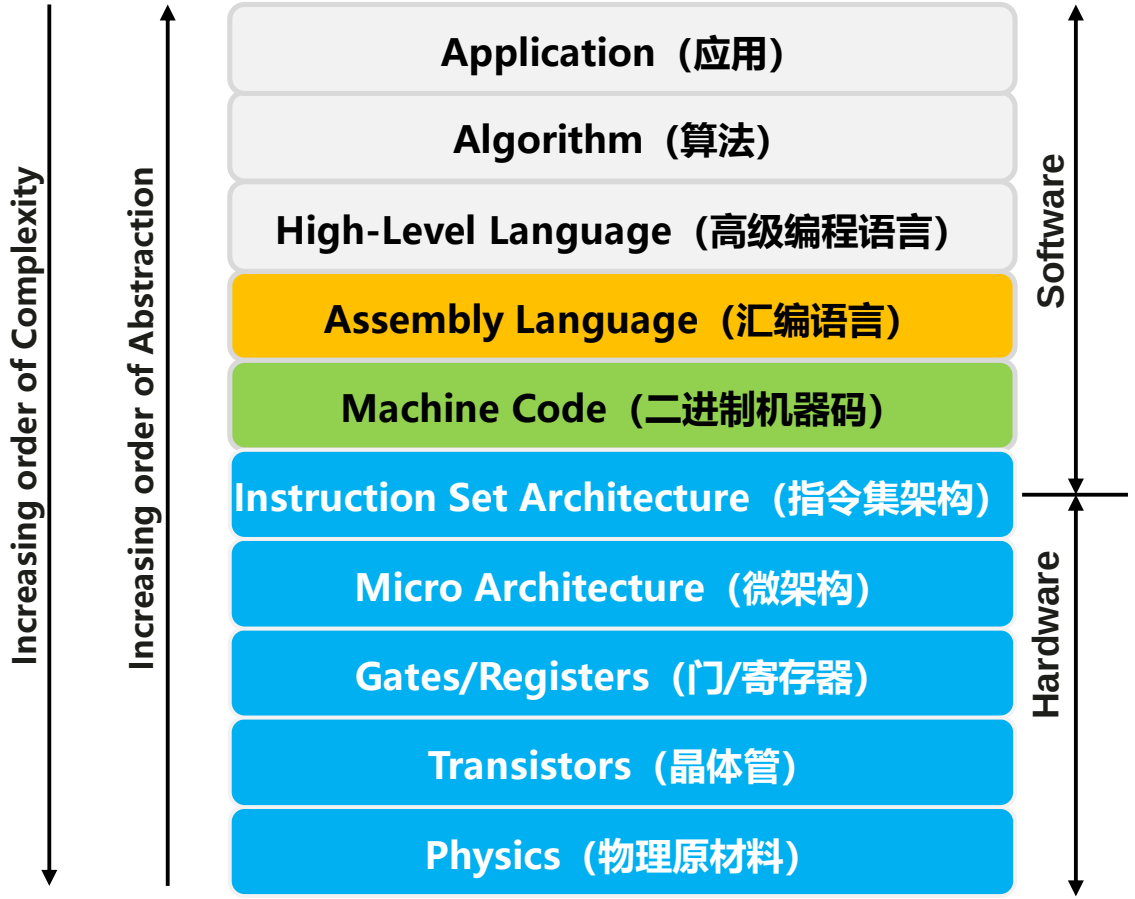
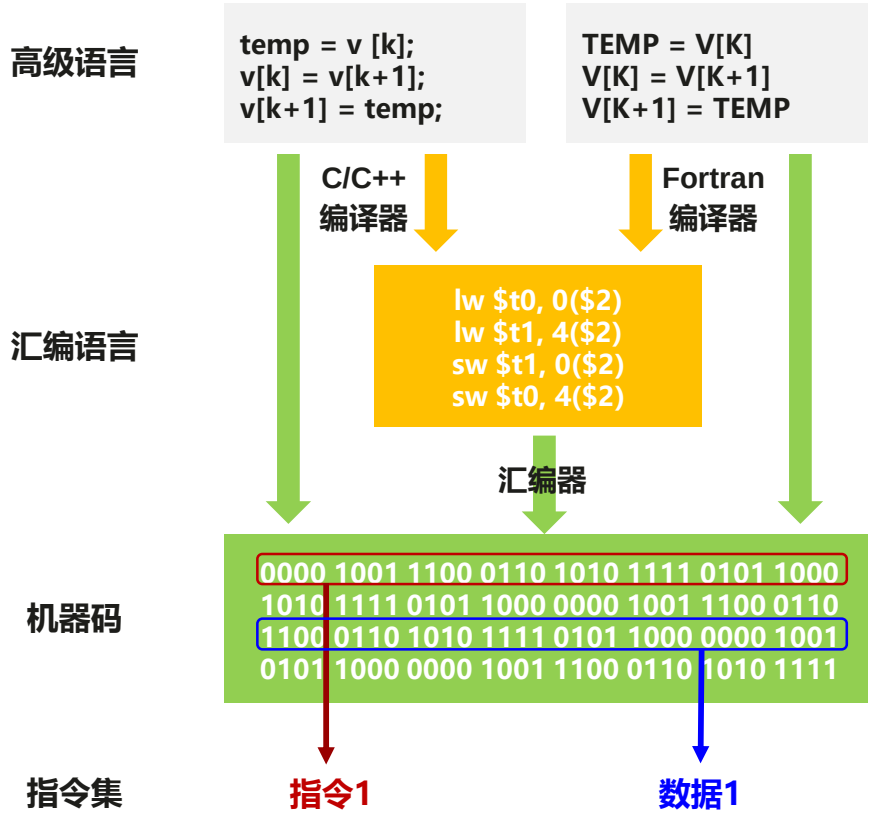
目录

