	新生代/老年代

下图展示了新生代GC和老年代GC配合使用方法,有连线的表示可以配合使用。注意 ParNew和Parallel Old是不能同时使用的。



垃圾回收器的选择方法没有通用的准则,要结合项目应用的实际并对GC运行数据的检测来决定。

根据收集模式经典垃圾回收器可分为三类:串行收集器、并行收集器、并发收集器。串行收集器只适用于小数据量的情况,选择主要针对并行收集器和并发收集器。默认情况下,JDK1.5以前都是使用串行收集器,如果想使用其他收集器需要在启动时加入相应参数。JDK1.5以后,JVM会根据当前系统配置进行判断。

垃圾回收器选择建议:

- 业务应用对吞吐量要求较高,对响应时间没有特别要求的,推荐使用并行收集器。如: 科学计算和后台处理程序等等。
- 对响应时间要求较高的中大型应用程序,推荐使用并发收集器。如:web服务器等。

鲲鹏性能优化十板斧 6 JVM 性能调优

 对应JDK版本1.8以上,多CPU处理器且内存资源不是瓶颈,建议优先考虑使用G1 回收器。

• 单线程应用使用串行收集器。

修改方式

以下表格汇总了各种回收器的分类、特点和修改参数:

名称	修改参数	特点
Serial	-XX:+UseSerialGC	用于新生代的单线程收集器,采用复制算法 进行垃圾收集。Serial 进行垃圾收集时,所 有的用户线程必须暂停(Stop The World)。
ParNew	-XX: +UseParNewGC	Serial 的多线程版本,在单核 CPU 环境并不会比 Serial 更优,它默认开启的收集线程数和 CPU 核数,可以通过 - XX:ParallelGCThreads 来设置垃圾收集的线程数。
Parallel Scavenge	-XX: +UseParallelGC jdk1.7、jdk1.8 新 生代默认使用	用于新生代的多线程收集器, ParNew 的目标是尽可能缩短垃圾收集时用户线程的停顿时间, Parallel Scavenge 的目标是达到一个可控制的吞吐量。通过-XX:MaxGCPauseMillis 来设置收集器尽可能在多长时间内完成内存回收,通过 -XX:GCTimeRatio 来精确控制吞吐量
Serial Old	-XX: +UseSerialOldGC	Serial 的老年代版本,采用标记-整理算法单 线程收集器
CMS	-XX: +UserConMarkSw eepGC	一种以最短回收停顿时间为目标的收集器,尽量做到最短用户线程停顿时间。CMS是基于标记-清除算法,所以垃圾回收后会产生空间碎片,通过 - XX:UserCMSCompactAtFullCollection 开启碎片整理(默认开启)。用 - XX:CMSFullGCsBeforeCompaction 设置执行多少次不压缩(不进行碎片整理)的 Full GC 之后,跟着来一次带压缩(碎片整理)的 Full GC。-XX:ParallelCMSThreads:设定 CMS的线程数量。
Parallel Old	-XX: +UseParallelOldG C jdk1.7、jdk1.8老 年代默认使用	Parallel Scavenge的老年代版本,使用- XX:ParallelGCThreads限制线程数量。
G1	-XX:+UseG1GC jdk1.7以后才提 供,jdk1.9默认	一款全新的收集器,兼顾并行和并发功能, 能充分利用多CPU资源,运行期间不会产生 内存碎片。通过 -XX:ParallelGCThreads 设 置限制线程数量;-XX:MaxGCPauseMillis 设 置最大停顿时间。

鲲鹏性能优化十板斧 6 JVM 性能调优

6.4 优化调优方法

6.4.1 GC 分析优化

原理

每次GC过程都是对JVM堆空间的整理,JVM执行GC业务进程会进入暂停状态,GC优化的总体方向是尽可能减少对业务的影响。总体来说,GC调优是尽量降低GC频率和每次GC停顿时间。可以通过GC日志和jstat工具对GC进行实时监控分析和调优。

