

鲲鹏性能优化十板斧

文档版本 09
发布日期 2024-06-07



版权所有 © 华为技术有限公司 2026。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



HUAWEI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暗示的声明或保证。

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。

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：

<https://www.huawei.com/cn/psirt/vul-response-process>

如企业客户须获取漏洞信息，请参见如下网址：

<https://securitybulletin.huawei.com/enterprise/cn/security-advisory>

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

2.1 调优简介.....4

2.2 常用性能监测工具.....5

2.2.1 top 工具.....5

2.2.2 Perf 工具.....7

2.2.3 numactl 工具.....9

2.3 优化方法.....10

2.3.1 修改 CPU 的预取开关.....10

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.....13

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

2.3.6 修改内存刷新速率.....15

3 网络子系统性能调优.....17

3.1 调优简介.....17

3.2 常用性能监测工具.....18

3.2.1 ethtool 工具.....18

3.2.2 strace 工具.....20

3.3 优化方法.....20

3.3.1 PCIe Max Payload Size 配置.....21

3.3.2 网络 NUMA 绑核.....21

3.3.3 中断聚合参数调整.....23

3.3.4 开启 TSO.....24

3.3.5 开启 LRO.....24

3.3.6 使用 epoll 代替 select.....25

3.3.7 单队列网卡中断散列.....27

3.3.8 TCP checksum 优化.....27

3.3.9 内核 CRC32 优化.....28

3.3.10 tuned 配置选择.....29

4 磁盘 I/O 子系统性能调优	31
4.1 调优简介	31
4.2 常用性能监测工具	32
4.2.1 iostat 工具	32
4.2.2 blktrace 工具	34
4.3 优化方法	35
4.3.1 调整脏数据刷新策略，减小磁盘的 I/O 压力	35
4.3.2 调整磁盘文件预取参数	36
4.3.3 优化磁盘 I/O 调度方式	37
4.3.4 文件系统参数优化	37
4.3.5 使用异步文件操作 libaio 提升系统性能	38
5 应用程序调优	39
5.1 调优简介	39
5.2 优化方法	39
5.2.1 优化编译选项，提升程序性能	39
5.2.2 文件缓存机制选择	40
5.2.3 执行结果缓存	41
5.2.4 减少内存拷贝	41
5.2.5 锁优化	42
5.2.6 使用 jemalloc 优化内存分配	43
5.2.7 CacheLine 优化	44
5.2.8 原子操作多核场景优化	45
5.2.9 热点函数优化	46
5.2.9.1 NEON 指令加速	46
5.2.9.2 软件预取	49
5.2.9.3 循环优化	51
5.2.9.4 数据布局优化	53
5.2.9.5 内联函数	54
5.2.9.6 OpenMP 并行化	56
5.2.9.7 SHA256 优化	58
5.2.9.8 乘除法优化	58
5.2.9.9 循环不变代码外提	59
5.2.10 毕昇编译选项优化	59
5.2.11 鲲鹏数学库	60
5.2.12 并行 I/O	60
6 JVM 性能调优	62
6.1 调优简介	62
6.2 常用性能监测工具	63
6.2.1 jstat 工具	63
6.2.2 jmap 工具	65
6.3 JVM 原理及配置建议	66
6.3.1 尽量使用高版本 JDK	66

6.3.2 设置 JVM 堆空间大小..... 67

6.3.3 选择合适的垃圾回收器..... 68

6.4 调优方法..... 71

6.4.1 GC 分析优化..... 71

6.4.2 JVM 线程优化..... 73

6.4.3 调整线程堆栈大小..... 74

A libaio 实现参考..... 75

B 进入 BIOS 界面..... 77

C NEON lib 库应用加速..... 79

D Linux 内核态 CRC32 算法优化..... 80

E 操作系统内核源码编译..... 83

F 常用操作系统内存参数说明..... 85

G 修订记录..... 86

1

简介

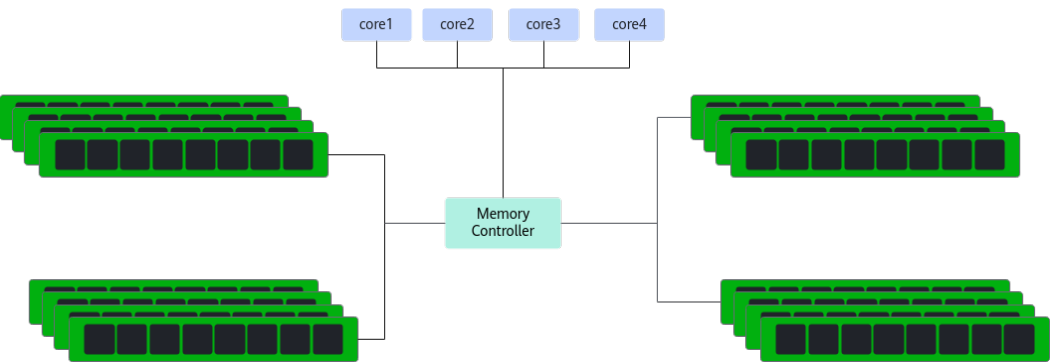
- 1.1 鲲鹏处理器NUMA简介
- 1.2 性能调优五步法

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，大量设备接入互联网、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地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，如图1-1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。

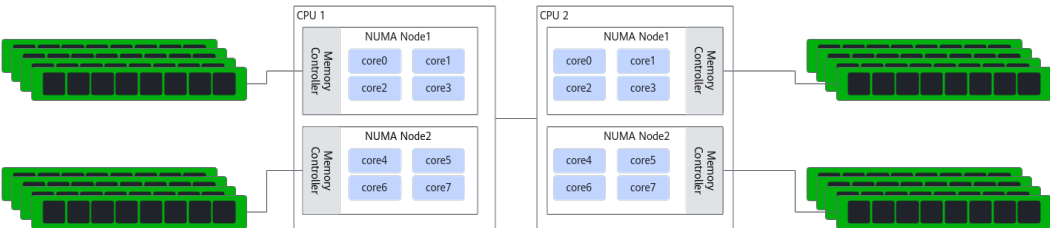
图 1-1 对称多处理器 SMP 架构



鲲鹏处理器支持NUMA（Non-uniform memory access，非统一内存访问）架构，能够很好地缓解SMP技术对CPU核数的制约。NUMA架构将多个核结成一个节点（Node），每一个节点相当于一个对称多处理机（SMP），一块CPU的节点之间通过On-chip Network通信，不同的CPU之间采用Hydra Interface实现高带宽低时延的片间通信，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，既能够获得很好的性能，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/I/O/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，才能更好地分析系统性能瓶颈，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，优化成果通常也很显著，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确定瓶颈。系统的瓶颈通常会在CPU过于繁忙、I/O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，在性能优化时应该首先花一点时间排查这些环节。

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，则需要将所有全部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I/O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，常见于磁盘、网络或应用程序自己的业务逻辑有休眠或信号等待状态，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3200MHz。

主要优化参数

优化项	优化项简介	默认值	生效范围	鲲鹏916处理器	鲲鹏920处理器
优化应用程序的NUMA配置	在NUMA架构下，CPU核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CPU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处理器	鲲鹏920处理器
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缓存中，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64KB	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64KB	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	调整应用的线程并发数，使充分利用多核能力与资源竞争之间达到平衡。	由应用本身决定	立即生效或重启生效（由应用决定）	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核资源使用情况。
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核。
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。

```
[root@localhost ~]# top
top - 19:23:49 up 3 days, 5:24, 4 users, load average: 3.17, 2.63, 2.80
Tasks: 704 total, 1 running, 338 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.8 us, 0.6 sy, 0.0 ni, 96.1 id, 0.5 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 26727008+total, 23552249+free, 25547456 used, 6200128 buff/cache
KiB Swap: 4194240 total, 4026176 free, 168064 used, 21798764+avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
22083	root	20	0	161.9g	7.1g	86208	S	132.0	2.8	215:35.41	impalad
21347	root	20	0	8871552	5.9g	38272	S	88.1	2.3	65:10.36	kudu-tserver
27953	root	20	0	118400	8448	3712	R	1.3	0.0	0:00.09	top
22761	root	20	0	118400	8448	3712	S	0.7	0.0	1:09.88	top
5709	root	20	0	433536	20480	10112	S	0.3	0.0	3:58.31	NetworkManager
11035	root	20	0	3344832	769984	35840	S	0.3	0.3	14:21.12	java
11202	root	20	0	3323072	818240	35840	S	0.3	0.3	15:34.67	java
12948	root	20	0	413120	46272	27584	S	0.3	0.0	3:07.55	statestored
12949	root	20	0	33.6g	1.7g	67776	S	0.3	0.7	11:26.51	catalogd
21248	root	20	0	820800	62976	36096	S	0.3	0.0	0:20.40	kudu-master
1	root	20	0	164480	16512	5824	S	0.0	0.0	0:17.23	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.38	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
5	root	20	0	0	0	0	I	0.0	0.0	0:00.03	kworker/u128:0
7	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq

- 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I/O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I/O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核的使用情况。

通过该命令可以查看单个CPU核的使用情况，如果CPU占用集中在某几个CPU核上，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,  0.0 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核上。

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核上运行，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 1:Def, 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)
* SHR     = Shared Memory (KiB)
* S       = Process Status
* %CPU    = CPU Usage
* %MEM    = Memory Usage (RES)
* TIME+   = CPU Time, hundredths
* COMMAND = Command Name/line
* PPID    = Parent Process pid
* UID     = Effective User Id
* RUID    = Real User Id
* RUSER   = Real User Name
* SUID    = Saved User Id
* SUSER   = Saved User Name
* GID     = Group Id
* GROUP   = Group Name
* PGRP    = Process Group Id
* TTY     = Controlling Tty
* TPGID   = Tty Process Grp Id
* SID     = Session Id
* nTH     = Number of Threads
* D       = Last Used Cpu (SMP)
* TIME    = CPU Time (secs)

top - 21:31:01 up 3 days, 7:52, 4 users, load average: 4.92, 8.62, 13.61
Tasks: 775 total, 1 running, 359 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.1 us, 0.0 sy, 0.0 ni, 97.4 id, 2.5 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2672768+total, 2472207+free, 18388784 used, 1658569 buff/cache
KiB Swap: 4194240 total, 4026176 free, 168064 used, 2273317+avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 44708 root        20   0 22144 11088  7936 S   7.2  0.9   0:04.40 cmake
 44712 root        20   0 1942592 631424 335936 D   5.9  6.2   0:02.54 ld.gold
 45286 root        20   0 156960 44480 17472 D   1.3  0.6   0:00.30 yum
 55542 root        20   0 110592 8512 3712 R   1.0  0.0   0:02.30 top
 21347 root        20   0 8871552 5.9g 38272 S   0.7 2.3 109:11.09 kudu-tserver
 5313 root         0   0 0 0 0 S   0.2  0.0   0:18.35 nfsaild/sdb5
 5709 root        20   0 433536 20480 10112 S   0.3  0.0   4:06.24 NetworkManager
11035 root        20   0 3344832 769984 35840 S   0.3  0.3 14:57.34 java
12940 root        20   0 33.6g 1.7g 60288 S   0.3  0.7 11:57.11 catalogd
21248 root        20   0 820880 62976 36996 S   0.3  0.0   0:39.74 kudu-master
 1 root         0   0 0 0 0 S   0.0  0.0   0:18.00 systemd
 2 root         0   0 0 0 0 S   0.0  0.0   0:00.42 kthreadd
 4 root         0 -20 0 0 0 I   0.0  0.0   0:00.09 kworker/0:0H
```

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。

```
[root@localhost ~]# top -p 22083 -H
top - 20:05:12 up 3 days, 6:06, 4 users, load average: 1.46, 2.44, 2.64
Threads: 357 total, 1 running, 356 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.6 us, 0.3 sy, 0.0 ni, 98.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2672768+total, 18573657+free, 68961728 used, 12571776 buff/cache
KiB Swap: 4194240 total, 4026176 free, 168064 used, 17138355+avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
29979 root        20   0 162.0g 46.4g 86336 R 99.0 18.2   3:35.07 impalad
22289 root        20   0 162.0g 46.4g 86336 S  6.2 18.2   0:01.23 rpc reactor-222
22083 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:03.29 impalad
22160 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:00.39 impalad
22161 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:00.14 impalad
22162 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:00.54 impalad
22163 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:00.92 impalad
22164 root        20   0 162.0g 46.4g 86336 S  0.0 18.2   0:00.95 impalad
```

----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内的调度事件。

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，从而有针对性地进行优化。

```
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] copy_page
 1.22% [kernel] [k] __ll_sc__cmpxchg_case_acq_4
 1.20% libc-2.17.so [.] libc_calloc
 1.17% [kernel] [k] el0_svc_naked
 1.11% [kernel] [k] _flush_cache_user_range
 1.00% libc-2.17.so [.] __int_malloc
 0.95% [kernel] [k] clear_page
 0.90% libc-2.17.so [.] __int_free
```

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。

```
[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			

----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，还用于将进程绑定到指定CPU核，绑定后由指定CPU核运行对应进程。

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 -H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核，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
```