计算技术栈与程序执行过程

计算技术栈



程序执行过程



鲲鹏处理器与x86处理器的指令差异

程序代码（C/C++）：

```
int main()
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = a + b;
    return c;
}
```

编译

鲲鹏处理器指令

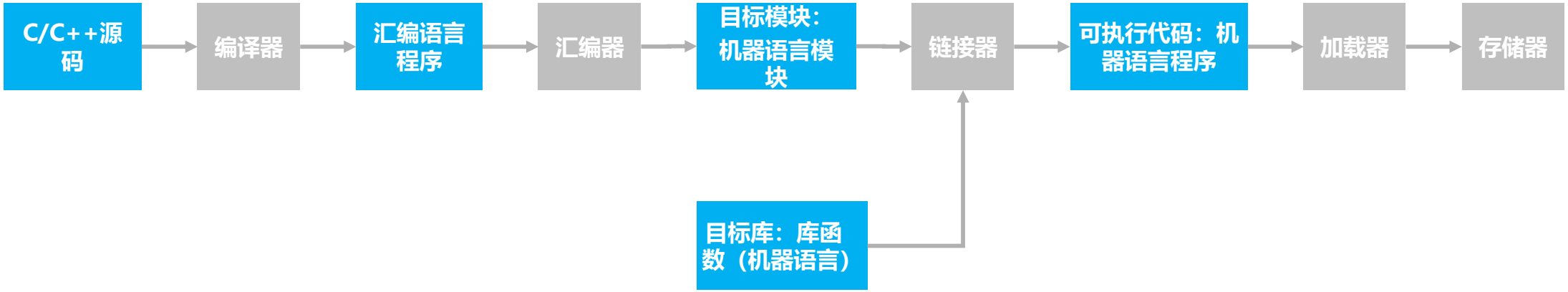
指令	汇编代码	说明
b9400fe1	ldr x1, [sp,#12]	从内存将变量a的值放入寄存器x1
b9400be0	ldr x0, [sp,#8]	从内存将变量b的值放入寄存器x0
0b000020	add x0, x1, x0	将x1(a)中的值加上x0(b)的值放入x0寄存器
b90007e0	str x0, [sp,#4]	将x0寄存器的值存入内存（变量c）

x86处理器指令

指令	汇编代码	说明
8b 55 fc	mov -0x4(%rbp),%edx	从内存将变量a的值放入寄存器edx
8b 45 f8	mov -0x8(%rbp),%eax	从内存将变量b的值放入寄存器eax
01 d0	add %edx,%eax	将edx(a)中的值加上eax(b)的值放入eax寄存器
89 45 f4	mov %eax,-0xc(%rbp)	将eax寄存器的值存入内存（变量c）

