

课程编号 1502760001-07

题目类型 实验 5

得分	教师签名	批改日期
	冯禹洪	

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(腾班)

指导教师: 冯禹洪

报告人: 黄亮铭 学号: 2022155028 班级: 腾班

实验时间: 2024 年 06 月 08 日至 06 月 28 日

实验报告提交时间: 2024 年 06 月 28 日

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

- (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。

首先使用VsCode打开main_a.c文件，阅读该文件中的代码，找到其中实现矩阵相乘的普通代码 A，如下图所示。

```

34     for(i=0;i<size;i++) {
35         for(j=0;j<size;j++) {
36             c[i*size+j] = 0;
37             for (k=0;k<size;k++)
38                 c[i*size+j] += a[i*size+k]*b[k*size+j];
39         }
40     }
    
```

图 1：普通代码 A

简单分析：代码 A 实现矩阵乘法的方法为依次遍历第一个矩阵（后称矩阵 a）的每一行和第二个矩阵（后称矩阵 b）的每一列，然后将对应位置的数据进行标量乘法，最后填写到新矩阵对应的位置即可。从空间局部性来看，矩阵 a 的每次访问步长为 1，因此矩阵 a 的空间局部性较好。矩阵 b 的每次访问步长为 size。Size 具体数据由用户输入决定，最坏情况是一个大数据，矩阵 b 的空间局部性因此较差，每一次访问可能需要较长的时间。

优化方向：考虑如何在保持矩阵 a 的空间局部性的前提下对矩阵 b 的空间局部性进行优化。

具体优化 I（自行设计）：我们首先矩阵 a 的每一个元素对于矩阵 c 的对应位置的贡献。为了实现这一点，我们需要更改循环的顺序，同时需要在三重循环外部对矩阵 c 进行清零操作。二重循环的运行时间对最终运行时间影响较小，因为二重循环的时间复杂度为 $O(n^2)$ ，三重循环的时间复杂度为 $O(n^3)$ 。具体代码见下图。

```

41     for (i = 0; i < size; i++) {
42         for (j = 0; j < size; j++) {
43             c[i * size + j] = 0;
44         }
45     }
46     for (i = 0; i < size; i++) {
47         for (k = 0; k < size; k++) {
48             float tmp = a[i * size + k];
49             for (j = 0; j < size; j++) {
50                 c[i * size + j] += tmp * b[k * size + j];
51             }
52         }
53     }
54 }

```

图 2：具体优化 I

具体优化 II：此种优化方法由文件`main_b.c`给出，具体代码见图 3。如果按行遍历一个矩阵（或者说是二维数组亦或是一维数组模拟矩阵），则空间局部性较好，如果按列遍历，则空间局部性较差。因此，很容易想到将第二个矩阵进行转置操作后，在进行矩阵乘法。

```

    for(i=0;i<size;i++) {
        for(j=0;j<size;j++) {
            b[i*size+j] = c[j*size+i];
        }
    }

    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];
        }
    }

```

图 3：具体优化 II

由文件`main_b.c`给出的代码中，对矩阵 b 进行转置操作这一步实际上可以进一步优化，将循环次数减少一半。但是在具体优化 I 中提到，二重循环的运行时间对总体运行影响较小，因此这里不再对附件的代码进行优化。

- (2) 改变矩阵大小，记录相关数据，并分析原因。

分别使用编译命令`gcc -o main_a main_a.c`和`gcc -o main_b main_b.c`，因为两种优化在同一个代码文件中修改。

在输入命令运行可执行文件时修改参数，运行结果见下图。

```

huangliangming_2022155028@ubuntu-2204:~/Desktop$ ./main_a 100
Executiontime=0.003708 seconds
huangliangming_2022155028@ubuntu-2204:~/Desktop$

```

矩阵大小	100	500	1000	1500	2000	2500	3000
一般算法执行时间/s	0.003708	0.385194	4.024414	18.5332	48.38033	127.0214	230.7918
优化算法 I 执行时间/s	0.002746	0.326212	2.780528	9.318378	22.30618	41.91596	72.76146
优化算法 II 执行时间/s	0.003576	0.458302	3.720665	12.47498	29.2768	56.52737	97.17852
加速比 I speedup	1.350327749	1.180809	1.447356	1.988886	2.16892	3.030383	3.171896
加速比 II speedup	1.036912752	0.840481	1.081638	1.485629	1.652514	2.247078	2.374926