步骤2 通过numactl将进程绑定到指定CPU核。

通过numactl -C 0-15 top命令将进程“top”绑定到0~15 CPU核上执行。

```
[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 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CPU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的值应保持在较低范围，如果过高，则应当考虑将进程绑定到特定CPU核。
- 结束

2.3 优化方法

2.3.1 修改 CPU 的预取开关

原理

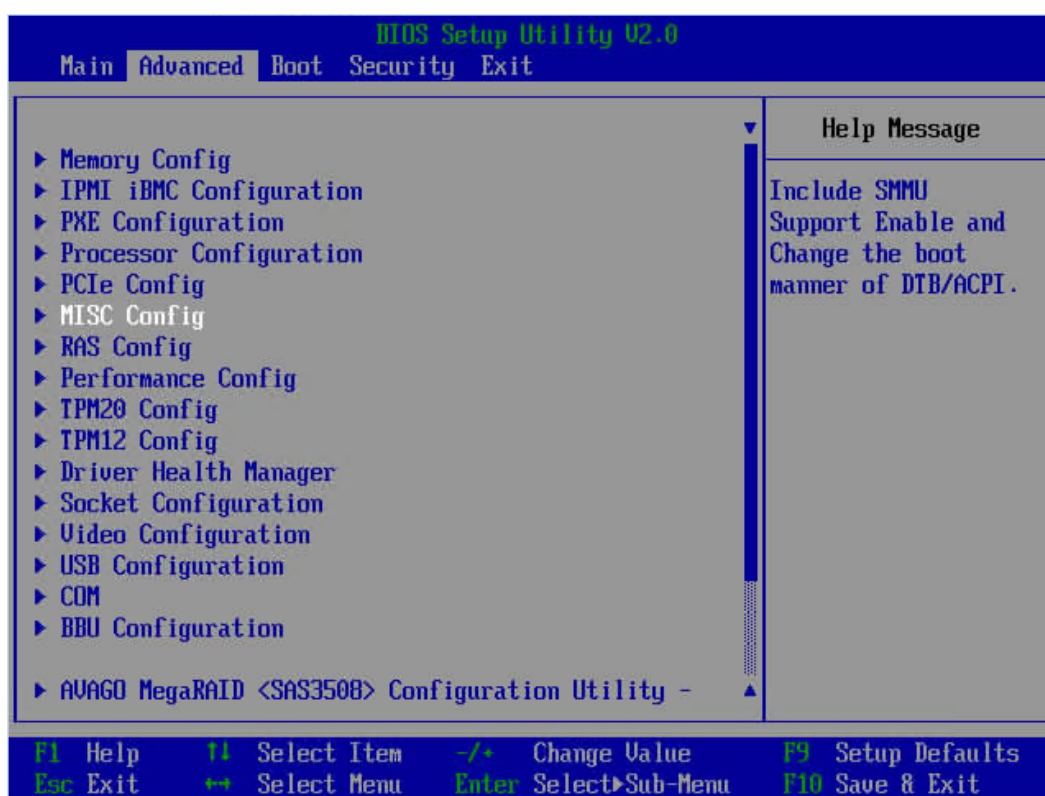
- 局部性原理分为时间局部性原理和空间局部性原理：
- 时间局部性原理（temporal locality）：如果某个数据项被访问，那么在不久的将来它可能再次被访问。
 - 空间局部性原理（spatial locality）：如果某个数据项被访问，那么与其地址相邻的数据项可能很快也会被访问。

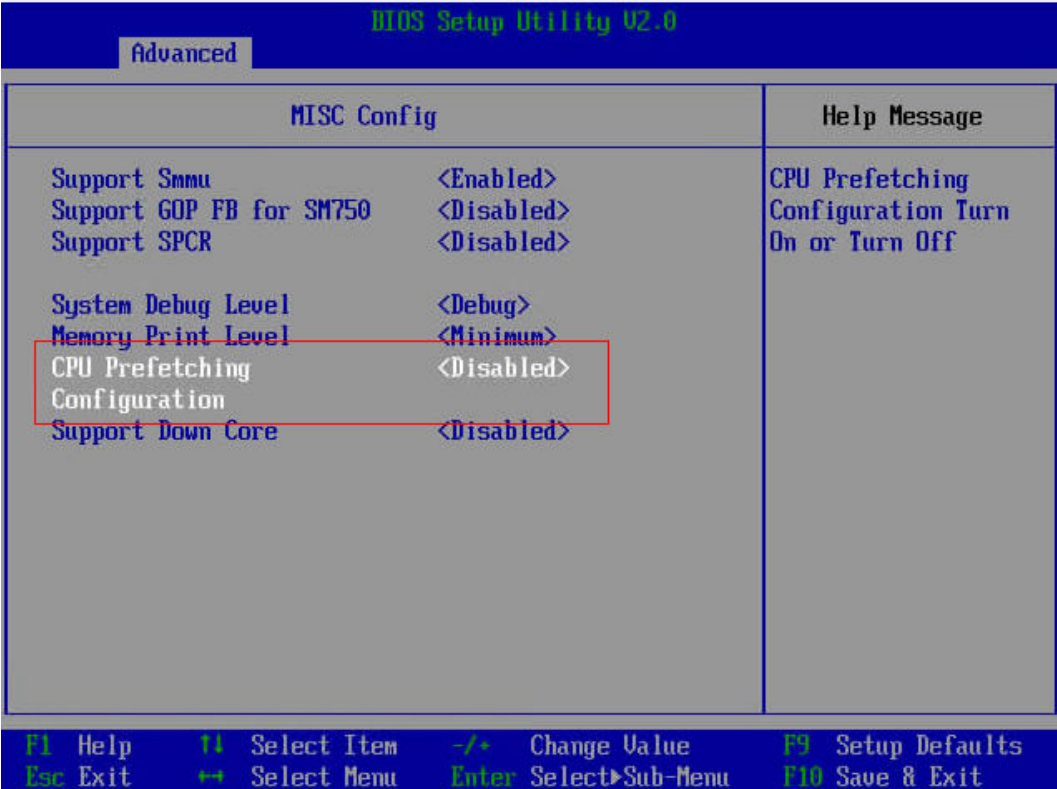
CPU将内存中的数据读到CPU的高速缓存时，会根据局部性原理，除了读取本次要访问的数据，还会预取本次数据的周边数据到Cache中，如果预取的数据是下次要访问的数据，那么性能会提升，如果预取的数据不是下次要访问的数据，那么会浪费内存带宽。

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，反之对于数据不集中的场景则需要关闭CPU预取。目前发现SPEC CPU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不需要定时器的時候，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
```

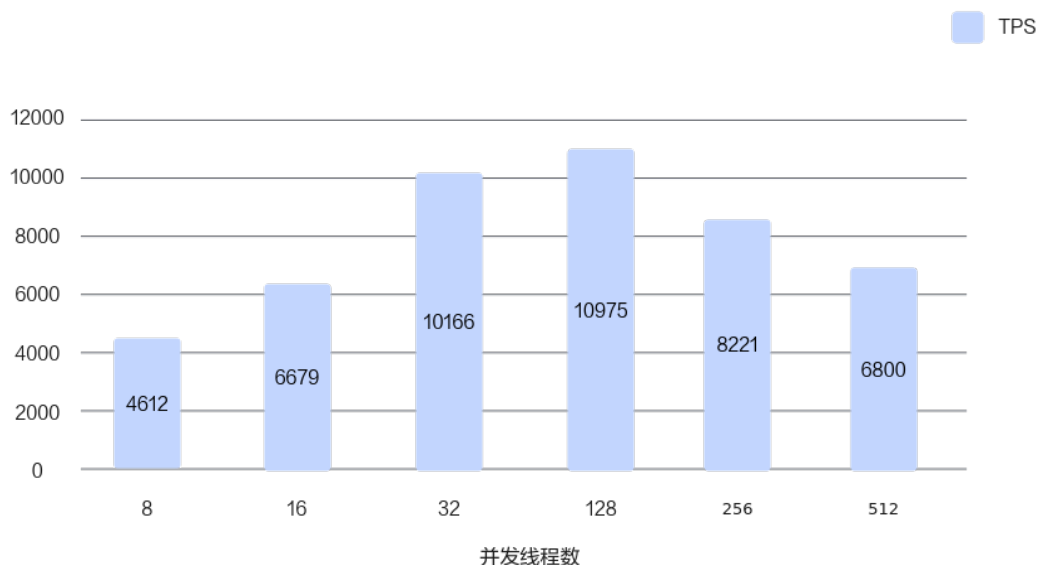
- 步骤1 在“/boot”目录下通过`find -name grub.cfg`找到启动参数的配置文件。
- 步骤2 在配置文件中将nohz=off去掉。
- 步骤3 重启服务器。

----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。当资源的竞争比较严重时，甚至会导致性能明显下降。以下数据为某业务场景下，不同并发线程数下的TPS，数据显示并发线程数达到128后，性能达到高峰，随后开始下降。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`参数设置进程数量。

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核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地避免跨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

- 可以通过如下方式绑定运行的CPU核，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核 28 到core 31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，即管理一个页大小的内存：

TLB管理的内存大小 = TLB行数 * 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1.21%为dTLB-load-misses占dTLB-loads的百分比，0.59%为iTLB-load-misses占iTLB-loads的百分比。

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**
- 设置PAGESIZE大小为64KB
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
```

操作系统还涉及更多的内存参数可以调整，具体参考[F 常用操作系统内存参数说明](#)。

2.3.6 修改内存刷新速率

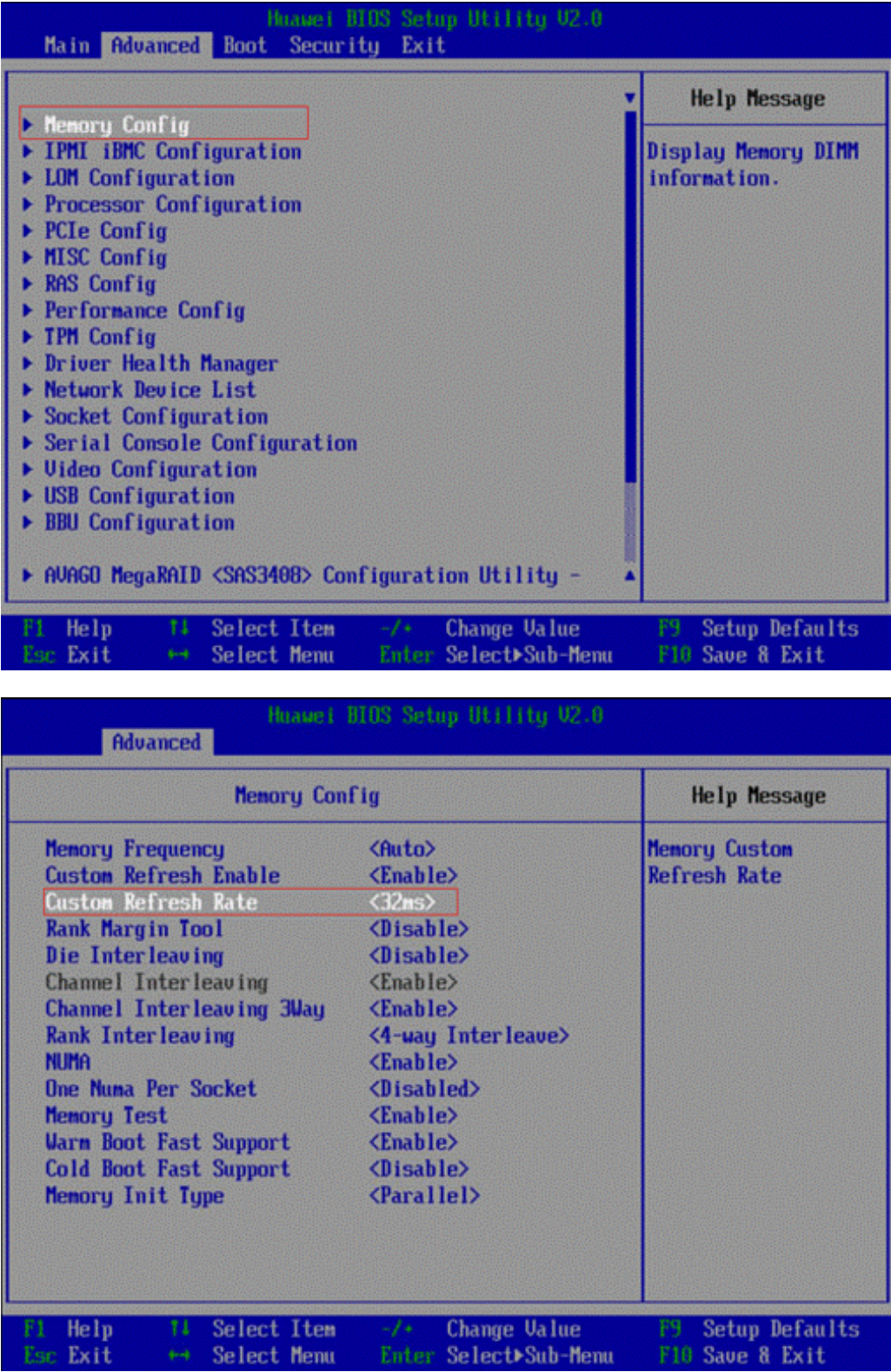
原理

DRAM存储器内部使用电容来存储数据，由于电容有漏电现象，经过一段时间电荷会泄放，导致数据不能长时间存储。因此需要不断充电，这个充电的动作叫做刷新。自动刷新是以“行”为单位进行刷新，刷新操作与读写访问无法同时进行，即刷新时会对内存的性能造成影响。同时温度越高电容泄放越快，器件手册通常要求芯片表面温度在0℃~85℃时，内存需要按照64ms的周期刷新数据，在85℃~95℃时，按照32ms的周期刷新数据。

BIOS中内存刷新速率选项提供了auto选项，可以根据工作温度自动调节内存刷新速率。相比默认的32ms配置，该选项可以提升内存性能，同时确保工作温度在85℃~95℃时内存数据的可靠性。

修改方式

按照附录B 进入BIOS界面的步骤进入BIOS，然后在BIOS的如下位置设置内存刷新速率。



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，减少跨节点内存访问的次数。即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处理器	鲲鹏920处理器
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
设置网卡队列数量	调整网卡队列数量	不同操作系统和网卡不同	立即生效	Y	Y

优化项	优化项简介	默认值	生效范围	鲲鹏916处理器	鲲鹏920处理器
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	Y	Y
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	Y	Y
开启TCP分段Offload（卸载）	将TCP的分段处理交给网卡处理	关闭	立即生效	Y	Y

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：

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	查询网卡中断信息。
-C	修改网卡聚合信息。
-l	查看网卡队列数量。
-L	设置网卡队列数量。

输出格式：

```
ethtool -k eth0
Features for eth0:
rx-checksumming: on
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数据包分段。
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：

参数	说明
-T	显示每一次调用所耗的时间。
-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。

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=281463144447425}}, {EPOLLIN|EPOLLOUT|EPOLLRDHUP, {u32=1052673241, u64=281463144482009}}, {EPOLLIN|EPOLLOUT|EPOLLERR|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	系统调用发生的时间。
epoll_pwait	系统调用的函数名。
(716…)	括号内的值为函数参数。
=5	系统调用的返回值。
<0.000038>	系统调用的执行时间。