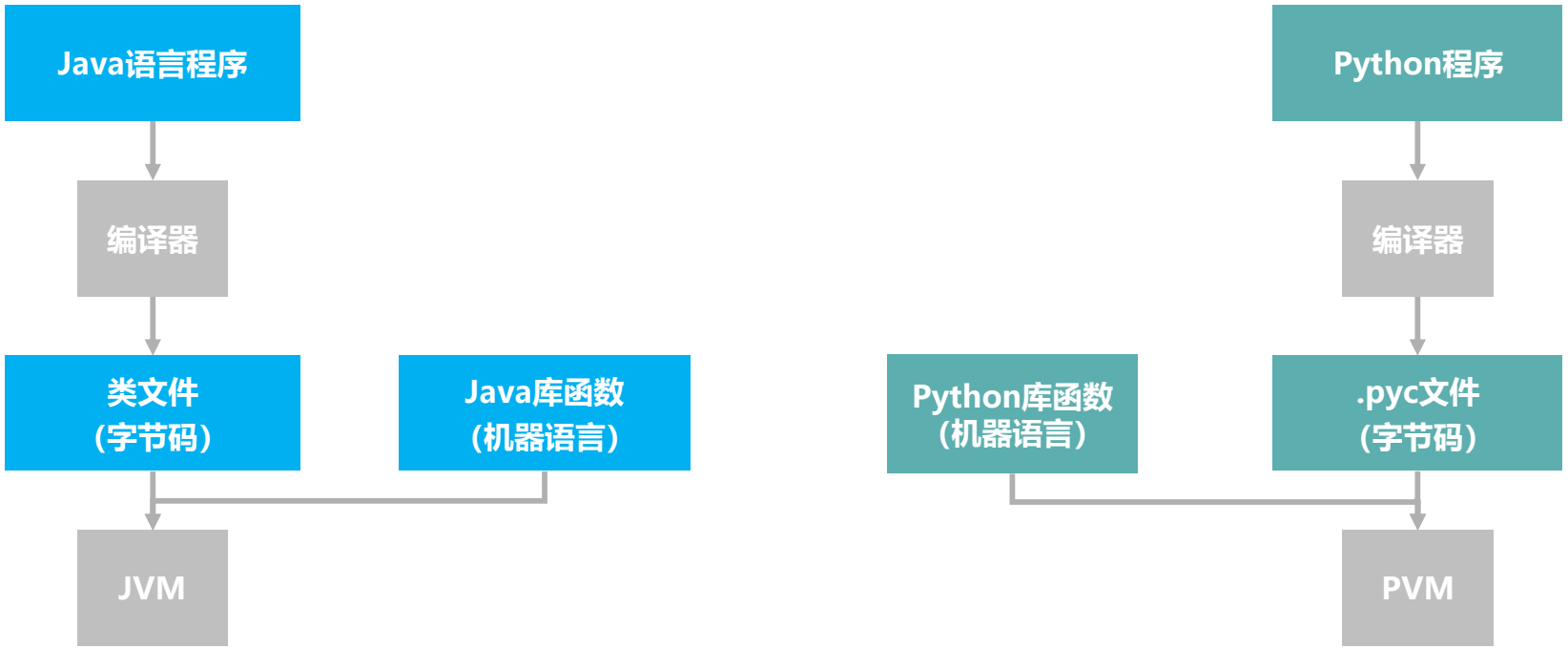
从源码到可执行程序 - 编译型语言

- **编译型语言**：典型的如C/C++/Go语言，都属于编译型语言。编译型语言开发的程序在从x86处理器迁移到鲲鹏处理器时，必须经过重新编译才能运行。
- 从源码到程序的过程：源码需要由编译器、汇编器翻译成机器指令，再通过链接器链接库函数生成机器语言程序。机器语言必须与CPU的指令集匹配，在运行时通过加载器加载到内存，由CPU执行指令。



从源码到可执行程序 - 解释型语言

- **解释型语言**：典型的如Java/Python语言，都属于解释型语言，解释型语言开发的程序在迁移到鲲鹏处理器时，一般不需要重新编译。
- 解释型语言的源代码由编译器生成字节码，然后再由虚拟机解释执行。虚拟机将不同CPU指令集的差异屏蔽，因此解释型语言的可移植性很好。但是如果程序中调用了编译型语言所开发的so库，那么这些so库需要重新移植编译。