将数据可视化后得到如下图像。

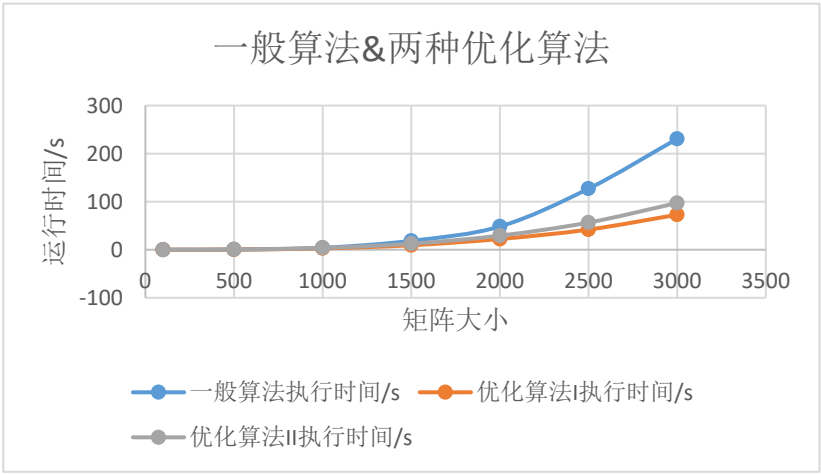


图 4a: 运行时间

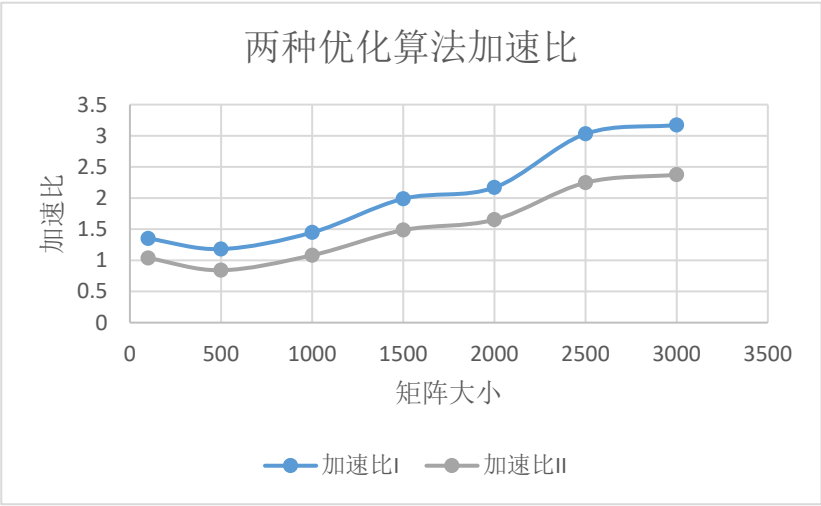


图 4b: 优化算法加速比

分析原因：① 两种优化后的算法花费的时间总体比一般算法花费的时间更少，并且几乎在所有数据规模下都体现了这一点，说明优化后的矩阵乘法在空间局部性上比一般算法确实更好。

② 具体优化 II 算法在小数据下的运行时间大于一般算法的原因可能是受矩阵

转置操作消耗的时间的影响。而在大数据下的运行时间明显优于一般算法，因为此时矩阵转置操作消耗的时间对总体运行时间的影响非常小。

③ 们发现优化加速比随着数据规模的变大，整体上呈现出增高的趋势。造成这一点的原因可能是当数据逐渐变大的时候，空间局部性的重要性体现的更加明显。矩阵 b 步长为 $size$ ，当 $size$ 变得越来越大，步长越来越大。一次访问的时间就会越大，而即使增大一点都会由于原算法是一个 $O(n^3)$ 算法的原因被放大到很大的情况。