3.3 优化方法

3.3.1 PCIe Max Payload Size 配置

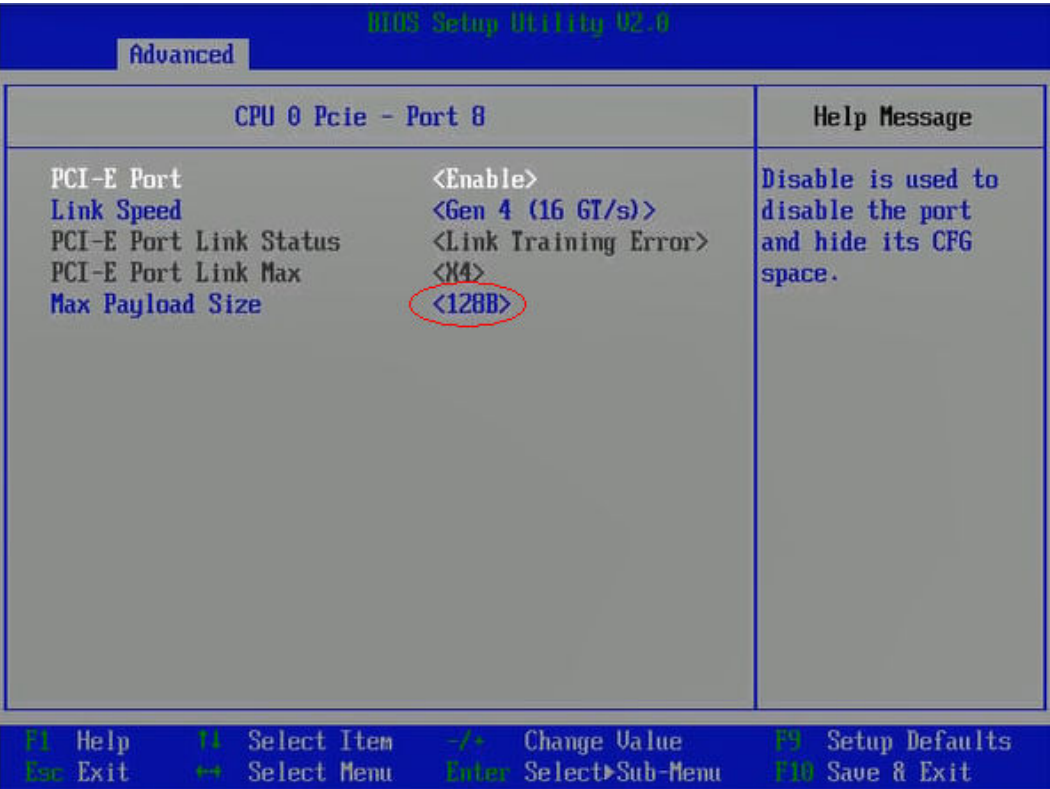
原理

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数据块大小（以字节为单位），它的大小与PCIe链路的传输效率成正比，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核处理，但是当处理中断的CPU核和网卡不在一个NUMA时，会触发跨

NUMA访问内存。因此，可以将处理网卡中断的CPU核绑定到网卡所在的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

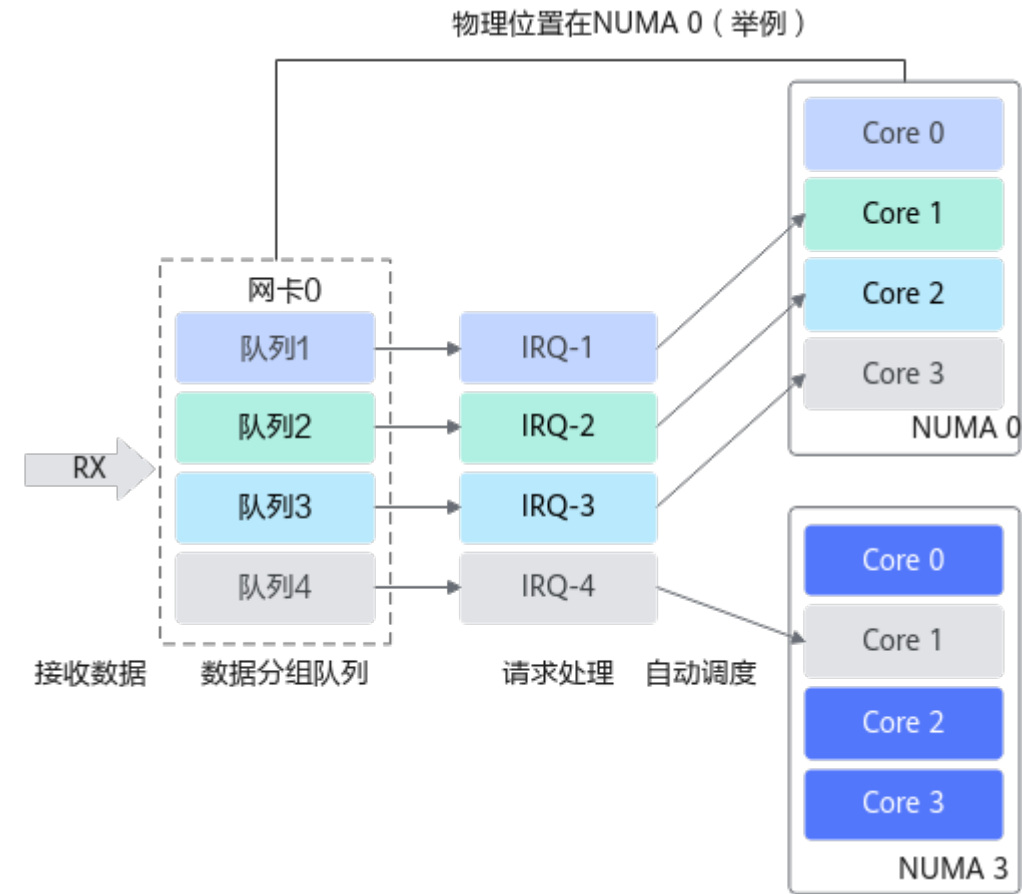
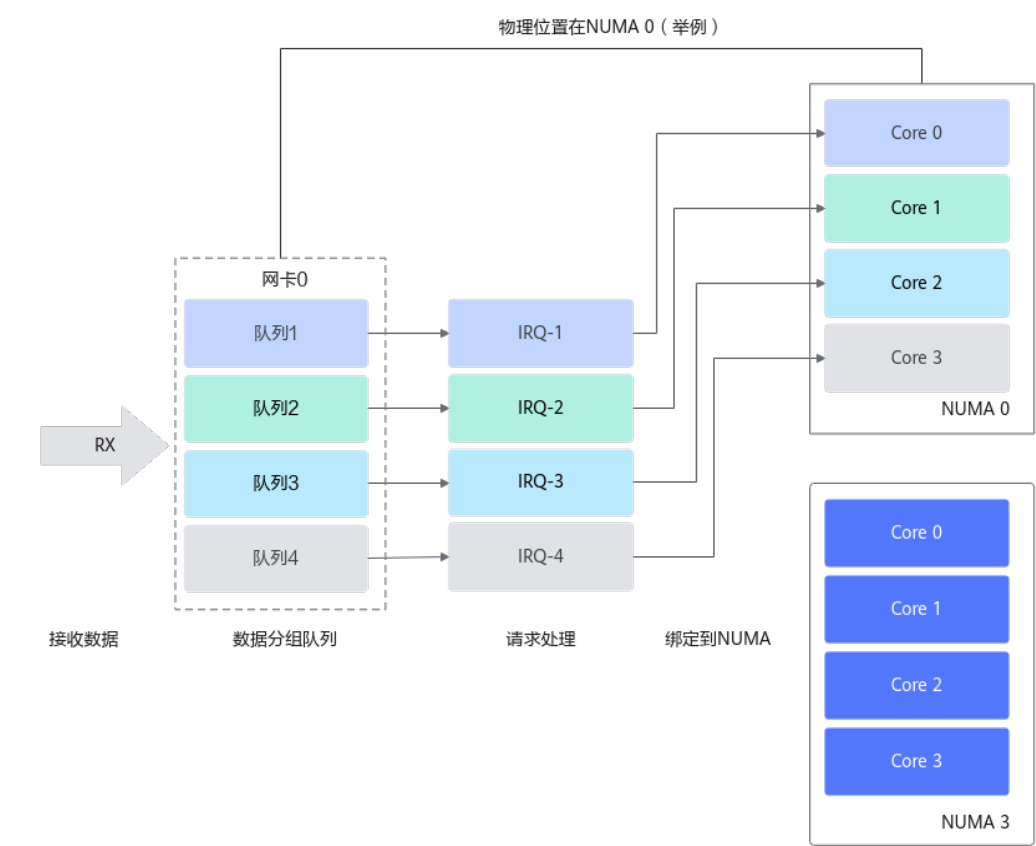


图 3-2 NUMA 绑定：中断绑定到指定核，避免跨 NUMA 访问内存



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 $ethx | 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cpuNumber是core的编号，从0开始。

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

----结束
- 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。
- 文档版本 09 (2024-06-07)

版权所有 © 华为技术有限公司

23

修改方式

使用ethtool -C \$eth命令调整中断聚合参数。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

当增大聚合度时，单个数据包的延时会以微秒的级别增加。

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的分段运算（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
```

3.3.5 开启 LRO

原理

LRO（Large Receive Offload），通过将接收到的多个TCP数据聚合成一个大的数据包传递给网络协议栈处理，减少上层协议栈处理开销，提高系统接收TCP数据包的能力。

该特性在存在大量网络小包I/O的情况下效果明显。

修改方式

执行如下命令查看网卡LRO功能是否开启：

```
ethtool -k Seth
```

```
[root@localhost ~]# ethtool -k enp4s0
Features for enp4s0:
rx-checksumming: on
tx-checksumming: on
    tx-checksum-ipv4: on
    tx-checksum-ip-generic: off [fixed]
    tx-checksum-ipv6: on
    tx-checksum-fcoe-crc: off [fixed]
    tx-checksum-sctp: on
scatter-gather: on
    tx-scatter-gather: on
    tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
    tx-tcp-segmentation: on
    tx-tcp-ecn-segmentation: off [fixed]
    tx-tcp-mangleid-segmentation: off
    tx-tcp6-segmentation: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off
```

开启网卡LRO功能：

```
ethtool -K Seth lro on
```

```
[root@localhost ~]# ethtool -K enp4s0 lro on
```

说明

开启LRO后，单个数据包的延时会增加，需结合业务综合评估是否需要开启。

3.3.6 使用 epoll 代替 select

原理

epoll机制是Linux内核中的一种可扩展I/O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，方便在一个线程中处理更多请求
- 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相关函数包括：epoll_create，epoll_ctl和epoll_wait。Linux-2.6.19又引入了支持屏蔽指定信号的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表示监测的文件句柄的最大个数。

步骤2 注册要监测的事件类型。

```
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);
```

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=281463144444784}}], 512, 1, NULL, 8) = 5 <0.000038>
```

3.3.7 单队列网卡中断散列

原理

RPS全称是Receive Packet Steering，从Linux内核版本2.6.35开始引入。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，表示将软中断散列到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，因为这是Linux内核官方推荐值。

3.3.8 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。

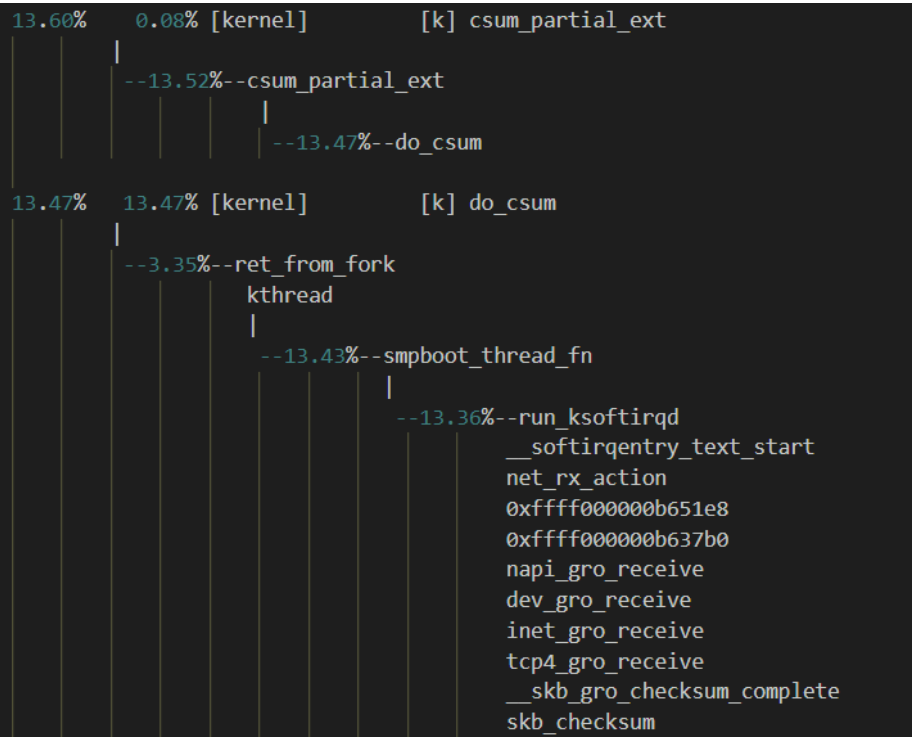
通常TCP checksum是由内核网络驱动来实现的，而且在5.6之前的内核版本是使用普通的算法实现。

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，且内核态do_csum函数热点占用过高的情况下，可以通过修改内核并合入checksum优化算法来提升性能。

网络通信中do_csum占用过高时的热点函数图如下图3-3所示：

图 3-3 do_csum 占用过高时的热点函数图



修改方式

5.6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](#)）。

3.3.9 内核 CRC32 优化

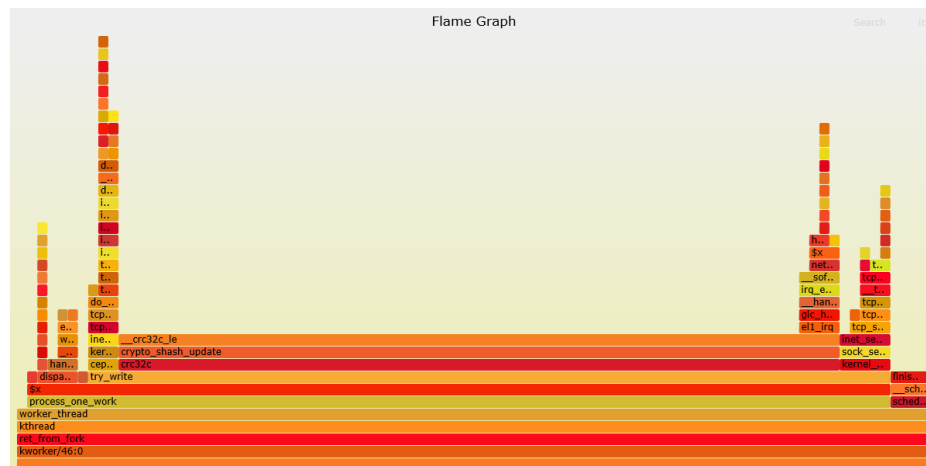
原理

CRC32算法在网络通信、存储等方面常用于数据正确性校验。以网络通信方面为例，在收到网络包之后，先通过CRC32算法计算校验值，并与收到的CRC32值做对比，以确定数据的完整性。

Linux内核（4.14）包含了CRC32算法的C语言实现，但是性能较低，可能会成为性能瓶颈。同时，在高版本内核（5.0以上版本），已经合入AArch64架构下的CRC指令实现。当内核态中CRC32函数调用占比很高，成为瓶颈时，可以考虑合入该实现，提升性能。