其它语种分类

Go　lua　Ruby　PHP

Scala　Perl　TypeScript　JavaScript

编译型　解释型

编译型：源码需要重新编译

解释型：源码不用编译，安装解释器即可

搭建程序运行环境方法:

1、操作系统自带解释器软件，直接安装即可使用；

2、从官网上下载支持ARM的解释器软件包，直接解压使用；

3、从官网下载解释器源码，编译后使用。

目录

C/C++语言char数据类型默认符号不一致问题（1）

问题现象

C/C++代码在编译时遇到如下提示：

告警信息：warning: comparison is always false due to limitedrange of data type

原因分析

char变量在不同CPU架构下默认符号不一致，在x86架构下为signed char，在ARM64平台为unsigned char，移植时需要指定char变量为signed char。

解决方案

1. 在编译选项中加入“-fsigned-char”选项，指定ARM64平台下的char为有符号数；
2. 将char类型直接声明为有符号char类型：signed char。

C/C++语言char数据类型默认符号不一致问题 (2)

鲲鹏弹性云服务器

```
[root@ecs-8225-kunpeng ~]#
[root@ecs-8225-kunpeng ~]# uname -a
Linux ecs-8225-kunpeng 4.14.0-115.5.1.el7a.aarch64 #1 SMP Mon Feb 4 16:38:08 UTC 2019 aarch64
[root@ecs-8225-kunpeng ~]# cat charTest.c
#include <stdio.h>
int main()
{
  char ch=-1;
  printf("ch=%d\n",ch);
  return 0;
}

[root@ecs-8225-kunpeng ~]# gcc -o charTest charTest.c
[root@ecs-8225-kunpeng ~]# ./charTest
ch=255
[root@ecs-8225-kunpeng ~]#
```

X86弹性云服务器

```
[root@ecs-82e5-x86 ~]# uname -a
Linux ecs-82e5-x86 3.10.0-1062.1.1.el7.x86_64 #1 SMP Fri Sep 13 22:55:44 UTC 2019 x86_64
[root@ecs-82e5-x86 ~]# cat charTest.c
#include <stdio.h>
int main()
{
  char ch = -1;
  printf("ch=%d\n",ch);
  return 0;
}
[root@ecs-82e5-x86 ~]# gcc -o charTest charTest.c
[root@ecs-82e5-x86 ~]# ./charTest
ch=-1
[root@ecs-82e5-x86 ~]# _
```

可以看到：相同的代码，鲲鹏下char默认为unsigned char类型，所以赋值为-1的时候，输出的为-1对256取模的结果即255，X86中的char默认为signed char类型，输出为-1

C/C++语言中调用汇编指令编译错误

问题现象

C/C++代码在编译时遇到如下提示：

错误信息：error：impossible constraint in ‘asm’ __asm__ __volatile__

原因分析

代码中使用汇编指令，而汇编指令与cpu指令集强相关。在x86架构cpu中的汇编指令需要修改为鲲鹏处理器平台的指令才能编译通过，实现功能替换。

解决方案

1. 本例中的代码调用了x86平台的汇编指令，修改为鲲鹏处理器对应的指令即可；
2. 部分功能可以修改为使用编译器自带的builtin函数，在基本不降低性能的前提下，提升代码的可移植性。

编译错误：无法识别-m64编译选项

问题现象

C/C++代码在编译时遇到如下提示：

错误信息：gcc: error: unrecognized command line option ‘-m64’

原因分析

-m64是x86 64位应用编译选项，m64选项设置int为32bits及long、指针为64 bits，为AMD的x86 64架构生成代码。在鲲鹏处理器平台无法支持。

解决方案

将鲲鹏处理器平台对应的编译选项设置为-mabi=lp64，重新编译即可。

超出整型取值范围时浮点型转整型与x86不一致

问题现象

C/C++双精度浮点型数转整型数据时，如果超出了整型的取值范围，鲲鹏处理器的表现与x86平台的表现不同。测试代码如下：

```
long aa = (long)0x7FFFFFFFFFFFFFFF;
long bb;
bb = (long)(aa*(double)10); // long->double->long
// x86      :  aa=9223372036854775807, bb=-9223372036854775808
// Kunpeng :aa=9223372036854775807, bb=9223372036854775807
```

