**深 圳 大 学 实 验 报 告**

**课程名称： 计算机系统(2)**

**实验项目名称： Cache实验**

**学院： 计算机与软件学院**

**专业： 计算机科学与技术**

**指导教师： 刘 刚**

**报告人： 吴嘉楷 学号： 2022150168 班级： 国际班**

**实验时间： 2024年6月1日至6月21日**

**实验报告提交时间： 2024年6月20日**

**教务处制**

|  |
| --- |
| **一、实验目的：**   1. 加强对Cache工作原理的理解； 2. 体验程序中访存模式变化是如何影响cahce效率进而影响程序性能的过程； 3. 学习在X86真实机器上通过调整程序访存模式来探测多级cache结构以及TLB的大小。 |
| **二、实验环境**  X86真实机器 |
| **三、实验内容和步骤**  **1、分析Cache访存模式对系统性能的影响**   * 1. 给出一个矩阵乘法的普通代码A，设法优化该代码，从而提高性能。   2. 改变矩阵大小，记录相关数据，并分析原因。   **2、编写代码来测量x86机器上（非虚拟机）的Cache 层次结构和容量**   1. 设计一个方案，用于测量x86机器上的Cache层次结构，并设计出相应的代码； 2. 运行你的代码获得相应的测试数据； 3. 根据测试数据来详细分析你所用的x86机器有**几级Cache**，**各自容量**是多大？ 4. 根据测试数据来详细分析**L1 Cache行**有多少？   **3、尝试测量你的x86机器TLB有多大？（选做）**  代码A：  #include <sys/time.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  float \*a,\*b,\*c, temp;  long int i, j, k, size, m;  struct timeval time1,time2;    if(argc<2) {  printf("\n\tUsage:%s <Row of square matrix>\n",argv[0]);  exit(-1);  } //if  size = atoi(argv[1]);  m = size\*size;  a = (float\*)malloc(sizeof(float)\*m);  b = (float\*)malloc(sizeof(float)\*m);  c = (float\*)malloc(sizeof(float)\*m);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  a[i\*size+j] = (float)(rand()%1000/100.0);  b[i\*size+j] = (float)(rand()%1000/100.0);  }  }    gettimeofday(&time1,NULL);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  c[i\*size+j] = 0;  for (k=0;k<size;k++)  c[i\*size+j] += a[i\*size+k]\*b[k\*size+j];  }  }  gettimeofday(&time2,NULL);    time2.tv\_sec-=time1.tv\_sec;  time2.tv\_usec-=time1.tv\_usec;  if (time2.tv\_usec<0L) {  time2.tv\_usec+=1000000L;  time2.tv\_sec-=1;  }    printf("Executiontime=%ld.%06ld seconds\n",time2.tv\_sec,time2.tv\_usec);  return(0);  }//main |
| **四、实验结果及分析**  **1、分析Cache访存模式对系统性能的影响**   1. 给出一个矩阵乘法的普通代码A，设法优化该代码，从而提高性能。   分析代码A，可知代码A在执行“矩阵乘法”的操作，于是我们猜测本题与cache以及局部性原理有关。  在访问矩阵b的元素时，程序是按行访问的，以6×6矩阵为例，其访问顺序如图1所示：    图 1 代码A访问矩阵元素顺序  在矩阵列数较大的情况下，其由于不具有空间局部性，效率较低。因此优化思路可以从空间局部性出发，使程序以列优先顺序访问矩阵b，从而充分利用cache资源完成数据传输，核心代码如图2所示：    图 2 代码B核心代码  代码思路：  i代表矩阵的行，j代表矩阵的列，k是矩阵叉乘时行或列的下标，我们只需要调换j和k的执行顺序即可改变程序队矩阵b的访问顺序。  此外，由于调换j和k的嵌套关系之后，a[ i×size+k ]仅会在i或k改变时改变，为了减少对地址的访问，我们可以使用变量temp将值记录下来。  优化后，对矩阵的访问顺序如下：    图3 代码A-pro访问矩阵元素顺序   1. 改变矩阵大小，记录相关数据，并分析原因。   首先，分别将优化前后的代码编译为可执行文件：（使用O2优化级别）    图4 编译代码为可执行文件  然后，不断改变矩阵大小，运行程序并记录数据：    图5.1 代码A运行时间    图5.2 代码A优化后的运行时间  将数据整理后得到以下表格：  表1、普通矩阵乘法与及优化后矩阵乘法之间的性能对比   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 矩阵大小 | 100 | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | | 一般算法执行时间 | 0.0011 | 0.1393 | 1.1717 | 4.8011 | 10.988 | 36.822 | 133.68 | | 优化算法执行时间 | 0.0009 | 0.0559 | 0.4533 | 2.2028 | 4.3997 | 9.5046 | 16.667 | | 加速比  speedup | 1.2222 | 2.4920 | 2.5848 | 2.1795 | 2.4974 | 3.8741 | 8.0206 |   加速比定义：加速比=优化前系统耗时/优化后系统耗时；  所谓加速比，就是优化前的耗时与优化后耗时的比值。