**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： 计算机与软件学院所有专业**

**指导教师： 刘 刚**

**报告人：XXX 学号： XXXXXXXX 班级： 高性能班**

**实验时间： 2024年 6 月 8 日至 6 月 16 日**

**实验报告提交时间： 2024年 6 月 16 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）** |
| **四、实验结果及分析**  **1、分析Cache访存模式对系统性能的影响**  **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 |   **优化后的代码：**   |  |  | | --- | --- | | #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, it, jt, kt, itt, jtt, ktt;  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);  c[i\*size+j] = 0;  }  }    gettimeofday(&time1,NULL);  int N = 5; // 分块数  int T = size / N; // 每个分块大小  for (kt = 0; kt < N; ++kt) {  for (it = 0; it < N; ++it) {  for (jt = 0; jt < N; ++jt) {  ktt = kt \* T;  itt = it \* T;  jtt = jt \* T;  for (k = ktt; k < ktt + T; ++k) {  for (i = itt; i < itt + T; ++i) {;  temp = a[i\*size + k];  for (j = jtt; j < jtt + T; ++j) {  c[i\*size + j] += temp \* 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 |  |   表格 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 | 0.004110 | 0.419468 | 3.404261 | 11.763365 | 27.864664 | 62.562705 | 101.036562 | | 优化算法执行时间 | 0.002410 | 0.330059 | 2.305234 | 8.122128 | 18.656562 | 36.076145 | 63.143212 | | 加速比  speedup | 1.705394 | 1.270888 | 1.476753 | 1.448311 | 1.493558 | 1.734185 | 1.600118 |   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。    图表 1 普通矩阵乘法与及优化后矩阵乘法之间的性能对比    图表 2 不同数据规模下的加速比  **分析原因：**   1. 矩阵分块   传统的矩阵乘法在大矩阵的计算中，缓存利用率通常很低，这是因为大矩阵遍历元素的范围较大，导致缓存内存的元素频繁更换，如此导致缓存的低命中率。  而对矩阵进行分块，每次只操作较小的块数据，这样计算时，就可以把更规整的内存数据搬到缓存中，提高缓存命中率。   1. 优化循环顺序   在代码A中，内层循环遍历 k，使得b[k\*size+j]的访问模式是逐行跳跃，这很可能导致每次都需要加载新的缓存行，缓存局部性差，最终导致大量缓存未命中。  而优化后的代码，内层循环遍历j，使得b[k\*size+j]、c[i\*size + j]被连续访问。在遍历j时，b[k\*size+j]、c[i\*size + j]的访问模式有良好的空间局部性，能更好地利用缓存行效应（每行的访问大约会被缓存一次，然后重复使用，提高了命中率）。同时，变量temp存储a[i\*size + k]的值，减少了对a[i\*size + k]的反复访问，使a[i\*size + k]的加载更有效。  **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，对于每一个层级边界的数据存取会产生较大的速率变化，如果能找到产生速率变化的数据存取范围，就能大概确定Cache各层级的Cache大小。  （2）测量方案及代码  根据原理，写一个测量Cache函数模块measure\_cache，参数传一个内存块大小（模拟的cache层级大小），用整型数组进行存储。通过10000000次访问数组随机位置，记录下总时间。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。  在测出Cache第一级大小后，在此范围内测量L1 Cache行大小，通过函数模块cacheline，参数传一个内存块大小line（模拟的L1 Cache行大小），记录下以line内存大小为间隔地访问数组总时间，间隔访问数组整个过程需要循环line次，如此来保证不同间隔大小下，对数组的总访问次数是相同的。如果相邻的line测量得到的总时间出现较大波动，则说明达到L1 Cache行大小边界。   |  | | --- | | #include <iostream>  #include <chrono>  #include <algorithm>  #include <cmath>  #include <bits/stdc++.h>  // 使用 `chrono` 来进行时间测量  using namespace std;  using namespace std::chrono;  random\_device rd;//随机数生成  mt19937 gen(rd());  // 测层级，参数为内存块大小（对应cache大小），存于整型数组，  // 随机产生访问位置存于位置数组，若设置的内存块大小刚好是cache大小，  // cache层级边界时间会有较大改变  void measure\_cache(int size) {  int n = size/sizeof(int);  int\* arr = new int[n];  int\* pos = new int[100000000];  int sum = 0;  uniform\_int\_distribution<> num(0, n - 1); //大规模随机数0~n-1  duration<double> elapsed{0.0};    for(int i = 0; i < 10; i++){  for (int i = 0; i < 50000000; i++) {  pos[i] = num(gen); // 随机生成访问位置  }  auto start = high\_resolution\_clock::now();  for (int i = 0; i < 50000000; ++i) {  sum += arr[pos[i]];  }  auto end = high\_resolution\_clock::now();  elapsed += end - start;  }  cout << "Array size: " << size / 1024 << " KB, "  << "Elapsed time: " << elapsed.count()/10 << " s "  << endl;  }  void cacheline(int line){  int n = 100000000/sizeof(int);  int\* arr = new int[n];  int sum = 0;  int step = line / sizeof(int);    auto start = high\_resolution\_clock::now();  for(int j = 0; j < step; ++j){  for(int i = 0; i < n; i += step){  sum += arr[i];  }  }  auto end = high\_resolution\_clock::now();  duration<double> elapsed = end - start;  cout << "line size: " << line << " B, "  << "Elapsed time: " << elapsed.count() << " s "  << endl;  }  int main() {  int size[] = {8,16,32,64,128,256,  384,512,768,1024,1536,2048,  3072,4096,5120,6144,7168,8192,  10240,11264,12288,13824,15360,16896,  18432,19968,21504,23040,24576,26112};  // 测试不同大小的数组  for (int i = 0; i < 30; ++i) {  measure\_cache(size[i]\*1024);  }    int line[] = {4,8,16,24,32,48,64,80,104,128};    for(int i = 0; i < 10; ++i){  cacheline(line[i]);  }  return 0;  } |   （3）测试结果  模块measure\_cache测量结果如下。    图表 3 Cache层级测试结果  表格 2 Cache层级测试结果   |  |  |  |  | | --- | --- | --- | --- | | 数组大小/KB | 时间/s | 数组大小/KB | 时间/s | | 8 | 0.0479889 | 6144 | 0.106979 | | 16 | 0.0491943 | 7168 | 0.106897 | | 32 | 0.0520794 | 8192 | 0.105495 | | 64 | 0.0527545 | 10240 | 0.124000 | | 128 | 0.0495383 | 11264 | 0.133257 | | 256 | 0.0535714 | 12288 | 0.148097 | | 384 | 0.0554181 | 13312 | 0.174364 | | 512 | 0.0722899 | 14336 | 0.234565 | | 768 | 0.0834133 | 15360 | 0.374111 | | 1024 | 0.0935942 | 16384 | 0.522729 | | 1536 | 0.0941834 | 17408 | 0.552733 | | 2048 | 0.0958038 | 18432 | 0.527697 | | 3072 | 0.1025144 | 19456 | 0.526757 | | 4096 | 0.102703 | 20480 | 0.521497 |     图表 4 Cache层级测试结果（可视化）  模块cacheline测量结果如下。    图表 5 Cacheline测试结果  表格 3 Cacheline测试结果   |  |  |  |  | | --- | --- | --- | --- | | Cacheline大小/B | 时间/s | Cacheline大小/B | 时间/s | | 4 | 0.0399474 | 48 | 0.072588 | | 8 | 0.0413645 | 64 | 0.0848864 | | 16 | 0.0384788 | 80 | 0.10366 | | 24 | 0.0429402 | 104 | 0.118102 | | 32 | 0.0544505 | 128 | 0.142125 |     图表 6 Cacheline测试结果（可视化）  （4）分析过程  在Cache层级测试中，由数据可以知道，模拟的Cache大小在384KB-512KB首次出现较大时间增加，384KB前的Cache大小测试时间均稳定在0.05s附近，所以估计L1 Cache大小在384KB-512KB；8192KB-10240KB再次出现较大时间增加，8192KB前的Cache大小测试时间均稳定在0.1s附近，所以估计L2 Cache大小在8192KB-10240KB；14336KB-16384KB最后出现较大时间增加，虽然8192KB-14336KB的测试时间也在增大，但显然增大的幅度没有14336KB-16384KB测试时间增大的幅度大，所以估计L3 Cache大小在14336KB-16384KB。  Cacheline测试里，发现在间隔超过32B长度取数时时间增加较大，所以估计cacheline为32B。  （5）验证实验结果  通过任务管理器和CPU-Z工具进行查看，发现Cache的3级大小分别为512KB、4MB、16MB，L1 Cache的大小和L3 Cache的大小在前面推断的384KB-512KB和14336KB-16384KB范围内，但推断的L2 Cache的大小的范围8192KB-10240KB显然高于L2 Cache的实际大小，说明L2 Cache的推断大小不准确，有较大误差。这可能是因为不同的系统和 CPU设计对缓存的行为有优化，在对较大规模数据访问时会有预取机制，这会影响Cache测试的结果。    图表 7 Cache各级实际大小 |
| **五、实验结论**  通过对缓存的理解，我们可以利用缓存的局部性原理来优化矩阵乘法。具体来说，通过**矩阵分块**和**改变循环顺序**，可以大幅度减少缓存未命中，从而提高计算效率。**矩阵分块**技术将矩阵划分为较小的块，使得每次操作都能完全装入缓存，从而利用空间局部性；而**改变循环顺序**则调整内外层循环的次序，使得访问数据的模式更符合缓存访问特性，减少缓存的频繁更新，以此提高数据命中率。这些方法结合起来，可以显著优化矩阵乘法的性能，特别是在处理大规模数据时，能有效减少对主存的访问次数，提升计算速度。  同时，我们还可以通过代码，捕捉不同Cache层级间访问数据速度的差异，以此来判断不同Cache层级的大小和Cache行大小。具体来说，我们让数组大小从很小逐步增大，超越每一级Cache的大小时，访问时间会出现明显跳跃，记录这些访问时间的变化，可以推断各级Cache的大小，同理，用相同的方法也能测出Cache行大小。  通过这次实验，加深了我对缓存的层次结构、各级缓存的容量以及Cache Line的理解，也让我意识到，编写代码时应尽量考虑缓存优化，按行优先存储数据可以显著提高性能。 |

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