原因分析

x86（指令集）中的浮点到整型的转换指令，定义了一个indefinite integer value——“不确定数值”（64bit：0x8000000000000000），大多数情况下x86平台确实都在遵循这个原则，但是在从double向无符号整型转换时，又出现了不同的结果。鲲鹏的处理则非常清晰和简单，在上溢出或下溢出时，保留整型能表示的最大值或最小值。

C/C++语言double类型超出整型取值范围向整型转换参照表

CPU	double值	转为long变量保留值
x86	正值超出long范围	0x8000000000000000
x86	负值超出long范围	0x8000000000000000
鲲鹏	正值超出long范围	0x7FFFFFFFFFFFFFFF
鲲鹏	负值超出long范围	0x8000000000000000

CPU	double值	转为long变量保留值
x86	正值超出long范围	0x8000000000000000
x86	负值超出long范围	0x8000000000000000
鲲鹏	正值超出long范围	0x7FFFFFFFFFFFFFFF
鲲鹏	负值超出long范围	0x8000000000000000

CPU	double值	转为long变量保留值
x86	正值超出long范围	0x8000000000000000
x86	负值超出long范围	0x8000000000000000
鲲鹏	正值超出long范围	0x7FFFFFFFFFFFFFFF
鲲鹏	负值超出long范围	0x8000000000000000

CPU	double值	转为long变量保留值
x86	正值超出long范围	0x8000000000000000
x86	负值超出long范围	0x8000000000000000
鲲鹏	正值超出long范围	0x7FFFFFFFFFFFFFFF
鲲鹏	负值超出long范围	0x8000000000000000

HUAWEI

对结构体中的变量进行原子操作时程序异常

问题现象

程序调用原子操作函数对结构体中的变量进行原子操作，程序coredump，堆栈如下：

```
Program received signal SIGBUS, Bus error.
0x000000000040083c in main () at /root/test/src/main.c:19
19          __sync_add_and_fetch(&a.count, step);
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400824 <+0>:     sub     sp, sp, #0x10
   0x0000000000400828 <+4>:     mov     x0, #0x1                   // #1
   0x000000000040082c <+8>:     str     x0, [sp, #8]
   0x0000000000400830 <+12>:    adrp    x0, 0x420000 <__libc_start_main@got.plt>
   0x0000000000400834 <+16>: add x0, x0, #0x31 //将变量的地址放入x0寄存器
   0x0000000000400838 <+20>: ldr  x1, [sp, #8] //指定ldxr取数据的长度(此处为8字节)
=> 0x000000000040083c <+24>: ldxr x2, [x0] //ldxr从x0寄存器指向的内存地址中取值
   0x0000000000400840 <+28>:    add     x2, x2, x1
   0x0000000000400844 <+32>:    stlxr   w3, x2, [x0]
   0x0000000000400848 <+36>:    cbnz    w3, 0x40083c <main+24>
   0x000000000040084c <+40>:    dmb     ish
   0x0000000000400850 <+44>:    mov     w0, #0x0                   // #0
   0x0000000000400854 <+48>:    add     sp, sp, #0x10
   0x0000000000400858 <+52>:    ret
End of assembler dump.
(gdb) p /x $x0
$4 = 0x420039 // x0寄存器存放的变量地址不在8字节地址对齐处
```

原因分析

鲲鹏处理器对变量的原子操作、锁操作等用到了ldaxr、stlxr等指令，这些指令要求变量地址必须按变量长度对齐，否则执行指令会触发异常，导致程序coredump。

一般是因为代码中对结构体进行强制字节对齐，导致变量地址不在对齐位置上，对这些变量进行原子操作、锁操作等会触发问题。

解决方案

代码中搜索“#pragma pack”关键字（该宏改变了编译器默认的对齐方式），找到使用了字节对齐的结构体，如果结构体中变量会被作为原子操作、自旋锁、互斥锁、信号量、读写锁的输入参数，则需要修改代码保证这些变量按变量长度对齐。