内核态CRC32占比高火焰图如图3-4所示。

图 3-4 内核态 CRC32 占比高火焰图



修改方式

修改方式可以参考：[D Linux内核态CRC32算法优化](#)。

修改后需要重新编译内核，详情可参考：[E 操作系统内核源码编译](#)。

3.3.10 tuned 配置选择

原理

tuned是针对Linux系统的一项服务，提供配置文件调整机制，可以根据系统状态调整系统配置，达到系统优化的目的。tuned-adm是配合tuned服务的工具，其预置了多种系统参数配置文件，通过调节调度时间、脏页刷新水位、CPU性能模式等参数来适应不同类型的业务。

用户可以根据自己的需要选择不同的配置，例如：

- 追求低时延，可以选择network-latency模式。
- 追求高网络I/O吞吐，可以选择network-throughput模式。
- 追求低I/O时延，可以选择latency-performance模式。
- 追求高I/O吞吐，可以选择throughput-performance模式。

修改方式

安装方法:

```
yum install tuned
```

tuned常用命令和常用模式如表3-1和表3-2所示:

表 3-1 tuned 常用命令

说明	操作
查看tuned服务运行状态	systemctl status tuned
查看当前运行的tuned模式	tuned-adm active

说明	操作
查看支持的tuned模式	tuned-adm list
设置所需tuned模式	tuned-adm profile <i>\$config</i>
不应用tuned模式配置	tuned-adm off

表 3-2 tuned 常用模式说明

配置名称	说明
balanced	均衡模式
latency-performance	低I/O延时模式
network-latency	低网络延时模式
network-throughput	高网络吞吐模式
throughput-performance	高I/O吞吐模式
powersave	节能模式

 说明

tuned工具所使用的配置文件位于“/usr/lib/tuned”目录，用户也可以参考其中的内核配置参数对系统进行优化。

4 磁盘 I/O 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、SSD、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，尽可能地将热点数据提前从磁盘中读取出来，减少CPU等待磁盘的时间浪费。因此部分优化手段实际上是围绕着如何更充分地利用Cache获得更好的I/O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916	鲲鹏920
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0.01秒）	立即生效	Y	Y
脏页占用总内存最大的比例	调整脏页占用总内存最大的比例（以memfree+Cached-Mapped为基准），提高PageCache命中率。	10%	立即生效	Y	Y

优化项	优化项简介	默认值	生效范围	鲲鹏916	鲲鹏920
脏页缓存占用总内存最大比例	调整脏页占用总内存最大的比例，避免磁盘写操作变为O_DIRECT同步，导致缓存机制失效。	40%	立即生效	Y	Y
调整磁盘文件预读参数	根据局部性原理，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	Y
磁盘I/O调度方式	根据业务处理数据的特点，选择合适的I/O调度器。	cfq	立即生效	Y	Y
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	Y

4.2 常用性能监测工具

4.2.1 iostat 工具

介绍

iostat是调查磁盘I/O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，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：

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。

参数	说明
-m	以M为单位显示。
-p	显示每个磁盘的详细情况。
-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次数，比如上面的样例中，是要求以1s为周期统计，总共统计100s。

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。如果值在统计周期内为非零，可能表明操作系统检测到逻辑上相邻或重叠的I/O请求，并通过合并优化减少实际下发到磁盘的I/O操作次数。这种合并通常意味着逻辑层面的连续访问模式，有利于提升HDD的吞吐量（ 减少寻址 ）或降低SSD的写入放大。
2. 如果%util接近100%，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，影响整个系统。这时可以考虑更换更高性能的磁盘，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` 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。`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 `await` 大于 `svctm` 差得过多的话一定是磁盘本身 I/O 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再查看，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，提供关于 I/O 子系统时间消耗的详细信息，从中可以分析是 I/O 调度慢还是硬件响应慢等。配套工具 `blkparse` 从 `blktrace` 读取原始输出，并产生用户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 I/O 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 I/O 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I/O层的请求时间间隔。
- Q2G：生成I/O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：I/O请求进入I/O 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：I/O请求在I/O Scheduler中等待的时间。
- D2C：I/O请求在driver和硬件上（I/O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I/O请求之后再把结果发回）所消耗的时间。
- Q2C：整个I/O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I/O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 I/O 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0.01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I/O集中，导致突发的I/O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`为脏页占用总内存最大的比例，超过这个值，系统不会新增脏页面，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，会导致系统性能下降。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，避免磁盘过早地进入到同步写状态。

说明

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

4.3.2 调整磁盘文件预取参数

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，则通过以下命令来查找。

```
find / -name read_ahead_kb
```

此参数的默认值为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 I/O 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

是早期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I/O资源，以此保证每个进程的I/O资源占用是公平的。这个算法在I/O压力大，且I/O主要集中在某几个进程的时候，性能不太好。

2. DeadLine，最终期限调度

这个调度算法维护了4个队列，分别为读队列、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I/O资源，适合I/O压力大且I/O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]
```

[]中为当前使用的磁盘I/O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，例如将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，可以达到提高性能的目的。

修改方式

假设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入磁盘介质，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，可先选择增加文件系统的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

由于文件读取是同步的，线程在读取文件时会处于阻塞状态。程序为了提升性能和磁盘的吞吐量，会创建多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，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地运行，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](#)。

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、缓存机制等优化。

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因素。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。

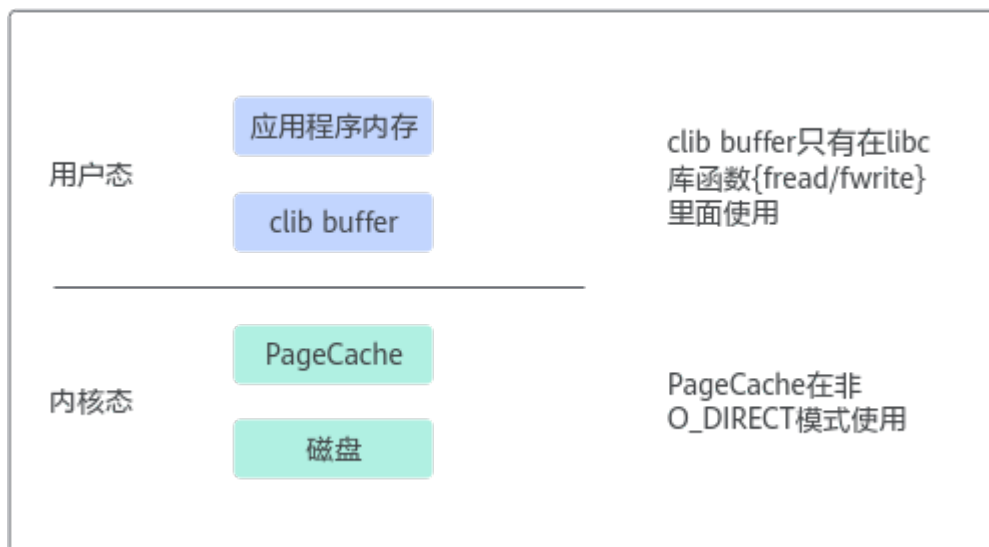
修改方式

- 在Euler系统中使用GCC编译器，可以在CFLAGS和CPPFLAGS中增加编译选项：
`-mtune=tsv110 -march=armv8-a`
- 在其它操作系统中，可以升级GCC版本到9.1.0，并在CFLAGS和CPPFLAGS中增加编译选项：
`-mtune=tsv110 -march=armv8-a`

5.2.2 文件缓存机制选择

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: 是用户态的一种数据缓存机制，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；当读取数据时，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: 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，但是因为缺少缓冲导致每次都是从磁盘里读取数据。
O_DIRECT主要适用场景为：应用程序有自己的缓存机制；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缓存

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无需去后端服务器取。

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中，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缓存不够，可以增加Java虚拟机ReservedCodeCacheSize参数。

3. MySQL查询缓存

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查询缓存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，本文列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将数据传输到内核的PageCache；内核将数据传输到应用态的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，内核再通过DMA将缓存搬运到网卡（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__ (
        "\n\t"
        "ldaxr %w[prev], %[ptr]          \n\t" // Load the previous value. ldaxr
        "cmp %w[prev], %w[old_value]    \n\t"
        "bne if                          \n\t" // goto end if not equal
        "stlaxr %w[temp], %w[new_value], %[ptr] \n\t" // Try to store the new value. stlaxr
        "cbrz %w[temp], 0b              \n\t" // Retry if it did not work.
        "1:                             \n\t"
        : [prev]="&r" (prev),
          [temp]="&r" (temp),
          [ptr]"&Q" (*ptr)
        : [old_value]"&r" (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核都会申请这个全局变量对应的锁，导致这个锁的竞争严重。可以基于业务逻辑，为每个CPU核或者线程分配对应的资源。
2. 使用ldaxr+stlxxr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指令并不能保证内存一致性，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+stlxxr指令比ldxr+stxr+dmb ish指令的性能高。
3. 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，从缓存中读取 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循环中会重复执行原子读、变量加以及原子写入操作，代码语句较为冗长。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+stlxxr 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可以在my.cnf文件中配置：`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，示意图如图5-1所示。虽然readHighFreq没有被改写，且存在于Cache中，但在发生伪共享时，仍需从内存中读。

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核中被改写后，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核标识为无效数据，虽然readHighFreq并没有被修改，但是CPU在访问readHighFreq时，依然会从内存重新加载，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

- 1. 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writeHighFreq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 原子操作多核场景优化

原理

LL/SC (Load-link/Store-condition) 原子指令需要把共享变量先加载到本核所在的L1 Cache中进行修改，在锁竞争少的情况下性能较好，锁竞争激烈时会导致系统性能下降严重。ARMv8.1规范中引入了新的原子操作指令扩展LSE (Large System Extensions)，将计算操作放到L3 Cache中进行。不仅扩大数据共享范围，减少Cache一致性耗时，还能在锁竞争激烈时可以提升锁的性能。

说明

在多核、原子锁竞争严重的情况下，建议在GCC编译选项中添加LSE相关选项，缓解锁竞争。

LL/SC指令(ldaxr&stlxr)：

```
GLIBCXX_ALWAYS_INLINE __int_type
fetch_add(__int_type __i,
memory_order __m = memory_order_seq_cst) noexcept
{ return __atomic_fetch_add(&_M_i, __i, int(__m)); }
0.00   ldr    x0, [sp,#40]
0.01   ldr    x1, [sp,#32]
16.28  2c:  ldaxr x2, [x0]
1.26   add    x3, x2, x1
81.60  stlxr w4, x3, [x0]
0.56   cbnz  w4, 2c
0.27   mov    x0, x2
_ZNSt13__atomic_baseIIEppEi():
{ return fetch_add(1); }
      add    sp, sp, #0x30
      ret
```

LSE指令 (ldaddal)：

```
memory_order __m = memory_order_seq_cst, noexcept
{ return __atomic_fetch_add(&_M_i, __i, int(__m)); }
0.00   ldr    x0, [sp,#40]
0.01   ldr    x1, [sp,#32]
0.00   ldaddal x1, x1, [x0]
99.97  mov    x0, x1
_ZNSt13__atomic_baseIIEppEi():
{ return fetch_add(1); }
      add    sp, sp, #0x30
      ret
```

修改方式

GCC 6.0以上版本支持（建议使用GCC 7.3.0以上版本）。

在编译选项中增加：

```
-march=armv8-a+lse
```

或者

```
-march=armv8.1-a
```

或者

```
-march=armv8.2-a
```

5.2.9 热点函数优化

5.2.9.1 NEON 指令加速

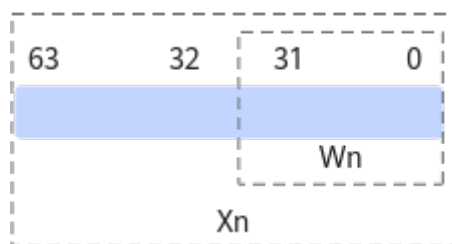
NEON是一种基于SIMD思想的技术，能够基于单条指令对多个数据同时进行操作，其使用的NEON指令类似于Intel CPU下的MMX/SSE/AVX指令，通过向量化的计算方式优化应用程序性能，通常应用于图像处理、音视频处理、数据并行处理等需要大量计算场景。

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。因此可视为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，同样也可视为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，以便尽可能地向量化。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];
    }
}
```

- d. 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）。
- e. 在满足需求情况下，使用较小的数据类型，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。
- f. 避免在循环中出现条件判断，尽量少用 `break` 跳出循环。
- g. 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。）
- h. 用数组下标来替代指针访问元素。
- i. 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>

5.2.9.2 软件预取

原理

数据预取通过将代码中后续可能使用到的数据提前加载到Cache中，减少CPU等待数据从内存中加载的时间，提升Cache命中率，进而提升软件的运行效率。预取指令格式通常如下：

```
PRFM prfop, [Xn|SP{, #pimm}]
```

1. prfop由type、<target>、<policy>三部分组成。
- type可选模式如下。

▪ PLD：数据预加载

▪ PLI：指令预取

▪ PST：数据预存储

- <target>可选模式如下。

▪ L1

▪ L2

▪ L3

L1、L2、L3分别表示对三个不同的Cache层级进行操作。

- <policy>可选模式如下。

▪ KEEP：数据预取使用后保存一定时间，适用于数据多次使用的场景。

▪ STRM：流式或非临时预取，数据使用后将淘汰，用于仅使用一次的数据。
2. Xn|SP通常表示64位通用寄存器或栈指针，使用场景中通常为预取的起始地址。
3. pimm是以字节为单位的偏移量，代表预取的字节长度，取值为8的整数倍，范围是0~32760，默认为0。预取长度可结合实际业务场景设定，尝试预取不同长度数据，获取最佳预取值。

从指令组成看，预取指令中核心部分为prfop，其决定了预取的类型、预取Cache层级以及预取的数据使用模式。本小节主要说明PLD数据预取，其他模式类似。数据预取核心指令部分如下表 [数据预取核心指令](#) 所示：

表 5-1 数据预取核心指令

数据预取指令	指令功能说明
PLDL1KEEP	数据预取到L1 Cache，策略为KEEP模式，数据使用后常驻Cache
PLDL2KEEP	数据预取到L2 Cache，策略为KEEP模式，数据使用后常驻Cache
PLDL3KEEP	数据预取到L3 Cache，策略为KEEP模式，数据使用后常驻Cache
PLDL1STRM	数据预取到L1 Cache，策略为STRM模式，数据使用后从Cache淘汰

数据预取指令	指令功能说明
PLDL2STRM	数据预取到L2 Cache，策略为STRM模式，数据使用后从Cache淘汰
PLDL3STRM	数据预取到L3 Cache，策略为STRM模式，数据使用后从Cache淘汰

GCC编译器针对预取也有对应的builtin函数实现，格式如下：

`_builtin_prefetch (const void *addr, int rw, int locality)`

- `addr`：数据的内存地址。
- `rw`：可选参数。`rw`可设置为0或1，0表示读操作，1表示写操作。
- `locality`：可选参数。`locality`可设置0-3（默认为3），表示数据在Cache中保持的时间，即时效性。取值为0表示访问的数据后续不再被访问，使用后在Cache中淘汰；取值为3表示访问的数据将再次访问；取值为1和2，则分别表示具有低时效性和中时效性。