加速比越高，表明优化效果越明显。  分析原因：  Cache和主存或cache和cache之间，以块为最小单位进行数据传输，而每个块里面存储着多个相邻地址的数据。例如，对矩阵a[6][6]，第一次使用a[0][0]时，会将其与相邻的部分元素a[0][1]，a[0][2]等放入到同一个块中，如果之后的访问按照连续的地址访问，则命中率更高，时间周期较少，效率更高。如果按照a[1][0]，a[2][0]顺序，即按行访问，因为Cache空间有限，a[1][0]可能不在块中，则命中率降低，需要替换，这增加了时间周期，因此性能较低。  原理图如下：（落在cache块里则无需再从内存中取数据）    图6 cache块命中  **2、测量分析出Cache 的层次结构、容量以及L1 Cache行有多少？**   1. **实验原理；**   Cache分为3层L1、L2、L3，如果存储的数据量超过了L1的容量，则会存储到L2，L3层。随着Cache层级的增加，访问速度会逐渐变慢，因为每个Cache层级都比前一个层级大，而访问速度则相应降低。在随机访问模式下，即CPU需要访问的数据不总是位于同一个Cache层级，因此可能会在Cache的各个层级之间频繁地存取数据。由于**Cache层级之间的访问速度不同，当数据在不同层级之间传输时，Cache Miss会增加，从而产生较大的速率变化。**  因此，**对于每一个cache层级边界的数据存取会产生较大的速率变化**。   1. **测量方案及代码；**   根据以上实验原理，我们可以生成容量大小递增的数组，通过随机访问数组位置来模拟从内存读取数据的操作，如何记录访问的总时间。随着内存块的递增，会出现几次较大时间增加，即达到cache层级边界，从而大概估计出不同cache层级大小。  对于L1 Cache行的容量，求法也类似于求Cache容量，且cache line的最大范围应该是L1 cache的容量。当访问的数组size小于cache line容量时耗时较短，大于cache line容量时耗时较大。  ①测量Cache层次结构的代码见附件，随机访问情况示意图如图7,图8所示：access访问函数中，首先开辟一块大小为 size（kb）的内存空间，并用一个数组存储随机产生的位置。之后根据这些位置对开辟的内存空间进行随机内存访问（较大次数），记录时间。test\_cache函数中传入不同大小的内存重复调用access函数进行测试。之后绘制size 与时间的关系表格并作图分析。当内存size大于Cache的大小时，访问miss的次数增加，时间周期增加，分析图表发生较大变化处即为Cache层次结构的边界。    图7 内存size<Cache时访问示意图    图8 内存size>Cache时访问示意图  ②在测出Cache L1的大小后，根据此范围等间距取数进行测试。  步长小于Cache行容量时，访问都能命中：    图9 步长<Cache行容量时访问示意图  而当步长大于Cache行容量时，会发生miss，增加耗时：    图10 步长>Cache行容量时访问示意图   1. **测试结果；**     图表11 Cache层次结构测试结果    图表12 Cache line测试结果   1. **分析过程；**   图13 Cache层次结构测试曲线  分析可知，从384KB增加到768KB，所消耗的时间增加较多，推测第一级Cache的大小在384 KB-768 KB范围内，而之后耗时继续增加，在8192 KB之后，大幅增加，推测第二级Cache的大小在该值附近，而大于16384 KB之后，耗时的增加量逐渐减少，则推测第三级Cache的大小在16384 KB左右。  即推测第一级Cache大小为512 KB，第二级Cache大小为8192 KB，第三级Cache大小为16384 KB。  图14 Cache line测试曲线  当访问的偏移量大于L1 Cache Line的容量时，访问数据会多次发生miss，小于时若干次访问才可能发生一次miss。而分析图表12、图14的曲线，当偏移的长度大于32B时，访问的耗时开始急剧增加，推测L1 Cache Line的大小在32 B左右。  综上所述，推测Cache的层次结构为L1、L2和L3；L1的容量为512 KB，L2的容量为8 MB，L3的容量为16 MB；L1的Cache line的容量为32 B，总行数等于512K / 32，即16000。   1. **验证实验结果：**   ①验证Cache层次结构及各层容量    图15 任务管理器查看Cache  如上图所示，查看电脑的任务管理器，可以得知第一级Cache的大小为512 KB，第二级Cache的大小为8 MB（8192 KB），第三级Cache的大小为16 MB（16384 KB），符合步骤（4）中的分析结果。  ②验证L1 Cacheline的容量    图16 CPU-Z软件查看L1的Cache Line  下载CPU-Z软件，并使用其查看CPU参数，结果如上图，可知L1 Cache Line的大小为32 B，符合步骤（4）中的分析结果。包括L1、L2、L3的容量也同时得到了进一步的验证。  **3、尝试测量你的x86机器TLB有多大？（选做）** |
| **五、实验结论**  （1）对于main()函数参数(int argc,char \*argv[])，argc为参数个数（及argv数组的有效长度），argv为我们运行可执行文件时输入的命令行字符串（包括了文件路径）。在本题中，argv[0]为运行目录路径和程序名，agrv[1]为我们想要设置的矩阵的size。  （2）利用Cache的工作原理，以及局部性原理可以来有效地优化代码，这能较大提高代码的性能。  （3）gcc编译时使用-O2优化级别可以有效地优化程序运行效率。  （4）通过本次实验，我对Cache的层次结构有了更深的了解，对其容量大小也有了一定的认识。  （5）通过本次实验，我学会了利用随机访问数据的方法来测量判断各级Cache及其行cache line的容量大小，可以通过记录他们大量存取大数据时的时间来判断（若时间变化较大，则为cache边界）。 |

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2024年 月 日 |
| 备注： |