**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： 软件工程**

**指导教师： 罗胜**

**报告人： 张欣杰 学号： 2020151091 班级： 软工2班**

**实验时间： 2022年 6 月 2日至 6月 23 日**

**实验报告提交时间： 2022年6月23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？   **4、选做：尝试测量你的x86机器TLB有多大？**  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**  使用vi编辑器编辑代码A，使用指令gcc test.c -o test.out，然后使用指令./test.out+数字，执行对应大小矩阵的乘法运算，记录运行的时间。    对代码进行优化，优化后的代码B:  #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);  }  }  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  b[i\*size+j]=c[j\*size+i];  }  }  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[j\*size+k];  }  }  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  标红部分为修改的代码，重新编译运行代码，记录不同矩阵大小下  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 | 0.0128 | 1.918 | 6.044 | 33.303 | 90.951 | 204.297 | 380.682 | | 优化算法执行时间 | 0.009 | 0.585 | 3.826 | 12.948 | 29.786 | 58.096 | 99.439 | | 加速比  speedup | 1.422 | 3.278 | 1.579 | 2.572 | 3.053 | 3.516 | 3.828 |   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。  通过优化对比图可以看到，随着数据个数的增大，优化前后的差于越来越大，随着数据量的增大，加速比也越来越大，但是在数据量为500的时候，加速比比后边的大，猜测是由于运行环境导致的。  分析原因：  通过空间局部性原理分析可以得到，如果一个内存的地址被引用了一次，若程序在后续引用其附近的一个内存地址即可实现空间局部性原理。在矩阵A和矩阵B 相乘的时候，矩阵A是按行读取的，而矩阵B是按列读取的，因此在实现A的行与B的列相乘并将结果存放在矩阵C的时候，按行读取时，由于元素间的地址是相邻的，符合空间局部性原理，而如果按列读取，上下元素间隔了一行的位置，空间局部性较差，此时在读取数据的时候Cache的命中率降低，读取速度慢。  因此优化的时候可以将B矩阵进行转置，将按列读取变为按行读取，从而提高空间局部性，提高Cache的命中率，提高运行效率。  **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；   参考书本，根据局部性原理，可以知道，读吞吐量可以体现读取某部分内存空间时存储系统的性能。根据之前所学过的知识，我们知道，Cache一般有三级，在读取速度上，L1>L2>L3>主存，当数据存放在不同位置的时候，数据的读取时间是会发现变化的，而且速度差距也是比较大的，即读吞吐量会发生阶跃性的改变，并且对于L1,L2,L3和主存来说读吞吐量是逐渐变小的。因此我们可以对不同数据集进行不断读写操作，然后测量程序的读吞吐量，当读吞吐量发生显著变化的时候，即可以推测Cache的层次结构以及容量的大致区间。   1. 测量方案及代码；   使用课本所给代码，通过调用run程序，通过传入的不同参数运行程序，分别查看不同size的读吞吐量以及stride的读吞吐量的对应关系，记录size与读吞吐量对应的关系和stride的读吞吐量的相应关系，分析两个关系，从而推测Cache的层次结构的容量和L1 Cache行的大小。     1. 测试结果；   读吞吐量与数据集大小的关系：    读吞吐量与步长的关系：     1. 分析过程；   通过分析读吞吐量与数据集大小的关系，可以看到，在前5个点的都吞吐量相差较小，但是当数据规模达到512的时候，曲线出现了较大幅度的下滑，当数据规模达到1536的时候，曲线又出现了一次较大的下滑，因此可以推测，0-512kb大致属于L1 Cache,512-1536kb大致属于L2 Cache，大于1536kb的时候为L3 Cache。  通过分析都吞吐量与步长的关系图，可以看到，当步长在16-192之间的时候，吞吐量大幅度下降，因此可以推测L1 Cache行的大小在16-192之间。   1. 验证实验结果。   打开任务管理器，查看各个缓存的大小，可以看到L1缓存为256kb，位于0-512kb之间，L2缓存为1.0M，位于512-1536kb之间，L3缓存为6.0M，大于1536kb。与实验结果一致    **3、选做：尝试测量你的x86机器TLB有多大？** |
| **五、实验结论与心得体会**  通过本次实验，我进一步加深了对Cache的理解，了解到了空间局部性对程序性能的影响，学会了如何利用空间局部性对程序进行优化。同时也了解了测量各级缓存的方法。  对于程序来说，若后续访问的空间地址在前面访问的空间地址附近，则说明该程序的空间局部性良好，利用这一特性，我们可以对程序进行优化，增强程序的空间局部性，降低对空间地址的访问时间，从而提高程序执行的效率，减小时间复杂度。 |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2022年 月 日 |
| 备注： |