更多关于预取指令的描述可参考ARM指令集手册：

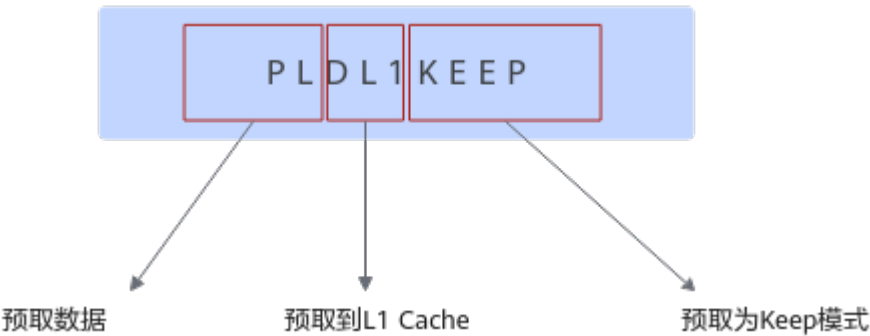
<https://developer.arm.com/documentation/ddi0596/2021-06/Base-Instructions/PRFM--immediate---Prefetch-Memory--immediate--?lang=en>

修改方式

进行数据预取时通常可以观察热点函数中LDR等数据加载指令的上下文代码，在代码中嵌入数据预取操作，常见的是在循环当中进行。在C/C++代码中，通常使用内嵌汇编形式调用预取指令，函数声明为inline形式，举例如下：

```
// 从ptr处预读取128字节数据
void inline Prefetch(int *ptr)
{
    __asm__ volatile("prfm PLDL1KEEP, [%0, #(%1)]"::"r"(ptr), "i"(128));
}
```

PLDL1KEEP指令组成如下：



PLDL1KEEP表示将数据预取到L1 Cache中，策略为KEEP模式。数据使用完后，将保留一定时间，适用于数据多次使用的场景。

以下示例代码功能为两数组对应元素相乘，通过每次预取多个数据，并将循环展开来对预取数据进行使用，提升计算性能，代码如下。

```
for (int i = 0; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayA[i] * arrayB[i];  
}
```

添加预取：

```
int i;  
Prefetch(&arrayA[0]);  
Prefetch(&arrayB[0]);  
for (i = 0; i < ARRAYLEN - ARRAYLEN % 4; i+=4) {  
    Prefetch(&arrayA[i + 4]);  
    Prefetch(&arrayB[i + 4]);  
    arrayC[i] = arrayA[i] * arrayB[i];  
    arrayC[i + 1] = arrayA[i + 1] * arrayB[i + 1];  
    arrayC[i + 2] = arrayA[i + 2] * arrayB[i + 2];  
    arrayC[i + 3] = arrayA[i + 3] * arrayB[i + 3];  
}  
for (; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayA[i] * arrayB[i];  
}
```

通过测试耗时分别为：使用预取耗时5569us，不使用预取耗时9359us，优化后代码性能显著提升。

5.2.9.3 循环优化

原理

循环优化是对程序中使用到的循环部分进行代码优化，合理的优化可以充分利用处理器的计算单元，提升指令流水线的调度效率，也可以提升Cache命中率。循环优化的方法有很多，如循环展开，循环融合，循环分离，循环交换，循环平铺等，根据具体业务代码选择使用。

修改方式

- 循环展开

循环展开是将循环体重复多次来减少循环，通常使用在小循环中。可以利用鲲鹏处理器具有多个指令执行单元的特点，增加计算密度，提升指令流水线的调度效率。如下是循环展开前后的代码对比。

修改前：

```
for (int i = 0; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayA[i] * arrayB[i];  
}
```

修改后：

```
int i;  
for (i = 0; i < ARRAYLEN - (ARRAYLEN % 4); i+=4) {  
    arrayC[i] = arrayA[i] * arrayB[i];  
    arrayC[i + 1] = arrayA[i + 1] * arrayB[i + 1];  
    arrayC[i + 2] = arrayA[i + 2] * arrayB[i + 2];  
    arrayC[i + 3] = arrayA[i + 3] * arrayB[i + 3];  
}  
for (; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayA[i] * arrayB[i];  
}
```

- 循环融合

循环融合是将相邻或紧密间隔的循环融合在一起，减少对循环变量的操作。迭代变量在循环体内被使用，可以提高Cache的使用率。合并小循环后也有利于增加鲲鹏处理器乱序执行的机会，提升指令流水线的调度效率。

修改前：

```
for (int i = 0; i < ARRAYLEN; i++) {  
    arrayA[i] = arrayB[i] + 3;  
}  
for (int i = 0; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayA[i] + 5;  
}
```

修改后：

```
for (int i = 0; i < ARRAYLEN; i++) {  
    arrayA[i] = arrayB[i] + 3;  
    arrayC[i] = arrayA[i] + 5;  
}
```

- 循环分离

循环分离主要针对较为复杂或计算密集的循环，将大循环拆分至多个小循环内执行，提升寄存器的利用率。

修改前：

```
for (int i = 1; i < ARRAYLEN; i++) {  
    arrayA[i] = arrayA[i-1] + 2;  
    arrayC[i] = arrayB[i] + 6;  
}
```

修改后：

```
for (int i = 1; i < ARRAYLEN; i++) {  
    arrayA[i] = arrayA[i-1] + 2;  
}  
for (int i = 1; i < ARRAYLEN; i++) {  
    arrayC[i] = arrayB[i] + 6;  
}
```

- 循环交换

循环交换是指交换循环的嵌套顺序，让内存访问尽可能满足局部性规则，提高Cache命中率。

修改前：

```
for (int j = 0; j < ARRAYLEN; j++) {  
    for (int i = 0; i < ARRAYLEN; i++) {  
        matrixC[i][j] = matrixA[i][j] + 3;  
    }  
}
```

修改后：

```
for (int i = 0; i < ARRAYLEN; i++) {  
    for (int j = 0; j < ARRAYLEN; j++) {  
        matrixC[i][j] = matrixA[i][j] + 3;  
    }  
}
```

- 循环平铺

循环平铺是指将一个循环拆分成一组嵌套循环，每个内部循环负责一个小数据块，以便最大化利用Cache中的现有数据，提高Cache命中率。通常针对比较大的数据集。

修改前：

```
for (int i = 0; i < n * ARRAYLEN; i++) {  
    for (int j = 0; j < n * ARRAYLEN; j++) {
```

```
matrixA[i][j] = matrixA[i][j] + matrixB[i][j];  
}  
}
```

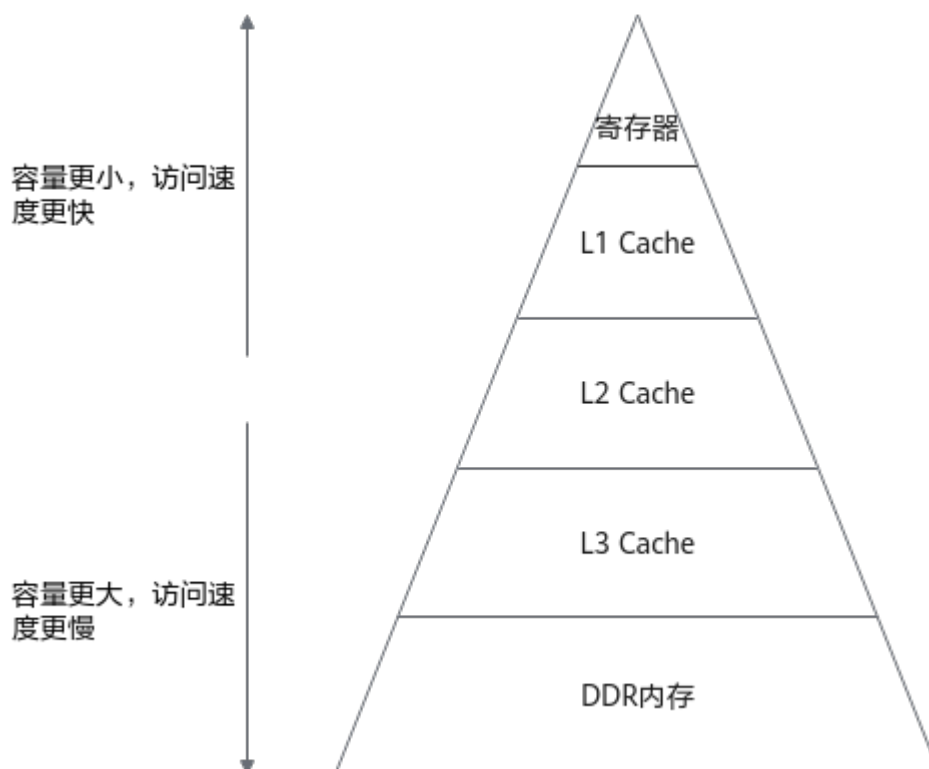
修改后：

```
// 假设定义一个合适的平铺块大小，这里取名为TILE_SIZE，可以根据实际缓存大小等因素调整  
#define TILE_SIZE 16  
for (int ii = 0; ii < n * ARRAYLEN; ii += TILE_SIZE) {  
    for (int jj = 0; jj < n * ARRAYLEN; jj += TILE_SIZE) {  
        for (int i = ii; i < ii + TILE_SIZE && i < n * ARRAYLEN; i++) {  
            for (int j = jj; j < jj + TILE_SIZE && j < n * ARRAYLEN; j++) {  
                matrixA[i][j] = matrixA[i][j] + matrixB[i][j];  
            }  
        }  
    }  
}
```

5.2.9.4 数据布局优化

原理

存储器各层次中，Cache速度快，面积小，当命中率高时可以提高处理器的数据访问速度。数据布局优化通过调整数据在内存中的排列，提升Cache和TLB命中率，以及CacheLine利用率，进而提升指令的执行效率，优化程序的时间开销。



常用优化方法如结构体布局优化，数据间重排，伪共享优化（见[5.2.7 CacheLine优化](#)）等。

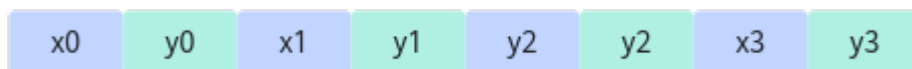
修改方式

- 关于结构体布局优化，下面以结构体数组和数组结构体两种数据组织方式来介绍。

- 结构体数组定义如下。

```
//结构体数组
struct Array{
    int x;
    int y;
};
struct Array stArray[N];
```

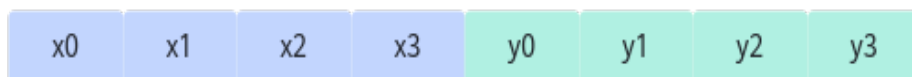
此时每一组x和y是连续存储的，在内存中的存储格式如下。



- 数组结构体定义如下。

```
//数组结构体
struct Array{
    int x[N];
    int y[N];
};
struct Array stArray;
```

此时x和y是分开存储的，在内存中的存储格式如下。



当业务场景主要针对x进行操作时，数组结构体中x加载到内存和Cache中的数据是连续的。相比于结构体数组，提高了CacheLine的有效性和Cache的命中率，从而提高性能。

- 数据间重排

以二维数组为例，当二维数组B按列与数组A相加，此时列元素在Cache中非连续的，不在一个CacheLine中，性能相对较差。通过对二维数组B重排，改为二维数组A与二维数组B的行元素相加，则可以从Cache中连续读取数据。重排前耗时544939us，重排后235722us。

按列读取：

```
for (int i = 0; i < ARRAYLEN; i++) {
    for (int j = 0; j < ARRAYLEN; j++) {
        arrayC[i][j] = arrayA[i][j] + arrayB[j][i];
    }
}
```

改为按行读取：

```
for (int i = 0; i < ARRAYLEN; i++) {
    for (int j = 0; j < ARRAYLEN; j++) {
        arrayC[i][j] = arrayA[i][j] + arrayB[i][j];
    }
}
```

5.2.9.5 内联函数

原理

对于频繁调用的小函数，函数调用的时间开销可能已经远高于函数本身的时间开销。这时可以通过在函数的定义前面增加inline关键字，将函数定义为内联函数来降低函数调用的开销，提高程序性能。

内联函数是在编译阶段将函数体嵌入到调用该函数的语句中。GCC编译器中函数内联的相关属性如下表 [GCC编译器中函数内联的相关属性](#) 所示。表格说明仅指定inline属性

的函数不一定内联，编译器会根据程序使用的编译选项和优化等级，同时结合实际情况，比如函数体大小、函数体内是否有递归、是否为可变数目参数等来决定是否允许内联展开。

表 5-2 GCC 编译器中函数内联的相关属性

属性名	属性说明
inline	建议编译器内联，但不一定内联
always_inline	强制内联
noline	防止考虑为内联函数

GCC内联相关的编译选项及可以支持的优化级别如下表 GCC内联相关的编译选项及可以使能的优化级别所示。

表 5-3 GCC 内联相关的编译选项及可以支持的优化级别

编译选项	优化级别	说明
early-inlining	-O0, -O1, -O2, -O3, -Os	针对两类函数（标记为always_inline的函数和函数执行体比调用开销小的函数），在编译器执行内联分析前完成内联。
inline-functions-called-once	-O1, -O2, -O3, -Os	考虑所有被调用一次的静态函数，即使它们没有被标记为内联，也可以内联到它们的调用者中。如果集成了对给定函数的调用，则该函数本身不会作为汇编代码输出。
inline-small-functions	-O2, -O3, -Os	当函数体小于预期的函数调用代码时，将函数内联到它们的调用者中（因此程序的整体大小变小）。编译器试探性地决定哪些函数足够简单，值得以这种方式集成。此内联适用于所有函数，甚至那些未声明为内联的函数。
inline-functions	-O2, -O3, -Os	考虑所有用于内联的函数，即使它们没有被声明为内联。编译器试探性地决定哪些函数值得以这种方式集成。 如果对给定函数的所有调用都被集成，并且该函数被声明为静态，则该函数本身通常不会作为汇编代码输出。

详细介绍请参见GCC手册：
<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

修改方式

修改前：

```
int func(int a, int b)
{
    return (a + b)*(a + b);
}
```

修改后：

```
/*建议内联*/
static inline int func(int a, int b)
{
    return (a + b)*(a + b);
}
```

或者：

```
/*强制内联*/
static inline
__attribute__((always_inline)) int func(int a, int b)
{
    return (a + b)*(a + b);
}
```

