**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： 数学与统计学院**

**专业： 信息与计算科学（数学与计算机实验班）**

**指导教师： 罗胜**

**报告人： 王曦 学号： 2021192010 班级： 数计班**

**实验时间： 2023年 06月09日至06月23日**

**实验报告提交时间： 2022年06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cache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** 2. 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。  |  | | --- | | 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 |   下面的代码实现了矩阵a[]与矩阵b[]相乘, 结果存在矩阵c[]中, 与上述代码相同:   |  | | --- | | for (i = 0; i < size; i++) {  for (j = 0; j < size; j++)  c[i \* size + j] = 0;  }  for (j = 0; j < size; j++) {  for (k = 0; k < size; k++) {  temp = b[k \* size + j];  for (i = 0; i < size; i++)  c[i \* size + j] += a[i \* size + k] \* temp;  }  } |     代码A对矩阵乘法的实现是: 遍历矩阵的每一行和每一列, 求出结果矩阵对应位置的元素. 在空间局部性上, a[]每次访问的步长为1, 空间局部性良好; b[]每次访问的步长为size, size较大时空间局部性较差, 访问耗时长,  优化: 优化b[]访问的空间局部性, 使其每次访问的步长为1. 具体地, 遍历a[]的每个元素, 将每个元素的贡献累加到c[]中, 代码如下, 注意清空c[]:   |  | | --- | | for (i = 0; i < size; i++) {  for (j = 0; j < size; j++)  c[i \* size + j] = 0;  }  for (i = 0; i < size; i++)  for (k = 0; k < size; k++) {  temp = a[i \* size + k];  for (j = 0; j < size; j++)  c[i \* size + j] += temp \* b[k \* size + j];  } |  1. 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。  |  | | --- | | int test(int elems, int stride) {  long i, sx2 = stride \* 2, sx3 = stride \* 3, sx4 = stride \* 4;  long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;  long length = elems;  long limit = length - sx4;  /\*Combine 4 elements at a time\*/  for (i = 0; i < limit; i += sx4) {  acc0 = acc0 + data[i];  acc1 = acc1 + data[i + stride];  acc2 = acc2 + data[i + sx2];  acc3 = acc3 + data[i + sx3];  }  /\*Finish any remaining elements\*/  for (; i < length; i += stride)  acc0 = acc0 + data[i];  return ((acc0 + acc1) + (acc2 + acc3));  } |   上述代码为教材中的参考代码, 其中的test()函数模拟计算机访问内存的过程.  用elems \* size(long) / stride可得test()函数访问的内存空间的大小, 则只需记录调用test()函数的过程中消耗的时间即可.  为精确测量时间, 将测量的精度调整到时间周期的级别, 即记录test()函数的调用过程花费的时钟周期, 用时钟周期 / 电脑频率得到程序的运行时间.     |  | | --- | | /\* run - Run test(elems, stride) and return read throughput  \* "size" is in bytes, "stride" is in array elements,  \* and Mhz is CPU clock frequency in Mhz.  \*/  double run(int size, int stride, double Mhz) {  double cycles;  int elems = size / sizeof(long);  test(elems, stride); // Warm up the cache  cycles = fcy2(test, elems, stride, 0); // Call test(elems, stride)  return (size / stride) / (cycles / Mhz); // Convert cycles to MB/s  } |  |  | | --- | | cycles = fcy2(test, elems, stride, 0); // Call test(elems, stride) |   上述代码中的fcy2()函数是通过在C语言中内嵌汇编实现的记录test()函数的调用过程花费的时钟周期的函数, 通过汇编的接口记录开始运行时的时间戳, 运行完后再记录时间戳, 两时间戳相减即得运行的时间周期.  记录时间戳的函数:   1. access\_counter()函数通过C语言中嵌入汇编来获取当前程序运行到现在的时钟周期的时间戳, 结果保存在hi和lo两元素中. 汇编内部通过rdtsc命令实现, 返回当前程序运行到现在的时间周期, 将时间周期的高位保存到edx寄存器, 低位保存到eax寄存器.  |  | | --- | | void access\_counter(unsigned \*hi, unsigned \*lo) {  /\* Get cycle counter \*/  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"  : "=r"(\*hi), "=r"(\*lo)  : /\* No input \*/  : "%edx", "%eax");  } |  1. start\_counter()函数返回当前的时间戳, 对应开始时的时间戳. get\_counter()函数得到当前的时间戳并作差, 得到花费的时间周期.  |  | | --- | | void start\_counter() {  access\_counter(&cyc\_hi, &cyc\_lo);  }  double get\_counter() {  unsigned ncyc\_hi, ncyc\_lo;  unsigned hi, lo, borrow;  double result;  /\* Get cycle counter \*/  access\_counter(&ncyc\_hi, &ncyc\_lo);  /\* Do double precision subtraction \*/  lo = ncyc\_lo - cyc\_lo;  borrow = lo > ncyc\_lo;  hi = ncyc\_hi - cyc\_hi - borrow;  result = (double)hi \* (1 << 30) \* 4 + lo;  if (result < 0) {  fprintf(stderr, "Error: Cycle counter returning negative value: %.0f\n", result);  }  return result;  } |  1. 为防止单次测量不精确, 采用多次测量取最小值作为运行的时钟周期. 具体地, 每次循环前先清理缓存, 再调用test()函数进行缓存热身, 再开始正式测量, 将测量的结果放到一个数组中进行处理, 该过程在add\_sample中实现. 最后判断结果数组中的记录结果是否在超过测试次数前足够稳定, 若是则返回对应结果. 判断是否稳定的标准: 多次测量, 取最小的5个值, 若5个值的最大值与最小值之比小于一个常数, 就认为结果已足够稳定; 否则认为结果不够稳定, 重新测量.  |  | | --- | | do {  double cyc;  if (clear\_cache)  clear();  start\_comp\_counter\_tod();  f(param1, param2);  cyc = get\_comp\_counter\_tod();  add\_sample(cyc, k);  } while (!has\_converged(k, epsilon, maxsamples) && samplecount < maxsamples); |  1. 得到时钟周期后还需将初始周期转化为对应的运行时间, 这可通过CPU的时钟周期频率计算. 在linux系统中, 可在/proc/cpuinfo文件中得到, 通过字符串处理， 得到cpu频率的关键字后返回即可, 代码如下:  |  | | --- | | static char buf[MAXBUF];  FILE \*fp = fopen("/procuinfo", "r");  cpu\_ghz = 0.0;  if (!fp) {  fprintf(stderr, "Can't open /procuinfo to get clock information\n");  cpu\_ghz = 1.0;  return cpu\_ghz \* 1000.0;  }  while (fgets(buf, MAXBUF, fp)) {  if (strstr(buf, "cpu MHz")) {  double cpu\_mhz = 0.0;  sscanf(buf, "cpu MHz\t: %lf", &cpu\_mhz);  cpu\_ghz = cpu\_mhz / 1000.0;  break;  }  } |  1. 根据上述信息可得运行test()函数所需的时间, 进而得到计算机吞吐量和memory mountain. 2. 运行你的代码获得相应的测试数据.   将运行结果输出到文本, 用excel的三维图标可视化.   1. 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. 