下面是jstat命令每隔1秒打印的GC信息,内容只保留了GC评论相关的列信息,其他列未列出。jstat命令试用方法和输出信息说明,请参考6.2.1 jstat工具。

```
YGCT
                   FGC FGCT
                                      GCT
579
      22.233
                    2 0.090
                                    22.323
579
      22.233
                    2
                       0.090
                                    22.323
      22.233
                       0.090
                                    22.323
579
      22.233
                       0.090
579
                    2
                                    22.323
579
      22.233
                       0.090
                                    22.323
579
      22.233
                    2
                       0.090
                                    22.323
579
      22.233
                       0.090
                                    22.323
579
      22.233
                       0.090
                                    22.323
      22.262
580
                    2
                       0.090
                                    22.352
580
      22.262
                       0.090
                                    22.352
                       0.090
580
      22.262
                    2
                                    22.352
580
      22.262
                    2
                       0.090
                                    22.352
580
      22.262
                       0.090
                                    22.352
      22.262
                       0.090
580
                    2
                                    22.352
580
      22.262
                       0.090
                                    22.352
      22 262
                    2
                       0.090
580
                                    22 352
581
      22.282
                       0.090
                                    22.372
581
      22.282
                    2
                       0.090
                                    22.372
581
      22.282
                    2
                       0.090
                                    22.372
581
      22.282
                       0.090
                                    22.372
                       0.090
581
      22.282
                    2
                                    22.372
581
      22.282
                       0.090
                                    22.372
                    2
                       0.090
581
      22.282
                                    22.372
582
      22.282
                       0.090
                                    22.372
582
      22.300
                    2
                       0.090
                                    22.391
582
      22.300
                    2
                       0.090
                                    22.391
582
      22.300
                    2
                       0.090
                                    22.391
      22.300
                       0.090
582
                    2
                                    22.391
582
      22.300
                    2
                       0.090
                                    22.391
582
      22.300
                    2
                       0.090
                                    22.391
582
    22.300
                    2 0.090
                                    22.391
```

- 1. 通过上面监控信息,YGC(Young GC/Minor GC)共执行了4次,也就是第579、580、581和582次; FGC(FULL GC)没有执行。
- 2. 计算GC频率:监控信息每秒打印一次,YGC标号为580的信息打印了8次,也就是说从上次GC完毕等了8秒时间触发了YGC,下次打印就进入了581记录区域。也就是说YGC频率是8秒1次。同样方法可以计算FGC的频率。
- 3. 计算GC耗时(业务停顿时间): YGCT表示从java进程启动开始所有YGC累计执行的总时间,计算一次YGC耗时,只需将后一个YGC标号时间减去当前YGC标号时间即可。例如,第580次YGC执行时间为: 22.282(YGCT 581标号值)减去22.262(YGCT 580标号值)的值,也就是20毫秒。同样方法可以计算FGC耗时。随业务运行压力不同,GC耗时可能会有变动。

关于GC频率和GC耗时在什么范围内比较合理这个问题,没有统一答案,它们的值和服务器硬件资源(如,内存大小、CPU处理能力)和软件相关,每个业务不尽相同。

鲲鹏性能优化十板斧 6 JVM 性能调优

参考建议:

- YGC平均耗时小于50ms
- YGC执行频率不低于10秒1次
- FGC平均耗时小于1s
- FGC执行频率不低于10分钟1次

修改方式

排查调优步骤和建议:

1. FGC过于频繁,检查业务是否有显示调用System.gc()代码,确认是否为业务触发FGC。优化业务代码删掉不必要的显示调用FGC,或者设置-XX:DisableExplicitGC关闭显示调用:

/path/java -XX:DisableExplicitGC <other parameters>

- 2. FGC过于频繁,建议尝试调大永久代(Perm区)空间或者调大老年代空间,具体设置参数见**6.3.2 设置JVM堆空间大小**。
- 3. YGC过于频繁,建议尝试调大年轻代空间或者调大整个堆空间,具体设置参数见 6.3.2 设置JVM堆空间大小。如果业务应用对象生成过多不能重复使用,也可以尝试优化业务代码降低YGC的频率。
- 4. YGC耗时过长,建议尝试缩小年轻代空间大小或者调小整个堆空间,具体设置参数 见**6.3.2 设置JVM堆空间大小**。
- 5. FGC/YGC 耗时过长,系统CPU资源不是瓶颈场景加可尝试增加GC线程数量,具体设置方法参考6.4.2 JVM线程优化。