5.2.9.6 OpenMP 并行化

原理

OpenMP是一个应用程序编程接口（API），用于在不同平台上使用C、C++和Fortran语言进行独立于平台的共享内存并行编程。作为一个API，它在低级多线程原语上提供了一个高级抽象层。它的编程模型和接口可移植到不同的编译器和硬件体系结构，可在任意数量的CPU核心上扩展。因此，它适用的场景较为广泛，从具有几个CPU核心的台式PC到使用多达数百个核心的超级计算机中的计算节点。现代编译器（GCC 4.4+/Clang 3.8+）通常默认支持OpenMP，无需安装依赖库，使用方便。

OpenMP的基本理念是通过使用特殊的、类似注释的编译器指令（通常称为pragma）来增强顺序代码，向编译器提供如何并行化代码的提示。只需添加适当的pragma，就可以轻松地用于并行化现有的顺序代码。

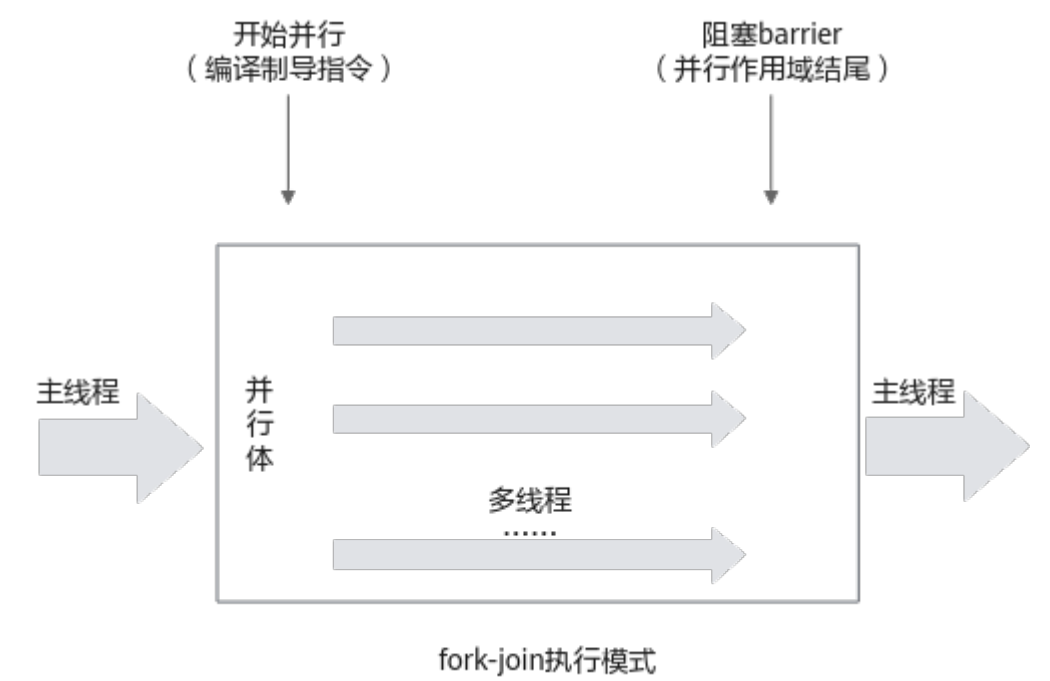
📖 说明

当遇到下面两种场景时，需要重新设计并行算法。

- 数据的处理顺序不能有依赖关系。并行化意味着同时处理不同的数据，如果此时处理的数据依赖于前一时刻处理的结果，逻辑上就不具备并行化的条件。
- 并行作用域中不存在竞争条件。OpenMP是一个在共享内存体系结构上运行的框架，每个线程都可以访问在并行作用域之外声明的任何变量或数组。如果多个线程同时操作一个数据，那么执行结果就会是随机的，加锁又会使性能下降，与并行的初衷背道而驰。

OpenMP采用fork-join的执行模式。开始只存在一个主线程，当需要进行并行计算时，OpenMP会派生出若干个分支线程来执行并行任务。当并行代码执行完成后，分支线程会合，并把控制流程交给单独的主线程。一个典型的fork-join执行模型如图5-2所示。

图 5-2 fork-join 执行模型



修改方式

OpenMP最常用的是并行化for循环，下面以矩阵乘法为例，简要介绍用法。

- 步骤1 包含OpenMP头文件，例如：`include <omp.h>`。
- 步骤2 在需要并行的for循环前，加上编译指导指令。OpenMP并行化测试用例如下图所示。

<pre>vector<vector<double>> a(SIZE, vector<double>(SIZE, 0)); vector<vector<double>> b(SIZE, vector<double>(SIZE, 0)); vector<vector<double>> ans(SIZE, vector<double>(SIZE, 0)); initialize_matrix(a, b); matrix_transpose(b); // serial matrix multiplication auto serial_start = steady_clock::now(); for (int i = 0; i < SIZE; ++i) { for (int j = 0; j < SIZE; ++j) { double sum = 0; for (int k = 0; k < SIZE; ++k) { sum += a[i][k] * b[j][k]; } ans[i][j] = sum; } } auto serial_end = steady_clock::now(); duration<double> time_serial = serial_end - serial_start; cout << "serial elapsed time:\t\t" << time_serial.count() << " \tseconds" << endl;</pre>	<pre>vector<vector<double>> a(SIZE, vector<double>(SIZE, 0)); vector<vector<double>> b(SIZE, vector<double>(SIZE, 0)); vector<vector<double>> ans(SIZE, vector<double>(SIZE, 0)); initialize_matrix(a, b); matrix_transpose(b); // openmp matrix multiplication auto openmp_start = steady_clock::now(); #pragma omp parallel for // 增加编译指导语句 for (int i = 0; i < SIZE; ++i) { for (int j = 0; j < SIZE; ++j) { double sum = 0; for (int k = 0; k < SIZE; ++k) { sum += a[i][k] * b[j][k]; } ans[i][j] = sum; } } auto openmp_end = steady_clock::now(); duration<double> time_openmp = openmp_end - openmp_start; cout << "openmp parallel elapsed time:\t" << time_openmp.count() << " \tseconds" << endl;</pre>
修改前	修改后

说明

并行作用域前定义的变量默认是公有的，并行作用域中定义的变量默认是私有的。紧跟指导语句的循环变量只能是私有的，检查并行作用域中的其他变量是否需要通过private显式私有化，避免出现竞争的情况。

在本例中，变量i、j、k、sum为私有变量，数组a、b、ans为不存在竞争的公有变量。

步骤3 编译程序。编译器对OpenMP API的支持由编译器标志-fopenmp指定，编译命令为：

```
g++ -O2 -std=c++11 -fopenmp matrix_multiplication.cpp -o matrix_multiplication
```

步骤4 执行程序。在运行时，默认线程数通常等于操作系统看到的逻辑CPU核心数。这里我们使用环境变量OMP_NUM_THREADS来修改它（也可以使用专用接口set_num_threads()或专用子句在源代码中进一步指定线程数量），执行的命令为：

```
OMP_NUM_THREADS=2 ./matrix_multiplication
```

----结束

本例在鲲鹏芯片上测试，当SIZE为3000时，串行执行时间56s。OpenMP两个线程并行执行，即0~1499与1500~2999次循环分别在两个线程中并行执行，执行时间为38s，性能提升显著。

5.2.9.7 SHA256 优化

原理

ARMv8指令集新增密码学指令，其中包含了AES加解密算法、SHA1和SHA256安全散列算法相关的硬件加速指令，可以显著提升计算性能。SHA256优化算法主要使用的指令如下表 [SHA256优化算法指令](#) 所示。

表 5-4 SHA256 优化算法指令

指令	功能	使用示例
SHA256H	SHA256哈希更新(part 1)	SHA256H q0, q1, v2.4s
SHA256H2	SHA256哈希更新(part 2)	SHA256H2 q0, q1, v2.4s
SHA256SU0	SHA256消息块更新0	SHA256SU0 v0.4s, v1.4s
SHA256SU1	SHA256消息块更新1	SHA256SU1 v0.4s, v1.4s, v2.4s

修改方式

使用sha256_aarch64.S文件替换原本的实现，并在sha256.c文件中调用原有sha256_compress函数的位置添加鲲鹏的sha256_compress函数。

具体实现方法请参见华为[鲲鹏论坛](#)。

5.2.9.8 乘除法优化

原理

CPU在处理不同指令的时候花费的指令周期是不同的，移位运算和加减运算只需要1个指令周期，而乘法运算需要3个指令周期，除法运算需要6~20个指令周期。因此应使用指令周期短的指令来实现相同功能，能够有效提高程序运行速度。

修改方式

- 使用移位运算替换乘除运算

修改前：

```
int a = 8;
int b = 2;
int c = a / b;
int d = a * b;
```

修改后：

```
int a = 8;
int c = a >> 1;
int d = a << 1;
```

- 使用乘法运算替换除法运算

修改前：

```
float a = x / 2.13;
```

修改后：

```
const float b = 1.0 / 2.13;
float a = x * b;
```

5.2.9.9 循环不变代码外提

原理

循环不变代码外提即结果不依赖于循环执行次数的表达式，放到循环执行前。可以显著减少循环中的代码运算量，提高代码运行速度。编译器可以对部分代码进行不变代码外提优化，如果表达式存在指针或者引用，则编译器不会进行自动优化。

修改方式

修改前：

```
void fun(int arrayA [], int *p) {
    for (int i = 0; i < ARRAYLEN; ++i) {
        arrayA [i] = *p + *p + i;
    }
}
```

修改后：

```
void fun(int arrayA [], int *p) {
    int tmp = *p + *p;
    for (int i = 0; i < ARRAYLEN; ++i) {
        arrayA [i] = tmp + i;
    }
}
```

5.2.10 毕昇编译选项优化

毕昇编译器是针对鲲鹏平台的高性能编译器。它基于开源LLVM开发，除LLVM通用功能和优化外，毕昇编译器的工具链对中端及后端的关键技术点进行了深度优化。利用编译器的优化能力可以提高程序的运行性能。

- 毕昇编译器支持jemalloc库的使用，jemalloc是一个通用的malloc实现，着重于减少内存碎片和提高并发性能，以动态库的方式存放于毕昇编译器工具链中。

使用方式：

增加编译选项-ljemalloc

- 毕昇编译器支持LTO（链接时优化），LTO是对整个程序的分析和跨模块的优化，同时还可以消除无用代码。但是会带来编译时内存占用变高和编译时间变长的问题。

使用方式：

增加编译选项-flto

毕昇还有其他的编译选项用于程序优化，包括毕昇自定义的优化选项。详情请参见《[毕昇编译器 用户指南](#)》。

5.2.11 鲲鹏数学库

鲲鹏数学库（Kunpeng Math Library，以下简称KML）提供了基于鲲鹏平台优化的高性能数学函数，所有接口由C/C++、汇编语言实现，部分提供Java语言封装的接口。KML基于鲲鹏架构，通过向量化、数据预取、编译优化、数据重排、内联汇编、算法优化等手段优化代码，以获得较高的运行性能。

KML包含多种数学计算的数学库，分别为基础线性代数运算数学库、稀疏基础线性代数运算库、向量数学库、基础运算数学库、快速傅里叶运算库、线性代数运算库、短向量数学库、稀疏迭代求解库，还包括基于JNI技术，用Java语言封装的KML数学库。官网提供详细的解释说明，安装使用方法请参见《[鲲鹏数学库 开发指南（KML）](#)》。

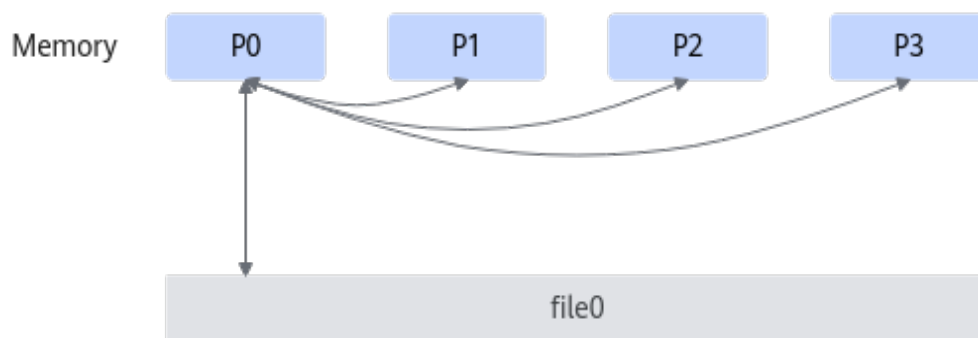
5.2.12 并行 I/O

原理

并行I/O可以使多个进程同时进行I/O操作，在HPC应用中有大量的I/O操作，并行I/O能够提高程序运行速度。并行I/O有三种模式：

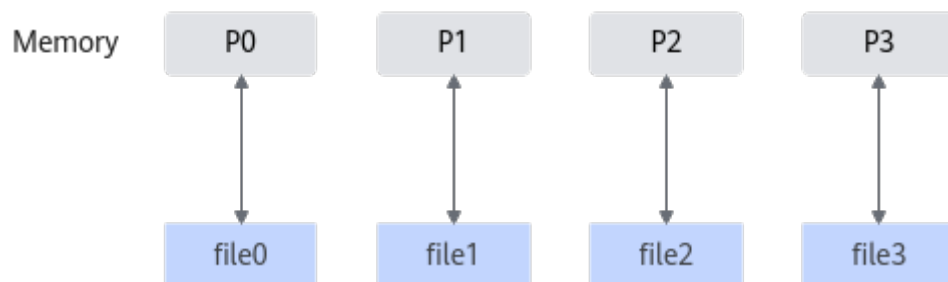
- 只有一个进程读写

一个进程（P0）将文件中的所有数据读入自己的缓冲区（buffer），然后用MPI发送和接收函数将大部分数据传递给其他进程。计算结束后，其他进程将计算结果传给进程P0，P0负责将所有数据结果写到文件。这个模式下负责读写文件的进程是性能瓶颈，读写带宽受限于P0所在计算服务器的网络带宽、存储系统的单进程性能上限。



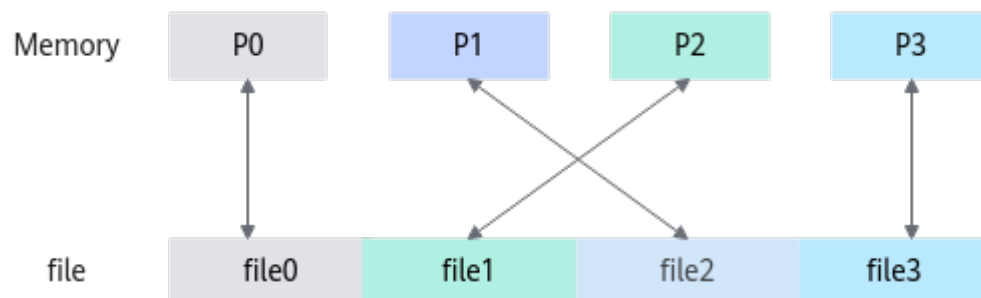
- 多个进程分别读写

每个进程只操作自己的文件，各进程之间不协调，相互独立。这种模式既能同时使用计算服务器的多个网络通道，又能发挥并行存储系统的多客户端接入能力。缺点是供读取的源数据文件可能没有进程数量多，造成负载不均，输出的文件数据太多，后续处理困难。