④ 一般算法耗时高的原因：从空间局部性来看，矩阵 a 的每次访问步长为 1，CPU 访问数据的时候，多数都能从 Cache 中找到，即 Cache 命中，因此矩阵 a 的空间局部性较好。矩阵 b 的每次访问步长为 $size$ 。Size 具体数据由用户输入决定，最坏情况是一个大数据，如果 $size$ 大于 Cache 的容量，则每一次访问都不会命中，矩阵 b 的空间局部性因此较差，每一次访问可能需要较长的时间。

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；具体代码附件的压缩包中已经给出。该代码的方法是使用一个 $test()$ 函数模拟计算机访问内存的过程。

```
/* $begin mountainfuncs */
/*
 * test - Iterate over first "elems" elements of array "data"
 *       with stride of "stride".
 */
void test(int elems, int stride) /* The test function */
{
    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride) {
        result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}
```

图 5: $test()$ 函数

$size$ 的大小即通过 $test()$ 函数访问的内存空间大小已经知道，此时我们还需要记录调用 $test()$ 函数所消耗的时间。为了精确测量时间，代码将测量的精度调整到了时间周期的级别，使用 $fcyc2()$ 函数记录 $test()$ 函数调用过程中花费的时钟周期（对应代码中的变量为 $cycles$ ），然后使用 $时钟周期cycles/电脑频率Mhz$ ，进而得到程序的运行时间。

```

/*
 * run - Run test(elems, stride) and return read throughput (MB/s).
 *       "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(double);

    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
/* $end mountainfuns */

```

图 6: run()函数

我们通过调整size和stride即可获得不同的数据结果。

(2) 运行你的代码获得相应的测试数据;

原始数据绘制出的图像因为参数（步长）问题比较难观察，因此我对参数进行了一定的修改，具体修改见下图。

```

8 #define MINBYTES (1 << 14) /* Working set size ranges from 2 KB 14*/
9 #define MAXBYTES ((1 << 27)) /* ... up to 64 MB 27*/
10 #define MAXSTRIDE 15 /* Strides range from 1 to 64 elems */
11 #define MAXElems MAXBYTES/sizeof(double)

```

图 7: 存储山参数

得到的测试数据见下图。

Open		1.txt	
		~/Desktop/mountain	
1 Clock frequency is approx. 2688.1 Mhz			
2 Memory mountain (MB/sec)			
3	s1	s2	s3
4 128m	7550.6	7137.2	5105.9
5 64m	10854.5	5625.0	4416.2
6 32m	7564.1	5921.2	5143.6
7 16m	8677.4	8397.9	7682.5
8 8m	9304.0	9284.0	9299.4
9 4m	9311.0	9310.7	9309.2
10 2m	9312.9	9309.5	9315.1
11 1024k	9314.0	9316.2	9317.4
12 512k	9271.5	9318.9	9327.2
13 256k	9318.9	9318.0	9332.0
14 128k	9315.5	9350.6	9353.4
15 64k	9345.6	9372.5	9398.3
16 32k	9370.5	9406.5	9428.1
17 16k	9434.7	9541.0	9670.3

图 8: 测试数据

除上述操作以外，我们还可以修改步长的变化幅度，将其幅度从 1 修改为 2。也即为修改每次循环后stride增加的数值。具体修改见下图（51 行）。

```

37 for (stride = 1; stride <= MAXSTRIDE; stride+=2)
38     printf("%d\t", stride);
39     printf("\n");
40
41 /* $begin mountainmain */
42 for (size = MAXBYTES; size >= MINBYTES; size >= 1) {
43     /* $end mountainmain */
44     /* Not shown in the text */
45     if (size > (1 << 20))
46         printf("%dm\t", size / (1 << 20));
47     else
48         printf("%dk\t", size / 1024);
49
50 /* $begin mountainmain */
51 for (stride = 1; stride <= MAXSTRIDE; stride+=2) {
52     printf("%.1f\t", run(size, stride, Mhz));
53
54 }
55 printf("\n");
56 }
57 exit(0);

```

图 9: 修改部分

得到的测试数据见下图。

[illegible]

图 10: 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？将上述两种不同操作得到的数据进行可视化，如下图所示。