□ 说明

- 1. GC优化需要结合业务分析,了解业务特点再做针对性优化。
- 2. 堆空间调大或者调小,均会带来优点和缺点,根据业务性能关注点决定。
 - 1. 更大的年轻代会使年老代变小,大的年轻代会延长YGC的周期,但会增加每次YGC的耗时;小的年老代会增加FGC的频率。
 - 2. 更小的年轻代会使老年代变大,小的年轻代会导致YGC很频繁,但每次YGC耗时会减少; 大的老年代会减少FGC的频率。

6.4.2 JVM 线程优化

原理

业务进程运行期间,JVM会开启回收线程进行及时编译和堆空间管理垃圾回收等操作,JVM运行线程多少对系统性能有直接影响。通过资源监控找到占用CPU较高的JVM 线程,并做相关调优处理,能优化业务程序运行性能。

通过下面方法找到占用CPU较高的JVM线程:

步骤1 利用top命令找到java进程ID,如此示例的进程ID是16498:

PID USER	PR NI	VIRT	RES	SHR S	%CPU	%MEM	TIME+ COMMAND
16498 grs	0 -20	20.8g	3.3g	21888 S	4.0	14.0	58:52.94 java

步骤2 利用top -H -p <pid>命令跟踪进程下面各个线程的CPU占用情况,如下图这里取 16810线程做示例。

鲲鹏性能优化十板斧 6 JVM 性能调优

top - 22:46:40 up 11:52, 1 user, load average: 0.03, 0.14, 0.10 Threads: 224 total, 0 running, 224 sleeping, 0 stopped, 0 zombie %Cpu(s): 0.3 us, 0.2 sy, 0.0 ni, 99.4 id, 0.0 wa, 0.1 hi, 0.0 si, 0.0 st MiB Mem : 23931.6 total, 13027.9 free, 3976.0 used, 6927.6 buff/cache MiB Swap: 0.0 total, 0.0 free, 0.0 used. 19541.5 avail Mem							
PID USER	PR NI	VIRT	RES	SHR S	%CPU	%MEM	TIME+ COMMAND
16810 grs	20 0	20.8g	3.3g	21888 S	0.3	14.0	0:04.85 java
17090 grs	20 0	20.8g	3.3g	21888 S	0.3	14.0	0:09.08 java
16498 grs	0 -20	20.8g	3.3g	21888 S	0.0	14.0	0:00.00 java
16506 grs	20 0	20.8g	3.3g	21888 S	0.0	14.0	0:00.68 java
16514 grs	20 0	20.8g	3.3g	21888 S	0.0	14.0	0:08.09 java

步骤3 将线程ID转换成16进制,可以使用Windows操作系统计算器转换,如16810转换为41aa

步骤4 使用jstack命令查询此线程栈信息: jstack -l <进程ID> | grep <线程16进制ID>

```
[grs@host-10-33-66-127 ~]$ jstack -l 16498 | grep 41aa
"cronJobScheduler_Worker-8" #54 prio=5 os_prio=0 tid=0x000003fd8d32e000 nid=0x41aa in Object.wait() [0x000003fd405de000
```