- 多个进程读写同一个文件

多个进程相互配合，避免无用操作。该模式下需要每个进程计算文件偏移量，避免数据冲突。这种模式下并行I/O性能可能达到最好。



修改方式

并行I/O有四种接口：POSIX I/O，MPI I/O，HDF5 I/O，NetCDF-4 I/O（Parallel-NetCDF），需要根据HPC应用支持的并行I/O进行对应修改。这里以Parallel-NetCDF为例，Parallel-NetCDF是一个使用MPI-I/O和一个定制版本的NetCDF API来实现高性能I/O的库，可以通过启用Parallel-NetCDF库来提升I/O性能，常应用于气象、海洋、环境等领域。

修改步骤如下：

步骤1 下载安装Parallel-NetCDF，方法参考链接：<https://github.com/Parallel-NetCDF/PnetCDF>。

步骤2 执行以下命令设置PNETCDF环境变量。

```
export PATH=/path/to/PNETCDF/bin:$PATH
export LD_LIBRARY_PATH=/path/to/PNETCDF/lib:$LD_LIBRARY_PATH
```

步骤3 编译应用软件时通过设置CPPFLAGS和LDFLAGS将PNETCDF链接到应用软件。

```
export PNETCDF=/path/to/PNETCDF
export CPPFLAGS="-I$PNETCDF/include"
export LDFLAGS="-L$PNETCDF/lib -lpnetcdf"
```

----结束

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 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：

1. 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；
2. 其次根据实际业务场景和硬件资源为JVM选择合理的堆空间；
3. 最后要选择合适的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	Y	Y

优化项	优化项简介	默认值	生效范围	鲲鹏916	鲲鹏920
-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	Y	Y
-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	Y	Y
-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	Y

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 and 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，需指定查看进程的PID和所选参数。

命令格式： jstat [参数] <pid>

一级参数主要功能如下：

参数	说明
jstat -class <pid>	显示加载class的数量，及所占空间等信息。
jstat -compiler <pid>	显示虚拟机实时编译的数量等信息。
jstat -gc <pid>	显示GC的信息，查看GC的次数及时间。
jstat -gccapacity <pid>	显示虚拟机内存中三代对象的使用和占用大小。
jstat -gcutil <pid>	显示GC的统计信息。

参数	说明
jstat -gcnew <pid>	显示年轻代对象的信息。
jstat -gcnewcapacity <pid>	显示年轻代对象信息及其内存占用情况。
jstat -gcold <pid>	显示老年代对象的信息。
jstat -gcpermcapacity <pid>	显示永久代对象的信息及其占用量。
jstat -printcompilation <pid>	显示当前虚拟机的执行信息。

GC优化中，利用jstat -gc <pid>被频繁使用，其输出参数及含义如下：

参数	说明
S0C	年轻代中第一个survivor区的容量（字节）
S1C	年轻代中第二个survivor区的容量（字节）
S0U	年轻代中第一个survivor区目前已使用空间（字节）
S1U	年轻代中第二个survivor区目前已使用空间（字节）
EC	年轻代中Eden区的容量（字节）
EU	年轻代中Eden区目前已使用空间（字节）
OC	老年代的容量（字节）
OU	老年代目前已使用空间（字节）
PC	Perm（永久代）的容量（字节）
PU	Perm（永久代）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老年代（全GC）GC次数
FGCT	从应用程序启动到采样时老年代（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
S0C  S1C  S0U  S1U  EC    EU    OC    OU    MC    MU    CCSC  CCSU  YGC  YGCT
FGC  FGCT  GCT
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
```

6.2.2 jmap 工具

介绍

jmap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并查看堆空间分配等信息，是Java性能调优常用工具之一。

安装方式

完整安装JDK后自带jmap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：

参数	说明	举例
-dump: [live,]format=b,file=/you/ path/filename.hprof <pid>	输出JVM的堆对象内容到指定文件。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，那么只输出存活对象到文件。	jmap - dump:format=b,file=/opt /log/java12123.hprof 12123
-finalizerinfo <pid>	输出正等候回收的对象的 信息。	jmap -finalizerinfo 12123
-heap <pid>	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。	jmap -heap 12123
-histo[:live] <pid>	输出当前class的实例数 目、内存占用、类全名信 息。	jmap -histo:live 12123 head -n 20
-permstat <pid>	打印classloader和JVM heap中perm代的信息， 包含每个classloader的名 字、活跃性、地址、父 classloader和加载的class 数量等信息。	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
NewSize          = 268435456 (256.0MB) ##新生代分配大小
MaxNewSize       = 268435456 (256.0MB) ##最大新生代可分配大小
OldSize          = 5439488 (5.1875MB) ##老年代大小
NewRatio         = 2 ##新生代比例
SurvivorRatio    = 8 ##新生代与survivor的比例
PermSize         = 134217728 (128.0MB) ##perm区大小
MaxPermSize      = 134217728 (128.0MB) ##最大可分配perm区大小

Heap Usage: ##堆使用情况
New Generation (Eden + 1 Survivor Space): ##新生代 ( Eden区 + survivor空间 )
capacity = 241631232 (230.4375MB) ##Eden区容量
used     = 77776272 (74.17323303222656MB) ##已经使用大小
free     = 163854960 (156.26426696777344MB) ##剩余容量
32.188004570534986% used ##使用比例
Eden Space: ##Eden区
capacity = 214827008 (204.875MB) ##Eden区容量
used     = 74442288 (70.99369812011719MB) ##Eden区使用
free     = 140384720 (133.8813018798828MB) ##Eden区当前剩余容量
34.65220164496263% used ##Eden区使用情况
From Space: ##survivor1区
capacity = 26804224 (25.5625MB) ##survivor1区容量
used     = 3333984 (3.179534912109375MB) ##survivor1区已使用情况
free     = 23470240 (22.382965087890625MB) ##survivor1区剩余容量
12.43827838477995% used ##survivor1区使用比例
To Space: ##survivor2 区
capacity = 26804224 (25.5625MB) ##survivor2区容量
used     = 0 (0.0MB) ##survivor2区已使用情况
free     = 26804224 (25.5625MB) ##survivor2区剩余容量
0.0% used ## survivor2区使用比例
concurrent mark-sweep generation: ##老年代使用情况
capacity = 1879048192 (1792.0MB) ##老年代容量
used     = 30847928 (29.41887664794922MB) ##老年代已使用容量
free     = 1848200264 (1762.5811233520508MB) ##老年代剩余容量
1.6416783843721663% used ##老年代使用比例
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, OpenJDK是官方JDK的开源版本, 两者核心源码和技术一致, OpenJDK更新发布更加频繁, 目前使用用户数量正在持续增加。毕昇JDK是OpenJDK的一个发行版本, 基于OpenJDK对鲲鹏芯片和Euler系统进行了针对性优化。

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, 可选择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OpenJDK版本，请到如下网址获取安装：<http://openjdk.java.net/>
- 使用毕昇JDK版本，请到如下网址获取安装：<https://www.hikunpeng.com/developer/devkit/compiler>。

修改方式

建议使用高版本JDK，并重新安装。

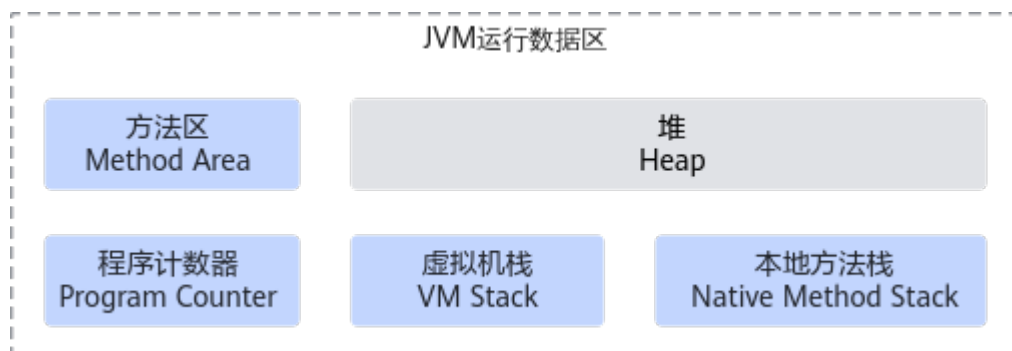
查询JDK版本的方式，执行：

```
java -version
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是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. 虚拟机栈存储的是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区域，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/To Survivor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；当物理内存大于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：
- 1. 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）。

修改方式

在Java应用程序启动时，添加如下参数并设置大小：

参数	说明
-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）与老年代的比值（不包括永久代）。如设置为4，则年轻代与老年代所占比值为1:4，年轻代占整个堆的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=	设置永久代堆大小。

举例：

```
java -Xmx3600m -Xms3600m -Xmn2g -Xss128k
```

设置最大堆空间为3600MB，初始化栈大小为3600MB，年轻代大小为2GB，线程栈大小为128KB。

6.3.3 选择合适的垃圾回收器

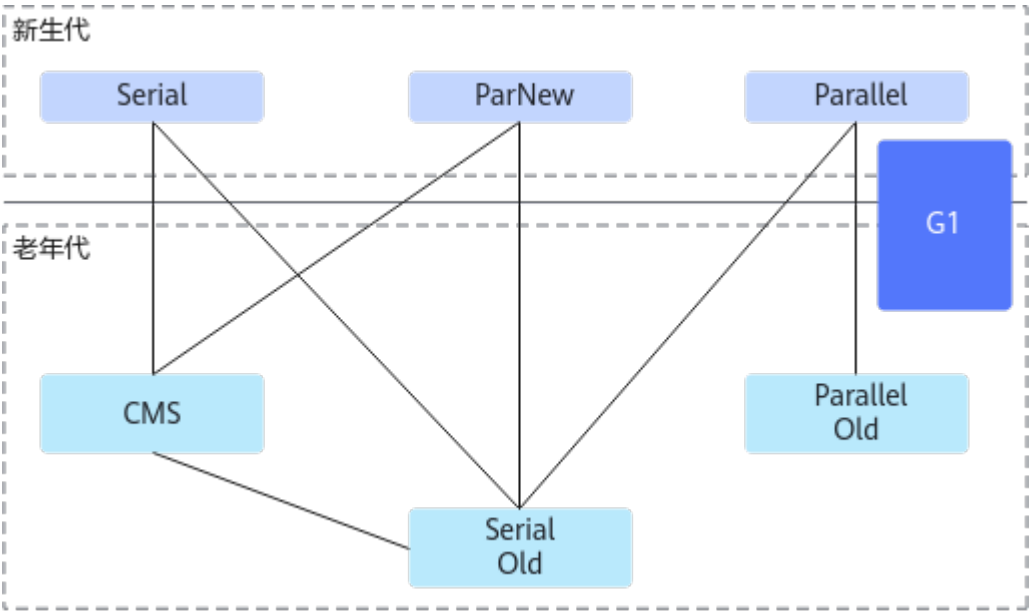
原理

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
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)	并发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运行数据的监测来决定。

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，JDK 1.5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。JDK 1.5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。
- 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。
- 对于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等于CPU核数，可以通过-XX:ParallelGCThreads来设置垃圾收集的线程数。
Parallel Scavenge	-XX:+UseParallelGC JDK 1.7、JDK 1.8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，通过-XX:MaxGCPauseMillis设置收集器完成内存回收的时长。 • Parallel Scavenge的目标是达到一个可控制的吞吐量，通过-XX:GCTimeRatio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:+UseConMarkSweepGC	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:UseCMSCompactAtFullCollection参数控制是否开启碎片整理功能（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。

名称	修改参数	特点
Parallel Old	- XX:+UseParallelOld GC JDK 1.7、JDK 1.8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 JDK 1.7以后才提供，JDK 1.9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。

6.4 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1s打印的GC信息，内容只保留了GC频率和耗时相关的列信息，其他列未列出。jstat命令使用方法和输出信息说明，请参考[6.2.1 jstat工具](#)。

YGC	YGCT	FGC	FGCT	GCT
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
579	22.233	2	0.090	22.323
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
580	22.262	2	0.090	22.352
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
581	22.282	2	0.090	22.372
582	22.282	2	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
582	22.300	2	0.090	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. 通过上面的监控信息，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8s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（例如内存大小、CPU处理能力）和软件相关，每个业务不尽相同。

参考建议：

- YGC平均耗时小于50毫秒
- YGC执行频率不低于10秒1次
- FGC平均耗时小于1秒
- 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，检查业务是否有显式调用System.gc()代码，确认是否为业务触发FGC。优化业务代码删除不必要的显式调用FGC，或者设置-XX:DisableExplicitGC关闭显式调用：
`/path/java -XX:DisableExplicitGC <other parameters>`
2. FGC过于频繁，建议尝试调大永久代（Perm区）空间大小参数值或者调大老年代空间大小参数值，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资源不是瓶颈场景下可尝试增加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. 更大的年轻代会使老年代变小，大的年轻代会延长YGC的周期，但会增加每次YGC的耗时；小的老年代会增加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即时编译和堆空间管理垃圾回收等操作，JVM运行线程数量对系统性能有直接影响。通过资源监控找到CPU占用率较高的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，如下图取16810线程作为示例。

```
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转换成十六进制，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 <线程十六进制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 runnable
"Gang worker#0 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4005b800 nid=0x4082 runnable
"Gang worker#1 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4005d800 nid=0x4083 runnable
"Gang worker#2 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4005f000 nid=0x4084 runnable
"Gang worker#3 (Parallel GC Threads)" os_prio=0 tid=0x000003fe40062800 nid=0x4085 runnable
"Gang worker#4 (Parallel GC Threads)" os_prio=0 tid=0x000003fe40064800 nid=0x4086 runnable
"Gang worker#5 (Parallel GC Threads)" os_prio=0 tid=0x000003fe40066000 nid=0x4087 runnable
"Gang worker#6 (Parallel GC Threads)" os_prio=0 tid=0x000003fe40068000 nid=0x4088 runnable
"Gang worker#7 (Parallel GC Threads)" os_prio=0 tid=0x000003fe40069800 nid=0x4089 runnable
"Gang worker#8 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4006b800 nid=0x408a runnable
"Gang worker#9 (Parallel GC Threads)" os_prio=0 tid=0x000003fe4006d000 nid=0x408b runnable
"Concurrent Mark-Sweep GC Thread" os_prio=0 tid=0x000003fe400f4000 nid=0x408e runnable
"Gang worker#0 (Parallel CMS Threads)" os_prio=0 tid=0x000003fe400f0000 nid=0x408c runnable
"Gang worker#1 (Parallel CMS Threads)" os_prio=0 tid=0x000003fe400f1800 nid=0x408d runnable
```