目录

NUMA

- 背景知识：

DDR：Double Data Rate SDRAM，双倍速率SDRAM。就是我们常说的内存条。

Socket：中文翻译为插槽，这里代表一颗可以在主板上独立插拔的CPU 。一个主板上可以放多个Socket。

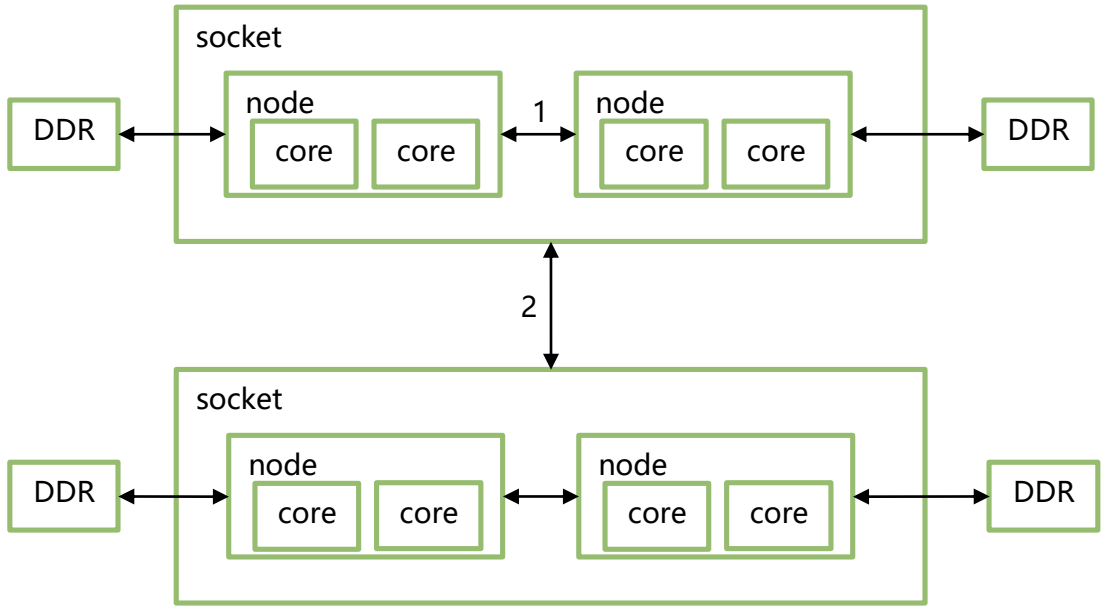
NUMA node：一个逻辑概念，属于同一个node的core共享一些资源，如内存控制器。 1个Socket可以包含1个或多个 NUMA node。

Core：一个独立的硬件执行单元，有独立的算术逻辑单元和寄存器等。一个NUMA node可以包含多个core。

- 物理上，一个DDR只挂载在一个node上，其它node要访问这个node上的DDR需要通过片内总线（如图中的1）或片间总线（如图中的2）进行通信，内存访问延迟从低到高为：

  NUMA内 < 跨NUMA不跨socket < 跨socket

- 一颗鲲鹏920有两个NUMA节点，每个节点有4个DDR通道



NUMA优化

- 程序要避免跨NUMA访问内存：
  - 在网络中断的CPU占用高时，可以通过设置网卡中断的CPU亲和性，防止中断被跨NUMA处理：

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

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

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

  - 在C/C++代码中通过sched_setaffinity函数来设置线程亲和性；
  - 很多开源软件已经支持在自带的配置文件中修改线程的亲和性，如KVM开源软件，通过virsh edit $vmname修改cpuset等参数来设置线程的亲和性。

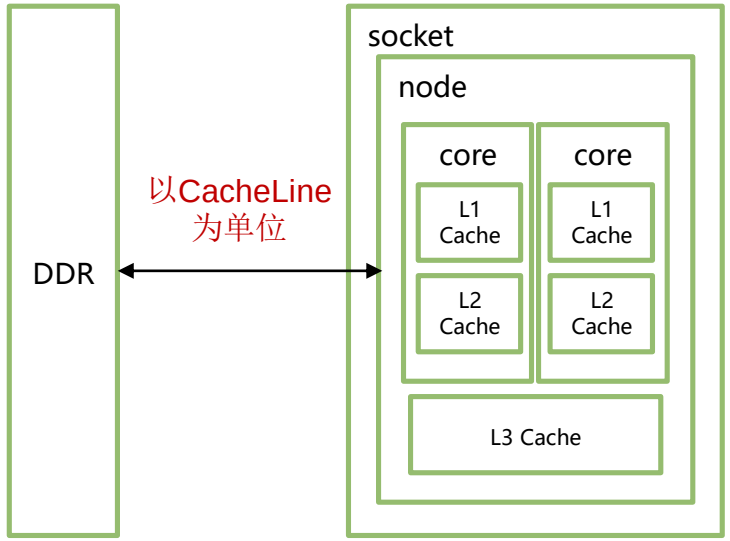
- 效果：在测试ceph软件的4K随机写性能时，通过设置线程的亲和性，性能提升了**32%** 。