可以看出此线程名称是"cronJobScheduler_Worker-8",这个不是JVM的线程。下图中 "Gang worker#0 (Parallel GC Threads)"、"Concurrent Mark-Sweep GC Thread"、"Gang worker#1 (Parallel CMS Threads)"都属于JVM线程。

```
'VM Thread" os_prio=0 tid=0x000003fe40157000 nid=0x408f <mark>runn</mark>able
'Gang worker#0 (Parallel GC Threads)" os_prio=0 tid=0x0000003fe4005b800 nid=0x4082 runnable
'Gang worker#1 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4005d800 nid=0x4083 runnable
                                                                                                                                     runnable
'Gang worker#2 (Parallel GC
'Gang worker#3 (Parallel GC
                                            Threads)" os_prio=0 tid=0x0000003fe4005f000 nid=0x4084 runnable
Threads)" os_prio=0 tid=0x0000003fe40062800 nid=0x4085 runnable
                                            Threads)" os_prio=0 tid=0x000003fe40064800 nid=0x4086
'Gang worker#4 (Parallel GC
                                                                                                                                    runnable
                                            Threads)" os_prio=0 tid=0x000003fe40066000 nid=0x4087
'Gang worker#5 (Parallel <u>G</u>C
                                                                                                                                    runnable
                                            'Gang worker#6 (Parallel GC
                       (Parallel GC
'Gang worker#7
"Gang worker#8 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4006b800 nid=0x408a runnable
"Gang worker#9 (Parallel GC Threads)" os_prio=0 tid=0x0000003fe4006d000 nid=0x408b runnable
"Concurrent Mark-Sweep GC Thread" os_prio=0 tid=0x000003fe400f4000 nid=0x408e <mark>runn</mark>able
"Gang worker#0 (Parallel CMS Threads)" os_prio=0 tid=0x000003fe400f0000 nid=0x408c <mark>runn</mark>able
"Gang worker#1 (Parallel CMS Threads)" os_prio=0 tid=0x000003fe400f1800 nid=0x408d <mark>runn</mark>able
```

----结束

修改方式

● 方法一:

a. 如果GC线程占用CPU较高且系统CPU资源充足,可以通过增加GC回收线程数量,降低GC线程的回收压力。不同垃圾回收方法设置GC方法参数不尽相同,CMS 使用-XX:ParallelCMSThreads参数设置,Parallel使用参数-XX:ParallelGCThreads设置。其他垃圾回收算法参数设置可参考《JVM垃圾回收算法与参数配置》。

下面是将CMS的GC线程数量设置成2的示例:

/path/java -XX:ParallelCMSThreads=2 <other parameters>

- b. 名称带有"C1"或"C2"的线程一般为即时编译(JIT)线程,若此类线程占用CPU较高,首先可以适当增加参数-XX:CompileThreshold的值,减少触发JIT频率,C2模式下该参数默认值是10000,C1模式下默认为1500。将JIT阈值设置为20000:
 - # /path/java -XX:CompileThreshold=20000 <other parameters>

方法二:

调整编译线程数量,设置-XX:+CICompilerCountPerCPU=true,编译线程数依赖于处理器核数自动配置;设置-XX:+CICompilerCountPerCPU=false -XX:

鲲鹏性能优化十板斧 6 JVM 性能调优

+ClCompilerCount=N,强制设定总编译线程数为N。比如,将JIT线程数量设置为4:

/path/java -XX:+CICompilerCountPerCPU=false -XX:+CICompilerCount=4
<other parameters>

● 方法三:

可以直接把JIT关闭,设置上述参数-XX:+CICompilerCount=0即可。

□ 说明

● 增加线程数量会导致CPU利用率上升,上下文切换增多可能会引起业务性能下降,调整参数需要根据业务实际情况进行优化适配。

关闭JIT可能会对业务性能有影响。

6.4.3 调整线程堆栈大小

原理

x86系统上JVM默认线程堆栈大小是1MB,鲲鹏上默认线程堆栈大小是2MB,同样线程量鲲鹏会比x86多耗1倍内存。如果系统内存不充足,可以使用-Xss参数设置线程堆栈大小,减少内存耗用。

修改方式

使用 -Xss参数在java进程启动时修改线程堆栈大小,下面是把堆栈设置成1MB的示例:

/path/java -Xss1024k <other parameters>

□ 说明

堆栈大小如果设置太小,容易导致栈溢出影响业务代码运行,需根据业务函数调用特点进行评估。

鲲鹏性能优化十板斧 A libaio 实现参考



对于磁盘文件,文件的读取是同步的,导致线程读取文件时,属于阻塞状态。程序为了提升性能和磁盘的吞吐,程序会创建几个单独的磁盘读写线程,并通过信号量等机制进行线程间通信(同时带有锁);显然线程多,锁多,会导致更多的资源抢占,从而导致系统整体性能下降。

libaio提供了磁盘文件读写的异步机制,使得文件读写不用阻塞,结合epoll机制,实现一个线程可以无阻塞的运行,同时处理多个文件读写请求,提升服务器整体性能。下面是libaio+epoll的一个实现示例。

<u> 注意</u>

代码删除了变量初始化、异常判断等分支。

```
/* AIO的回调函数*/
void aio_callback(io_context_t ctx, struct iocb *iocb, long res, long res2)
  /* 失败时-res为errorno;成功时,res为真正读取的字节数;
    iocb->u.c.buf为读取的内容*/
*使用libaio和epoll,对同一个文件的若干个不同偏移位置发起异步读取请求,
* 如对文件的偏移位置0、1024*1、1024*2、、1024*127发起总共128个异步读取请求(读取1024字
节)
int libaio_epoll_read_file()
  efd = eventfd(0, EFD_NONBLOCK);
  fd = open(TEST_FILE, O_RDWR | O_CREAT | O_DIRECT, 0644);
  io setup(8192, &ctx);
  for (int i = 0; i < NUM EVENTS; i++) {
    posix_memalign(&buf, ALIGN_SIZE, RD_WR_SIZE);
    io_prep_pread(iocbs[i], fd, buf, RD_WR_SIZE, i * RD_WR_SIZE);
    io_set_eventfd(iocbs[i], efd);
    io_set_callback(iocbs[i], aio_callback);
  io submit(ctx, NUM EVENTS, iocbs);
  epfd = epoll create(1);
  epevent.events = EPOLLIN | EPOLLET;
```

鲲鹏性能优化十板斧 A libaio 实现参考

```
epevent.data.ptr = NULL;
epoll_ctl(epfd, EPOLL_CTL_ADD, efd, &epevent);
while (i < NUM_EVENTS) {
    epoll_wait(epfd, &epevent, 1, -1);
    read(efd, &finished_aio, sizeof(finished_aio);
    while (finished_aio > 0) {
        r = io_getevents(ctx, 1, NUM_EVENTS, events, &tms);
        for (j = 0; j < r; ++j) {
            // 调用对应事件的回调函数,处理数据
            ((io_callback_t)(events[j].data))(
            ctx, events[j].obj, events[j].res, events[j].res2);
        }
    }
}
```

鲲鹏性能优化十板斧 B 进入 BIOS 界面

B 进入 BIOS 界面

介绍如何进入BIOS Setup Utility界面。

操作场景

该任务指导用户在需要进行系统启动设置或系统信息查询的情况下,进入BIOS Setup Utility界面。

对系统的影响

该操作对系统正常运行无影响。

操作步骤

步骤1 连接好本地线缆并外接键盘、鼠标、显示器或开启iBMC Web管理的"远程控制"界面。

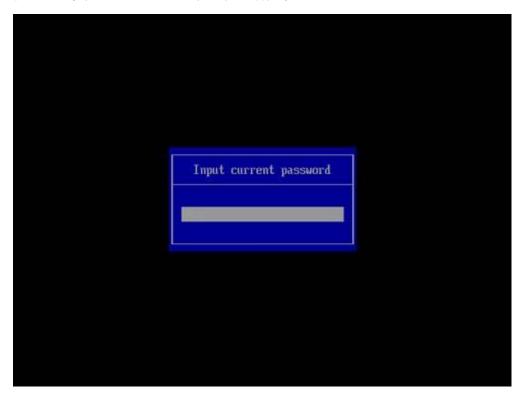
步骤2 将服务器上电。

步骤3 当出现如下界面时,按"Delete"进入BIOS Setup输入密码界面。

鲲鹏性能优化十板斧 B 进入 BIOS 界面



步骤4 在启动过程中,按"Delete"、"F11"(进入启动管理器)或"F12"(从网络启动快捷方式),均需要输入密码,请在对话框中输入密码。



步骤5 进入Setup Utility程序后,可以根据需要进行相关设置。

----结束

鲲鹏性能优化十板斧 C NEON lib 库应用加速

C NEON lib 库应用加速

原理

利用现有的一些基于NEON 技术开发的典型lib库,进行功能模块接口调用,达到应用程序加速目的。使开发者在获得最大效益的同时,而不必使用繁琐的NEON 汇编或NEON intrinsic函数进行编码加速。