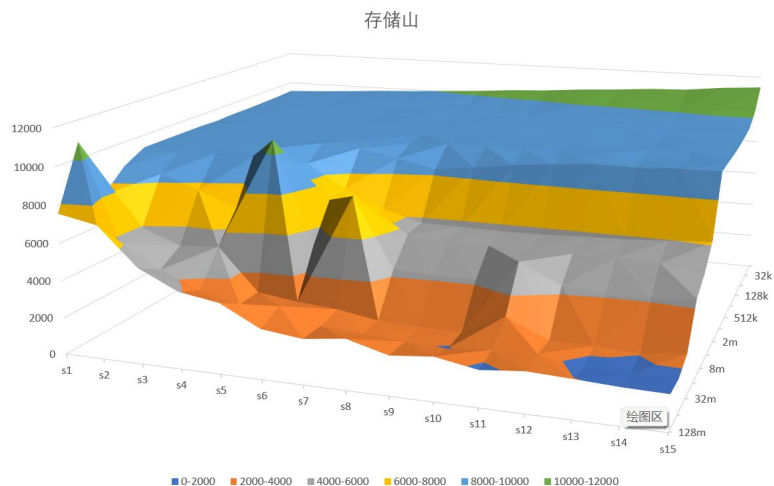


图 11a: 可视化结果

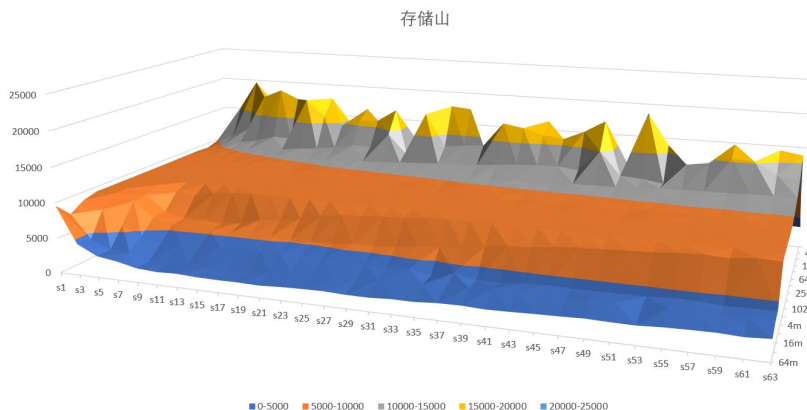


图 11b: 可视化结果

由上述两张图像可以看出，一共有三级缓存，不同缓存的分界线的范围分别为 $32K < L1 < 64K$ 、 $512K < L2 < 2M$ 和 $16M < L3 < 32M$ 。

下面我们在linux下输入命令`getconf -a|grep CACHE`来验证由图像得到的缓存大小的范围是否正确。

```

huangliangming_2022155028@ubuntu-2204:~/Desktop/mountain$ getconf -a|grep CACHE
LEVEL1_ICACHE_SIZE                32768
LEVEL1_ICACHE_ASSOC                64
LEVEL1_ICACHE_LINESIZE            64
LEVEL1_DCACHE_SIZE                49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE            64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 10
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                   25165824
LEVEL3_CACHE_ASSOC                  12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                   0
LEVEL4_CACHE_ASSOC                  0
LEVEL4_CACHE_LINESIZE              0

```

图 12: 缓存相关参数

L1 缓存分为了两部分，分别对应指令缓存和数据缓存，我们关注数据缓存。缓存 L1、L2 和 L3 分别对应的大小就是 48K，1280K，24576K。

由此，我们得出图像显示的不同缓存的分界线的范围与通过指令查看的实际缓存基本相符。

(4) 根据测试数据来详细分析 **L1 Cache 行**有多少？

从第一个测试结果可以看到，当步长不断变大的时候，计算机的吞吐量也在不断的变小，这个与程序的空间局部性有关。当步长大于缓存的一行的块所具有的字节的数量的时候就会趋于稳定。

可以得到步长在大于 48 的时候计算机的吞吐量基本趋于稳定，所以可以推测缓存的一个块可以容纳 48 个 float 类型大小的元素，而 float 类型的大小在 x86-64 中为 4 字节，所以对应的大小就是 $4 \times 48 = 192$ 字节，即一个块可以容纳 192 个字节。