CacheLine

- 背景知识：

为了使DDR的访问性能匹配CPU的计算能力，CPU内部通过高速缓存(Cache)缓冲部分DDR 的数据，从而能减少对低速DDR 的访问次数，从而提升性能。其中Cache又分为3层：L1 Cache,L2 Cache和L3 Cache。从制造难度上看，L1 Cache>L2 Cache>L3 Cache>DDR。从性能上看， L1 Cache>L2 Cache>L3 Cache>DDR。

- L3 Cache标识数据是否有效是以CacheLine为单位，数据从内存读到L3 Cache也是以CacheLine为单位。不同CPU，L3 CacheLine大小不一样：如x86为64字节，KunPeng 920为128字节。

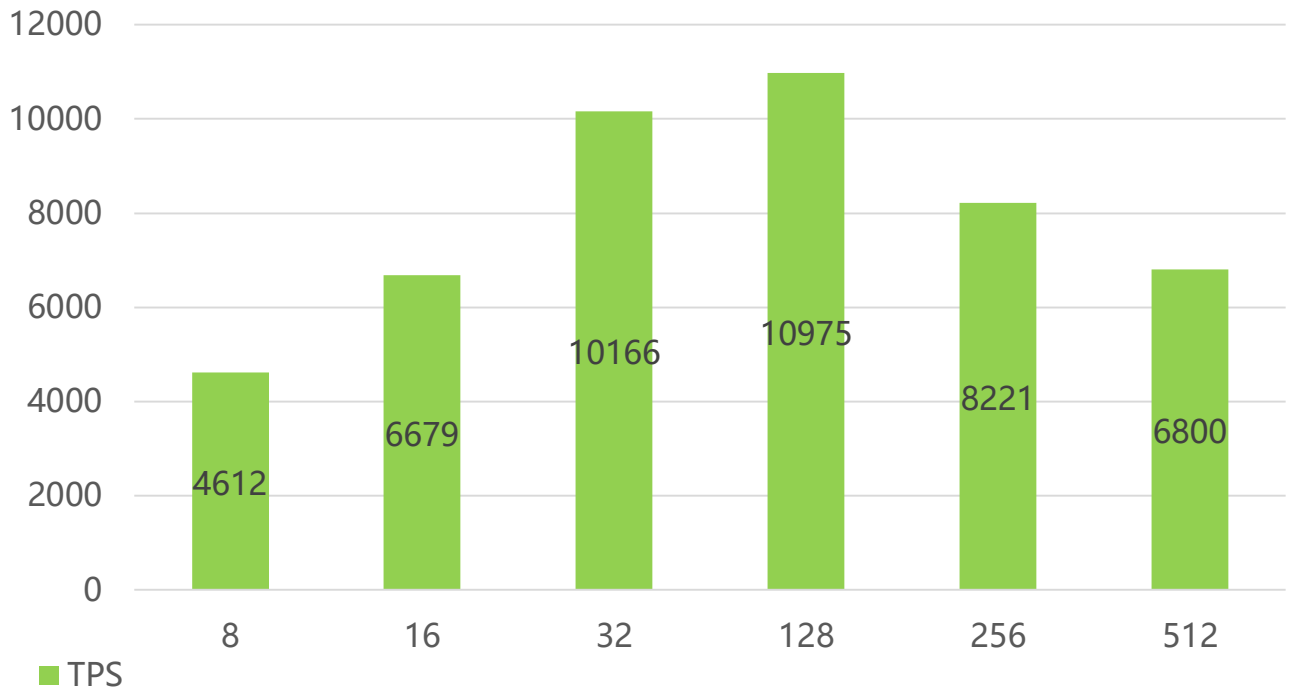


CacheLine优化

- 原因：上一节讲到的CacheLine机制会导致伪共享（false sharing），造成伪共享的原因：
  - 假设有如下两个变量：int readHighFreq, writeHighFreq在同一个CacheLine中，其中readHighFreq是读频率高的变量，writeHighFreq为写频率高的变量。
  - writeHighFreq被修改：其它CPU Cache中writeHighFreq所在的CacheLine被标识为无效状态，
  - 从内存中读而不是从CPU cache中读readHighFreq：readHighFreq所在CacheLine已经被标识为无效状态，虽然readHighFreq没有被其它core修改，但是还是会重新从内存中导入数据到CPU cache，导致伪共享。
- 解决方法：将写频繁的变量，如各种锁变量声明为CacheLine字节对齐。
- 效果：MySQL数据库的锁使用128字节对齐后，性能提升**5%**。