修改方式

• Arm Compute Library(ACL)

ARM Compute Library是ARM公司发布的开源工程,旨在为图像/视频/多媒体/计算机视觉等领域的开发者提供arm平台的硬件加速库。这个库中分别用OpenCL与NEON的方式实现了一些上述领域的基本算法,OpenCL主要是arm的Mali GPU加速,NEON是针对arm的A系列CPU。该Lib库包含基础数学矩阵运算、图像处理基本运算、基础机器学习算法等功能模块。

参考链接: https://developer.arm.com/technologies/compute-library
Github 地址: https://github.com/ARM-software/ComputeLibrary

Ne10

Ne10是一个开源的C库,基于C接口的NEON汇编实现,由Arm托管在github上,包含一组最常见的处理密集型函数,这些函数基于Arm做了大量优化。Ne10采用的是模块化结构,由几个较小的库组成。主要包含领域(Math functions、Signal processing functions、Image processing functions、Physics functions)

参考链接: http://projectne10.github.com/Ne10/

Libyuv

libyuv是Google开源的实现各种YUV与RGB格式图像之间相互转换、图像旋转、缩放的基础库,其也是基于SIMD指令集进行开发的。

参考链接: https://code.google.com/p/libyuv/

skia

skia是一个开源2D图形库,用于谷歌Chrome和Chrome OS、Android、Mozilla Firefox和Firefox OS以及许多其他产品的图形引擎。

常用操作系统内存参数说明

常用的操作系统内存参数如下表所示:

优化项	优化项简介
dirty_background_bytes	触发pflush后台回写的脏内存字节数
dirty_background_ratio	触发pflush后台回写的脏内存所占百分比
dirty_bytes	触发一个写进程开始回写的脏内存字节数
dirty_ratio	触发一个写进程开始回写的脏内存所占百分比
dirty_expire_centisecs	适于pflush的脏内存的最小时间
dirty_writeback_centisecs	pflush活跃时间间隔
nr_hugepages	内存大页的数量
max_map_count	定义一个进程可以使用的最大内存映射区域数量
min_free_kbytes	设置期望的空闲内存数
overcommit_memory	0 = 利用探索法允许合理的过度分配; 1 = 始终过度分配,这可以提高内存密集型任务的性 能; 2 = 禁止过度分配
swappiness	内核倾向于使用换页来释放内存的程度

更多详情请参考Documentation/sysctl/vm.txt。

鲲鹏性能优化十板斧 E 修订记录

□ ■ 修订记录

发布日期	修订记录
2020-07-17	第四次正式发布
	● "2 CPU与内存子系统性能调优"中修改"2.3.5 调整内存页的大小"章节。
	● "3 网络子系统性能调优"中新增"3.3.6 单队列网卡中断散列" 和"3.3.7 TCP checksum优化"章节。
	● " 4 磁盘IO子系统性能调优 "中新增" 4.2.2 blktrace工具 "章 节。
	● "5 应用程序调优"中新增"5.2.8 NEON指令加速"章节。
	新增 "6 JVM性能调优"内容。
	● 新增 "C NEON lib库应用加速"和 "D 常用操作系统内存参数说明"章节。
2020-06-09	第三次正式发布
	修改 3.3.5 使用epoll代替select 中命令格式。
2020-05-15	第二次正式发布
	● 修改A libaio实现参考中的内容。
	● 设置透明大页中增加CentOS内核源码编译的内容。
2020-03-20	第一次正式发布