----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线程数量设置为2的示例：

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

- b. 名称带有“C1”或“C2”的线程通常为即时编译（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的频率，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:+CICompilerCount=N，强制设定总编译线程数为N。例如，将JIT线程数量设置为4：

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

- 方法三：

直接关闭JIT，将参数“-XX:+CICompilerCount”设置为“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的运行，需根据业务函数调用特点进行评估。

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，处于阻塞状态。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地运行，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。

说明

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errno；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;
    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);
    }
}
}
```

B 进入 BIOS 界面

介绍如何进入BIOS Setup Utility界面。

操作场景

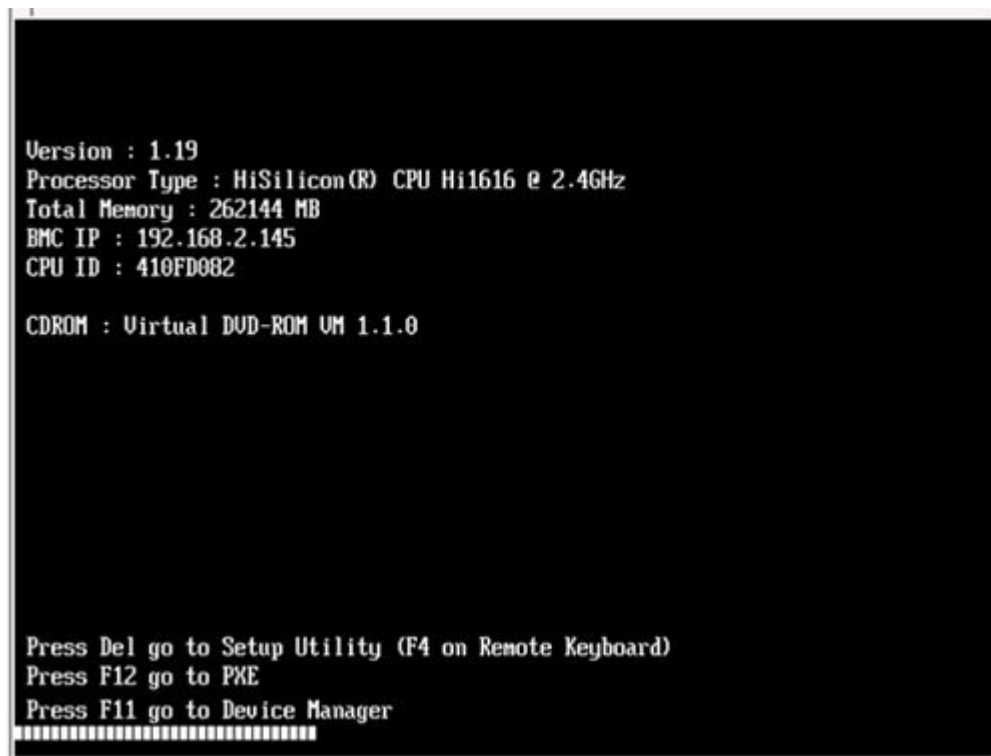
该任务指导用户在需要进行系统启动设置或系统信息查询的情况下，进入BIOS Setup Utility界面。

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** 开启服务器电源。
- 步骤3** 当出现如下界面时，按“Delete”进入BIOS Setup输入密码界面。



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

C NEON lib 库应用加速

原理

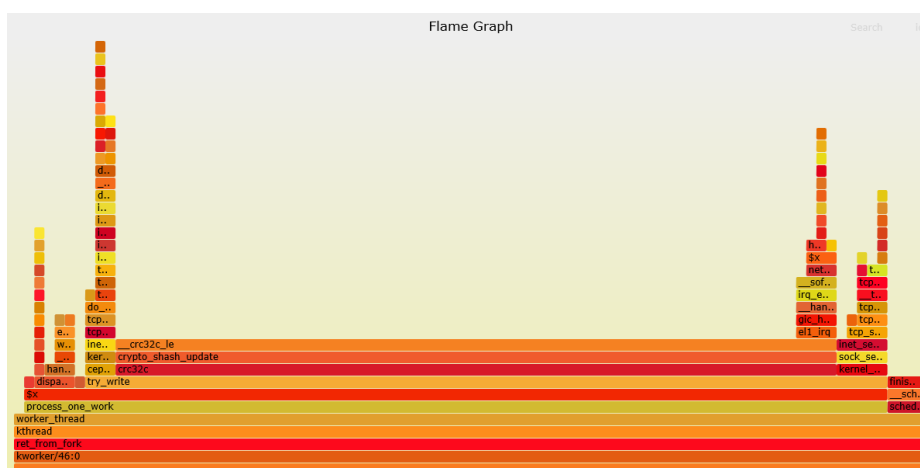
利用现有的一些基于NEON技术开发的典型lib，进行功能模块接口调用，达到应用程序加速目的。使开发者在获得最大效益的同时，而不必使用繁琐的NEON汇编或NEON Intrinsic函数进行编码加速。

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。主要包含以下领域：Math functions、Signal processing functions、Image processing functions、Physics functions，参考链接：<https://github.com/projectNe10/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://chromium.googlesource.com/libyuv/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。
参考链接：<https://github.com/google/skia>

D Linux 内核态 CRC32 算法优化

CRC32算法常用于信息的完整性校验，如网络通信中，在收到网络包之后，先对数据段做CRC32运算，然后将结果与接收到的CRC校验值对比，从而判断收到网络包的完整性。在Linux内核中，包含了一套由C语言实现的CRC32校验代码，但是执行效率不高。在内核态出现大量网络吞吐时，CRC32校验有可能成为性能瓶颈。



同时，在高版本的内核中（Kernel 5.x），已经实现了通过CRC指令的CRC32校验方式，该方式效率更高，能大大减少CRC32算法耗时。接下来介绍如何将该实现合入到Kernel 4.x中。

步骤1 从<https://github.com/torvalds/linux/blob/v5.19/arch/arm64/lib/crc32.S>仓库中拷贝crc32.S文件到“linux-4.14.105/arch/arm64/lib/”目录。

以4.14.105版本内核为例，将CRC32的汇编实现拷贝到“linux-4.14.105/arch/arm64/lib/”目录：

```
cp crc32.S linux-4.14.105/arch/arm64/lib/
```

步骤2 修改Makefile文件。

在“linux-4.14.105/arch/arm64/lib/Makefile”的末尾加一行，保证添加的crc32.S文件能编译到内核。

```
obj-$(CONFIG_CRC32) += crc32.o
```

```
lib-$(CONFIG_ARM64_LSE_ATOMICS) += atomic_ll_sc.o
CFLAGS_atomic_ll_sc.o := -ffixed-x1 -ffixed-x2 \
    -ffixed-x3 -ffixed-x4 -ffixed-x5 -ffixed-x6 \
    -ffixed-x7 -fcall-saved-x8 -fcall-saved-x9 \
    -fcall-saved-x10 -fcall-saved-x11 -fcall-saved-x12 \
    -fcall-saved-x13 -fcall-saved-x14 -fcall-saved-x15 \
    -fcall-saved-x18
lib-$(CONFIG_ARCH_HAS_UACCESS_FLUSHCACHE) += uaccess_flushcache.o
obj-$(CONFIG_CRC32) += crc32.o
```

步骤3 修改lib/crc32.c源码。

对C代码的CRC32校验接口添加__weak标识，通过该标识指明调用crc32_le接口时，该C接口不作为第一优先级调用。此时，若接口存在汇编实现，会优先调用汇编实现。同时，对汇编中实现的接口进行指向。

图 D-1 修改前

```
184 #if CRC_LE_BITS == 1
185 u32 __pure crc32_le(u32 crc, unsigned char const *p, size_t len)
186 {
187     return crc32_le_generic(crc, p, len, NULL, CRCPOLY_LE);
188 }
189 u32 __pure __crc32c_le(u32 crc, unsigned char const *p, size_t len)
190 {
191     return crc32_le_generic(crc, p, len, NULL, CRC32C_POLY_LE);
192 }
193 #else
194 u32 __pure crc32_le(u32 crc, unsigned char const *p, size_t len)
195 {
196     return crc32_le_generic(crc, p, len,
197                             (const u32 (*)[256])crc32table_le, CRCPOLY_LE);
198 }
199 u32 __pure __crc32c_le(u32 crc, unsigned char const *p, size_t len)
200 {
201     return crc32_le_generic(crc, p, len,
202                             (const u32 (*)[256])crc32ctable_le, CRC32C_POLY_LE);
203 }
204 #endif
205 EXPORT_SYMBOL(crc32_le);
206 EXPORT_SYMBOL(__crc32c_le);
```

华为云社区

图 D-2 修改后

```
184 #if CRC_LE_BITS == 1
185 u32 __pure __weak crc32_le(u32 crc, unsigned char const *p, size_t len)
186 {
187     return crc32_le_generic(crc, p, len, NULL, CRCPOLY_LE);
188 }
189 u32 __pure __weak __crc32c_le(u32 crc, unsigned char const *p, size_t len)
190 {
191     return crc32_le_generic(crc, p, len, NULL, CRC32C_POLY_LE);
192 }
193 #else
194 u32 __pure __weak crc32_le(u32 crc, unsigned char const *p, size_t len)
195 {
196     return crc32_le_generic(crc, p, len,
197                             (const u32 (*)[256])crc32table_le, CRCPOLY_LE);
198 }
199 u32 __pure __weak __crc32c_le(u32 crc, unsigned char const *p, size_t len)
200 {
201     return crc32_le_generic(crc, p, len,
202                             (const u32 (*)[256])crc32ctable_le, CRC32C_POLY_LE);
203 }
204 #endif
205 EXPORT_SYMBOL(crc32_le);
206 EXPORT_SYMBOL(__crc32c_le);
207
208 u32 __pure crc32_le_base(u32, unsigned char const *, size_t) __alias(crc32_le);
209 u32 __pure __crc32c_le_base(u32, unsigned char const *, size_t) __alias(__crc32c_le);
210
```

华为云社区

步骤4 重新编译内核。

参考：[E 操作系统内核源码编译](#)

----结束

E 操作系统内核源码编译

步骤1 查询当前内核的版本号和OS镜像来源。

```
[root@localhost ~]# uname -a
Linux localhost.localdomain 4.14.0-115.el7a.0.1.aarch64 #1 SMP Sun Nov 25 20:54:21 UTC 2018 aarch64
aarch64 aarch64 GNU/Linux
[root@localhost ~]# cat /etc/os-release
NAME="CentOS Linux"
VERSION="7 (AltArch)"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
PRETTY_NAME="CentOS Linux 7 (AltArch)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/"
BUG_REPORT_URL="https://bugs.centos.org/"

CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7"
REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"
```

当前操作系统发行厂商为：CentOS-7。内核版本号： 4.14.0-115.el7a.0.1

步骤2 从官方<http://vault.centos.org/7.6.1810/os/Source/SPackages/> 下载部分源码rpm包kernel-alt-4.14.0-115.el7a.0.1.src.rpm。

步骤3 编译内核之前，必须安装编译内核基本工具链。

如果遇到如下报错：

```
error: Failed build dependencies:
xmlto is needed by kernel-alt-4.14.0-115.el7.0.1.aarch64
asciidoc is needed by kernel-alt-4.14.0-115.el7.0.1.aarch64
newt-devel is needed by kernel-alt-4.14.0-115.el7.0.1.aarch64
pciutils-devel is needed by kernel-alt-4.14.0-115.el7.0.1.aarch64
```

可以使用如下命令解决：

```
yum install -y xmlto asciidoc newt-devel pciutils-devel
```

通常要求的工具链：

```
sudo yum install rpm-build redhat-rpm-config asciidoc hmaccalc perl-ExtUtils-Embed pesign xmlto
sudo yum install audit-libs-devel binutils-devel elfutils-devel elfutils-libelf-devel
sudo yum install ncurses-devel newt-devel numactl-devel pciutils-devel python-devel zlib-devel
```

步骤4 编译之前确保mockbuild用户和mockbuild所属的组有效。

```
groupadd mockbuild
useradd mockbuild -g mockbuild
```

步骤5 安装RPM包。此RPM包实际为rpmbuild构建工程。

```
rpm -ivh kernel-alt-4.14.0-115.el7a.0.1.src.rpm
```

安装完成，rpm构建工程自动部署在“/root/rpmbuild/SPECS”和“/root/rpmbuild/SOURCES”目录中。

步骤6 rpmbuild构建，会自动下载全量内核代码（包括补丁合入）。

```
cd /root/rpmbuild/SPECS  
rpmbuild -bp --target=$(uname -m) kernel-alt.spec
```

步骤7 构建完成后，可以获取到全量内核源码，路径为：

```
cd /root/rpmbuild/BUILD/kernel-alt-4.14.0-115.el7a/linux-4.14.0-115.el7a.0.1.aarch64
```

步骤8 内核编译。

```
make -j 64
```

步骤9 （可选）安装驱动。

```
make modules_install
```

步骤10 （可选）安装内核。

```
make install
```

步骤11 （可选）重启服务器，选择对应的内核。

----结束

F

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	当脏内存达到该字节数时触发pdflush后台回写
dirty_background_ratio	当脏内存所占百分比达到该值后触发pdflush后台回写
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	脏内存保留时间
dirty_writeback_centisecs	pdflush唤醒间隔
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：利用启发式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](#)。

G

修订记录

发布日期	修订记录
2024-06-07	第九次正式发布 更新crc32.S文件获取地址。
2022-02-25	第八次正式发布 - 应用程序优化方法中新增5.2.10 毕昇编译选项优化、5.2.11 鲲鹏数学库和5.2.12 并行I/O优化方法。 - 应用程序调优的热点函数优化中新增5.2.9.8 乘除法优化和5.2.9.9 循环不变代码外提优化方法。
2021-09-01	第七次正式发布 - 新增5.2.8 原子操作多核场景优化、5.2.9.2 软件预取、5.2.9.3 循环优化、5.2.9.4 数据布局优化、5.2.9.5 内联函数、5.2.9.6 OpenMP并行化、5.2.9.7 SHA256优化章节。 - 删除原“Linux内核态CRC32算法优化”和“操作系统内核源码编译”章节。
2021-01-30	第六次正式发布 新增3.3.5 开启LRO、3.3.9 内核CRC32优化、3.3.10 tuned配置选择、D Linux内核态CRC32算法优化、E 操作系统内核源码编译章节。
2020-11-19	第五次正式发布 修改“2.3.4 NUMA优化，减少跨NUMA访问内存”和“3.3.2 网络NUMA绑核”部分参数。

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“3.3.7 单队列网卡中断散列”和“3.3.8 TCP checksum优化”章节。 “4 磁盘I/O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“5.2.9.1 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“F 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3.3.6 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