已知：行数 = 空间大小 / 块大小。所以总共的行数为 $48K / 192B = 64$ ，即总共 64 行。

3、尝试测量你的 x86 机器 TLB 有多大？（选作）

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

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	0.003708	0.385194	4.024414	18.5332	48.38033	127.0214	230.7918
优化算法 I 执行时间/s	0.002746	0.326212	2.780528	9.318378	22.30618	41.91596	72.76146
优化算法 II 执行时间/s	0.003576	0.458302	3.720665	12.47498	29.2768	56.52737	97.17852
加速比 I speedup	1.350327749	1.180809	1.447356	1.988886	2.16892	3.030383	3.171896
加速比 II speedup	1.036912752	0.840481	1.081638	1.485629	1.652514	2.247078	2.374926

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。
将数据可视化后得到如下图像。

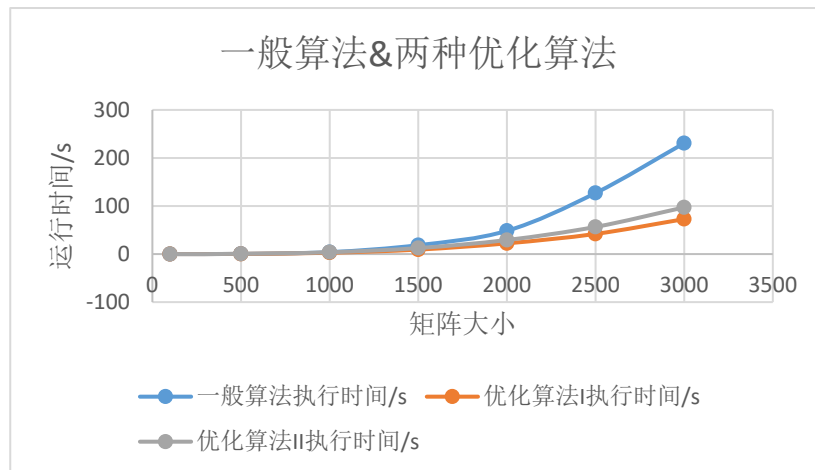


图 13: 可视化结果

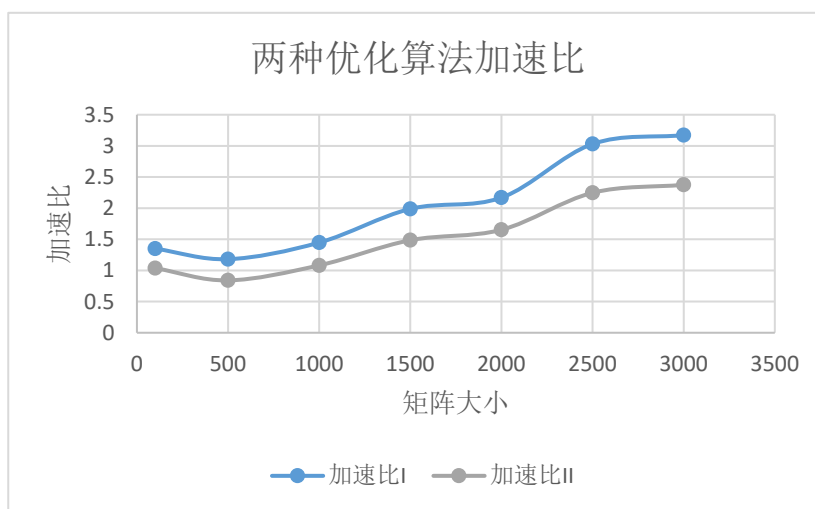


图 14: 可视化结果

分析原因:

- ① 两种优化后的算法花费的时间总体比一般算法花费的时间更少，并且几乎在所有数据规模下都体现了这一点，说明优化后的矩阵乘法在空间局部性上比一般算法确实更好。
- ② 具体优化 II 算法在小数据下的运行时间大于一般算法的原因可能是受矩阵转置操作消耗的时间的影响。而在大数据下的运行时间明显优于一般算法，因为此时矩阵转置操作消耗的时间对总体运行时间的影响非常小。
- ③ 我们发现优化加速比随着数据规模的变大，整体上呈现出增高的趋势。造成这一原因可能是当数据逐渐变大的时候，空间局部性的重要性体现的更加明显。矩阵 b 步长为 $size$ ，当 $size$ 变得越来越大，步长越来越大。一次访问的时间就会越大，而即使增大一点都会由于原算法是一个 $O(n^3)$ 算法的原因被放大到很大的情况。
- ④ 一般算法耗时高的原因：从空间局部性来看，矩阵 a 的每次访问步长为 1，CPU 访问数据的时候，多数都能从 Cache 中找到，即 Cache 命中，因此矩阵 a 的空间局部性较好。矩阵 b 的每次访问步长为 $size$ 。Size 具体数据由用户输入决