线程并发数调整

- 从一个线程变为多线程时，CPU和内存资源得到充分利用，性能得到提升。但是超过一定限制后，因为资源的争抢比较严重，性能会下降。
- TPS（Transactions Per Second）：每秒处理的事务数。

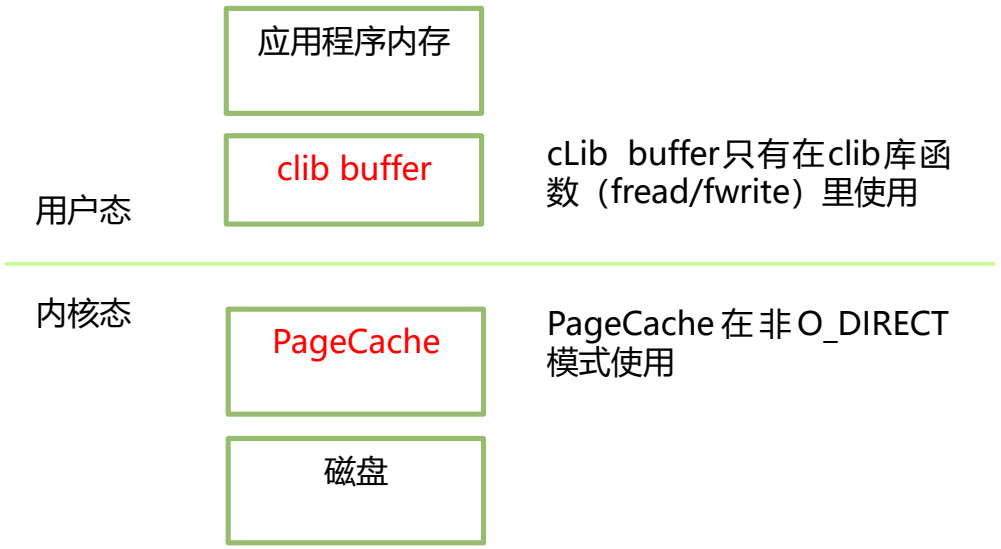


磁盘读写优化

三种方式(fread/fwrite, read/write O_DIRECT和read/write 非O_DIRECT模式)都有自己的优势和劣势，需要根据业务特点选择不同的方式：

- 使用fread/fwrite函数比read/write函数多了一层缓冲(图中的clib buffer)，也就多了一次内存拷贝，但是减少了用户态和内核态的切换。
- read/write函数的O_DIRECT模式比非O_DIRECT模式少了一次内存拷贝(图中的PageCache)，但是每次读写都直接操作磁盘。
- fread/fwrite和read/write函数有个明显的缺点，就是磁盘文件，文件的读取是同步的，导致线程读取文件时，属于阻塞状态。一般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，使得文件读写不再阻塞，结合epoll机制，实现一个线程可以无阻塞地运行，同时处理多个网络请求和文件读写请求，提升服务器整体性能。具体代码实现可以在网上搜索libaio-epoll实现。



学习推荐

- 《TaiShan代码移植指导》
  - https://bbs.huaweicloud.com/forum/thread-22606-1-1.html
- 华为云鲲鹏社区
  - https://www.huaweicloud.com/kunpeng/
- 《计算机组成与设计（ARM版）》


谢 谢
www.huawei.com