2. 根据测试数据来详细分析**L1 Cache行**有多少. |
| **四、实验结果及分析**  **1、分析Cache访存模式对系统性能的影响**  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 (s) | 0.004 | 0.507 | 4.743 | 19.199 | 37.134 | 112.479 | 185.441 | | 优化算法执行时间(s) | 0.003 | 0.355 | 2.839 | 9.604 | 17.712 | 44.429 | 76.510 | | 加速比  speedup | 1.333 | 1.428 | 1.671 | 1.999 | 2.097 | 2.532 | 2.424 |   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。   1. 上述结果用图表表示如下:     图1.1: 一般算法与优化算法的执行时间对比    图1.2: 加速比随矩阵大小的变化关系   1. 分析原因：    1. 在实验采用的所有数据规模下, 优化算法都比一般算法花费的时间少, 效率更高, 这表明: 优化算法优化了一般算法的空间局部性, 且空间局部性的优化可提高运行效率.    2. 加速比随数据规模的增大而增大, 因为数据规模较大, .即size增大时, 一般算法访问b[]的步长增大, 空间局部性的影响因一般算法是O(n³)的时间复杂度而被放大. 2. **测量分析出Cache 的层次结构、容量以及L1 Cache行有多少？** 3. 实验原理: 见实验内容和步骤部分. 4. 测量方案及代码: 见实验内容和步骤部分. 5. 测试结果: 6. 程序的测试内存取16 kB到128 MB, 步长取1到15, 得到下表所示的数据:     表2.1: 内存、步长与传输效率间的关系  可视化如下:    图2.2: 内存、步长与传输效率间的关系的可视化   1. 程序的测试内存取16 kB到128 MB, 步长取2^0到2^11, 得到下表所示的数据:     表2.3: 内存、步长与传输效率间的关系    图2.4: 内存、步长与传输效率间的关系的可视化   1. 分析过程:    * 1. 结果一: 程序根据不同的访问大小被分为4个山脊, 分别对应L1、L2、L3级缓存和主存. 每条山脊在步长为8或9时趋于平缓, 与教材中的存储器山模型大致相同.      2. 结果二: 步长较大时图2.4与图2.2有较大差异, 这是因为步长过长时, 对size较小的情况, 访问的元素较少, 时间开销主要在初始化上, 使得时间偏高. size逐渐增大时该现象逐渐减缓直至消失, 验证了算法的正确性. 2. 验证实验结果:    * 1. 用命令getconf -a|grep CACHE查看机器的不同级别的缓存, 结果如下图所示:   2022-06-17 19-35-51 的屏幕截图  图2.5: 机器的各级缓存   * + 1. 由上图: L1缓存分为两部分, 分别对应指令缓存和数据缓存. 对数据缓存部分, 三级缓存的大小分别为32KB、256KB、8MB. 图2.2和图2.4中四个山脊的分界线也分别在32KB、256KB、8MB处.  1. 根据测试数据分析使用的x86机器由几级Cache, 容量分别为多大.    * 1. 图象中有四个山脊, 分别对应L1、L2、L3级缓存和主存, 故机器有3级Cache.      2. 各Cache的容量为山脊的分界线, 分别为32KB、256KB、8MB. 2. 根据测试数据分析L1 Cache的行数.    * 1. 步长增大时, 计算机的吞吐量减小, 这与程序的空间局部性有关. 步长大于缓存的一行的block的字节数时, 吞吐量趋于稳定.      2. 由图象知: 步长大于8时吞吐量基本趋于稳定, 则缓存的一个block可容纳8个long类型的元素, 而long类型在x86-64系统中为8 Bytes, 则一个block的大小为8 \* 8 = 64 Bytes.      3. 行数 = 空间 / 块大小 = 32 kB / 64 B = 512 行. |
| **五、实验结论与心得体会**  通过这次实验，我对空间局部性和存储器结构有了更深入的了解。我了解了缓存通常的结构是怎样的，以及在通用系统中缓存是分为几级的，还学会了如何通过吞吐量来反映不同级别缓存的大小关系和块大小关系。  首先，我意识到空间局部性对编写性能优化代码非常重要。为了塑造具有良好空间局部性的代码，我们需要深入了解缓存的结构，并了解步长大小对程序性能的影响。通过进行矩阵乘法的优化实验，我更加深刻地理解了这个概念，明白了即使在同样的时间复杂度和相同的算法下，不同的空间局部性也会导致巨大的性能差异。  其次，在存储器结构方面，缓存的大小似乎是一个很底层的概念，很难以直观的方式展现出来。然而，我认识到各级缓存的大小与计算机的吞吐量有着密切的关系。通过调整步长和访问数组的大小，我们可以推导出存储器中各级缓存的大小关系，形象地认识到了不同级别存储器的存在。  此外，通过调整不同的步长大小，我还可以观察存储器之间传输的块的大小，从而得到每一级缓存的行数。我觉得这非常神奇，我们可以通过运行一个程序来获取底层硬件的信息，这让我更加意识到了硬件对程序性能的影响。  通过这次实验，我对缓存的理解更深入了。我意识到在以后的代码编写中，需要考虑空间局部性以提升代码的性能。这对我以后的学习和实践都非常有帮助。 |

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2022年 月 日 |
| 备注： |