定，最坏情况是一个大数据，如果 size 大于 Cache 的容量，则每一次访问都不会命中，矩阵 b 的空间局部性因此较差，每一次访问可能需要较长的时间。

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：

size 的大小即通过 `test()` 函数访问的内存空间大小已经知道，此时我们还需要记录调用 `test()` 函数所消耗的时间。为了精确测量时间，代码将测量的精度调整到了时间周期的级别，使用 `fcyc2()` 函数记录 `test()` 函数调用过程中花费的时钟周期（对应代码中的变量为 `cycles`），然后使用 `时钟周期cycles/电脑频率Mhz`，进而得到程序的运行时间。

我们通过调整 `size` 和 `stride` 即可获得不同的数据结果。

继续深入阅读 `fcyc2()` 函数的代码。

发现执行 `fcyc2()` 函数实际上是在执行 `fcyc2_full_tod()` 函数。首先执行 `init_sampler()` 函数初始化采样器，分配内存并重置采样计数。然后进入计时循环，因为上述参数的设置，选择分支只会进入 `compensate` 为 0 的分支。进入该分支后，首先会清除缓存，然后执行测试函数（即 `test()`）并计时。再将获取的运行时间存储到数组中，同时调用 `has_convcrged()` 函数检查数据是否收敛或采样次数是否达到最大值。最后返回结果。

具体计算时间周期的原理。

`access_counter()` 函数可以通过嵌入汇编得到当前程序运行到现在的时钟周期的时间戳，将结果返回到 `hi` 和 `lo` 两个元素中。汇编语言内部通过 `rdtsc` 命令实现，会返回当前程序运行到现在的时间周期，然后将时间周期的高位返回到 `%edx` 寄存器，将时间周期的低位返回到 `%eax` 寄存器。

```
85 void access_counter(unsigned *hi, unsigned *lo)
86 {
87     /* Get cycle counter */
88     asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
89         : "=r" (*hi), "=r" (*lo)
90         : /* No input */
91         : "%edx", "%eax");
92 }
```

图 15: `access_counter()` 函数

代码使用 `start_counter()` 函数调用 `access_counter()` 函数获得开始的时间戳，使用 `get_counter()` 函数调用 `access_counter()` 函数获得当前时间戳。两者之差的绝对值即为消耗的时间周期。

(2) 测量方案及代码：

具体代码附件的压缩包中已经给出。该代码的方法是使用一个 `test()` 函数模拟计算机访问内存的过程。

```

/* $begin mountainfun */
/*
 * test - Iterate over first "elems" elements of array "data"
 *        with stride of "stride".
 */
void test(int elems, int stride) /* The test function */
{
    int i;
    double result = 0.0;
    volatile double sink;

    for (i = 0; i < elems; i += stride) {
        result += data[i];
    }
    sink = result; /* So compiler doesn't optimize away the loop */
}

```

图 16: `test()`函数

`size`的大小即通过`test()`函数访问的内存空间大小已经知道,此时我们还需要记录调用`test()`函数所消耗的时间。为了精确测量时间,代码将测量的精度调整到了时间周期的级别,记录`test()`函数调用过程中花费的时钟周期(对应代码中的变量为`cycles`),然后使用`时钟周期cycles/电脑频率Mhz`,进而得到程序的运行时间。

```

/*
 * run - Run test(elems, stride) and return read throughput (MB/s).
 *       "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(double);

    test(elems, stride); /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0); /* call test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
/* $end mountainfun */

```

图 17: `run()`函数

我们通过调整`size`和`stride`即可获得不同的数据结果。

(3) 测试结果:

将两种不同操作得到的数据进行可视化,如下图所示。

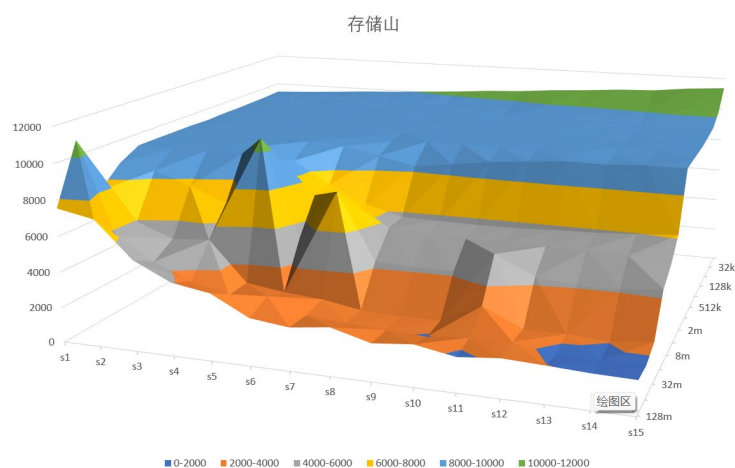


图 18a: 可视化结果

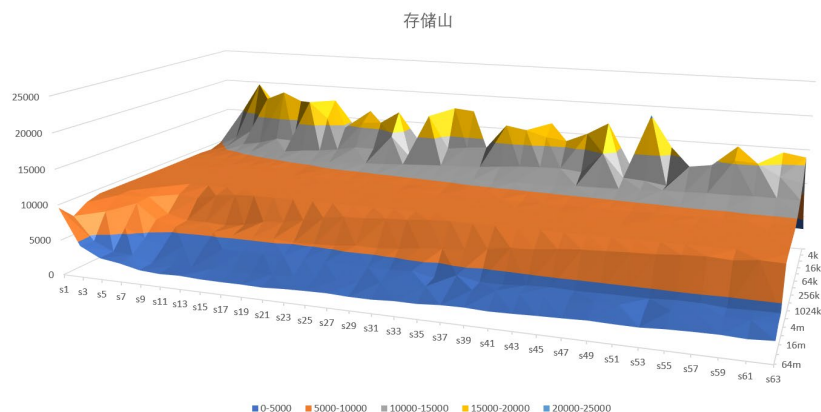


图 19b: 可视化结果

由上述两张图像可以看出，一共有三级缓存，不同缓存的分界线的范围分别为 $32K < L1 < 64K$ 、 $512K < L2 < 2M$ 和 $16M < L3 < 32M$ 。

(4) 分析过程:

可以得到程序根据不同的访问大小被分为了四区块，应分别对应 L1, L2, L3 级缓存以及主存。

异常情况分析: size 较小的情况下，访问的元素数量较少，大部分时间周期开销反而在一些初始化或矩阵转置的操作上，从而使得运行时间偏高，吞吐量下低。

(5) 验证实验结果。

下面我们在 linux 下输入命令 `getconf -a | grep CACHE` 来验证由图像得到的缓存大小的范围是否正确。

```

huangliangming_2022155028@ubuntu-2204:~/Desktop/mountain$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE      64
LEVEL1_DCACHE_SIZE          49152
LEVEL1_DCACHE_ASSOC         12
LEVEL1_DCACHE_LINESIZE      64
LEVEL2_CACHE_SIZE            1310720
LEVEL2_CACHE_ASSOC           10
LEVEL2_CACHE_LINESIZE        64
LEVEL3_CACHE_SIZE            25165824
LEVEL3_CACHE_ASSOC           12
LEVEL3_CACHE_LINESIZE        64
LEVEL4_CACHE_SIZE            0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE

```

图 20: 缓存相关参数

L1 缓存分为了两部分，分别对应指令缓存和数据缓存，我们关注数据缓存。缓存 L1、L2 和 L3 分别对应的大小就是 48K, 1280K, 24576K。

由此，我们得出图像显示的不同缓存的分界线的范围与通过指令查看的实际缓存基本相符。

3、尝试测量你的 x86 机器 TLB 有多大？（选做）

具体代码见下图。

```
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <sched.h>
#define MAX_NUM_PAGES 256
#define TRIALS 10000000

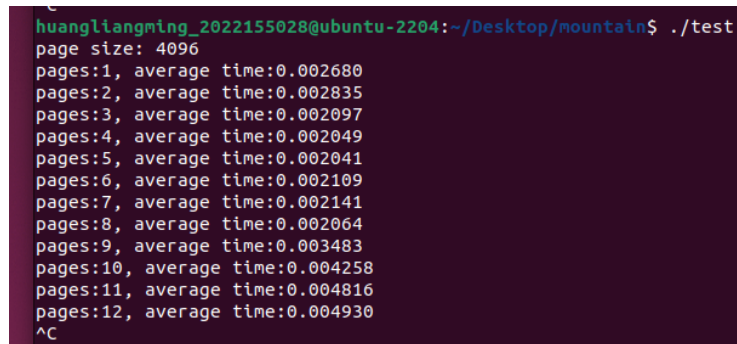
void trials(int jump, int num_pages, int arr[])
{
    struct timeval start, end;
    gettimeofday(&start, NULL);
    for (int i = 0; i < TRIALS; i++)
    {
        for (int i = 0; i < num_pages * jump; i += jump)
        {
            arr[i] += 1;
        }
    }
    gettimeofday(&end, NULL);
    uint64_t trial_time = (end.tv_sec - start.tv_sec) * 1000000
        + end.tv_usec - start.tv_usec;
    double one_trial_time = trial_time
        / (double)(num_pages * TRIALS);
    printf("pages:%d, average time:%f\n",
        num_pages, one_trial_time);
}

int main(int argc, char *argv[])
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(0, &cpuset);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &cpuset))
    {
        fprintf(stderr, "Error setting cpu affinity\n");
        exit(EXIT_FAILURE);
    }
    int page_size = getpagesize();
    int jump = page_size / sizeof(int);
```

```
int *arr = (int *)calloc(MAX_NUM_PAGES * jump, sizeof(int));
for (int pages = 1; pages < MAX_NUM_PAGES; pages++)
{
    trials(jump, pages, arr);
}
}
```

图 20: 代码

代码分析: 首先将程序绑定到指定的 CPU 核, 减少干扰。然后获取页面大小和计算步长, 同时分配一个足够大的数组, 用于存储不同页面的数据。最后测量访问时间, 这里多次测量取平均)。如果访问的页面数超过 TLB 的大小时, 访问时间会显著增加, 进而推出 TLB 的大小。



```
huangliangming_2022155028@ubuntu-2204:~/Desktop/mountain$ ./test
page size: 4096
pages:1, average time:0.002680
pages:2, average time:0.002835
pages:3, average time:0.002097
pages:4, average time:0.002049
pages:5, average time:0.002041
pages:6, average time:0.002109
pages:7, average time:0.002141
pages:8, average time:0.002064
pages:9, average time:0.003483
pages:10, average time:0.004258
pages:11, average time:0.004816
pages:12, average time:0.004930
^C
```

图 21: 访问页时间

由上图可知: 页的大小为 4096B。页 9 和页 10 之间的访问时间显著增加。可以认为 TLB 的大小的范围为 $[4096 \times 9B, 4096 \times 10B]$, 即 $[36K, 40K]$ 。

五、实验结论与心得体会

1. 通过这次实验, 我对于空间局部性以及存储器的结构有了更深刻的了解, 了解了缓存的结构了解了通用系统下缓的分级情况, 以及了解了吞吐量是如何反映出各级缓存的大小关系以及块大小关系的。
2. 通过矩阵乘法的优化实验, 我明白同样的时间复杂度相同的算法仅是空间局部性不同, 运行时间就可以相差甚远。
3. 存储器结构是一个非常底层的概念, 很难直观的展示在我们面前, 但是实际上各级缓存的大小与计算机的吞吐量有莫大的关系, 我们可以通过调整步长和访问的数组大小进而得到存储器各缓存的大小关系, 从而形象地认识到各级存储器的存在。

指导教师批阅意见：

成绩评定：

指导教师签字：

2024 年 月 日

备注：