**2. Cache配置参数测试**

**题目要求：**

设计并运行一组数据密集型程序，通过分析观察到的性能变化，推导出你计算机上CPU的Cache主要参数配置：

（1）层级数

（2）各层：容量、块大小、组相联度、命中时间、缺失代价

通过操作系统或库提供的API函数、命令或文件，获得该处理器的Cache参数配置，与你通过自己设计的程序运行得到的参数进行对比，若有不吻合的地方，试着分析原因。

**2.1 层级数与Cache容量**

由于每一级缓存的大小是有限的，而且通常缓存的空间比较小，当需要访问的数组空间小于缓存空间时，CPU只需要将数组从主存读取到缓存中，之后重复访问该数组时不需要再访存，因此重复访问数组的速度很快；而当访问的数组空间大于缓存空间时，已经放入缓存中的数据会被之后的数组元素替换，导致每次重复访问该数组时都要访存，因此执行速度急剧下降。因此，可以通过重复遍历大小逐渐增加的数组序列，程序执行速度大幅度下降之处的数组大小即为缓存空间大小。

本次实验通过单字节的整型数组，进行读写测试来获取各级Cache的大小，数组大小从64MB到4KB进行遍历，实现代码如下：

#include <time.h>

#include <stdio.h>

#define KB 1024

#define MB KB \* KB

#define dataSize 64 \* MB

#define minSize 4 \* KB

char x[dataSize];

clock\_t startTime, endTime;

int numOfLoop, loopFactor, repeat;

double numBytesToMove, timeTaken;

double getTimeDifference(clock\_t start, clock\_t end)

{

return (end / 1000.0) - (start / 1000.0);

}

double loop(int numOfLoop, int numBytesToMove, int bytesToJump)

{

int i;

startTime = clock();

for (repeat = 1; repeat <= numOfLoop; repeat++) {

for (i = 0; i < numBytesToMove; i += bytesToJump) {

x[i]++;

}

}

endTime = clock();

return getTimeDifference(startTime, endTime);

}

int main()

{

FILE \*fp = fopen("result.txt", "w");

double times[20], sizes[20];

double sizeFactor, k, notedPercentDrop, timeforFiftyLoops;

int threshold, nextCacheLikely = 1;

int i, j;

timeforFiftyLoops = loop(50, dataSize, 64);

loopFactor = (int)1000 / timeforFiftyLoops;

sizeFactor = 3.0 / 4; k = 8.0 / 9;

i = 0;

numBytesToMove = dataSize;

while (numBytesToMove >= minSize) {

numOfLoop = (dataSize / numBytesToMove) \* loopFactor;

timeTaken = loop(numOfLoop, (int)numBytesToMove, 64);

times[i] = timeTaken;

sizes[i] = numBytesToMove; i++;

if (numBytesToMove > 2 \* MB) {

numBytesToMove \*= sizeFactor;

sizeFactor \*= k;

k = 1 / k;

}

else {

numBytesToMove /= 2;

}

printf("numBytesToMove:%8d, time: %lfs\n", (int)numBytesToMove, timeTaken);

fprintf(fp, "%8d\t%lf\n", (int)numBytesToMove, timeTaken);

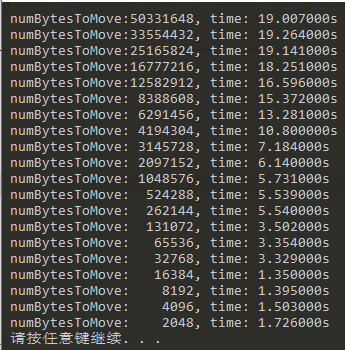
}

fclose(fp);

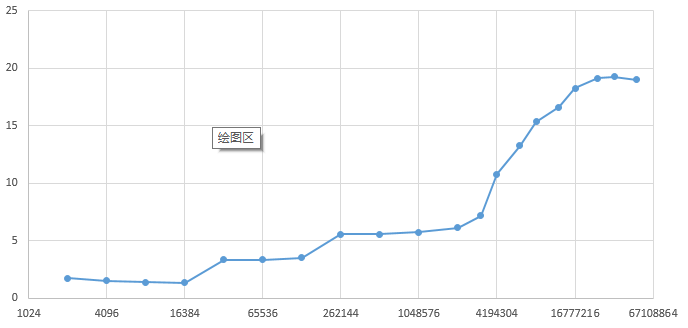
return 0;

}

运行结果如下：



对其进行绘图分析：



从折现图中我们可以很的看出，有3处变化比较陡峭的地方，位置分别为16K-32K、128K-256K和4M-8M处，由此我们可以知道Cache的层数为3层，其大小分别为32K、256K和6M。由于测试机拥有4个核心，且L1 Cache被分为数据Cache和指令Cache两部分且大小相同，L3 Cache四个核心共用，因此本机的L1 Cache、L2 Cache和 L3 Cache大小分别为：256K、1M和6M。

**2.2 块大小**

因为不命中时间要远远大于命中时间，所以假定块大小为64Byte， 因此步长为2时平均的运行时间应该是步长1的2倍，以此类推，当步长为64时最接近2倍。当步长大于64后，都不会命中，所以时间不会增长很快，由此可以得到块的大小，代码如下：

#include <time.h>

#include <stdio.h>

#define KB 1024

#define MB KB \* KB

#define dataSize 64 \* MB

#define minSize 4 \* KB

char x[dataSize];

clock\_t startTime, endTime;

int blockSize;

double timeTaken, prevTime;

double getTimeDifference(clock\_t start, clock\_t end)

{

return (end / 1000.0) - (start / 1000.0);

}

double loop(int numOfLoop, int numBytesToMove, int bytesToJump)

{

int i;

startTime = clock();

for (int repeat = 1; repeat <= numOfLoop; repeat++) {

for (i = 0; i < numBytesToMove; i += bytesToJump) {

x[i]++;

}

}

endTime = clock();

return getTimeDifference(startTime, endTime);

}

int main()

{

int i, j, currentBlock, sizes[8];

double ratio = 0; currentBlock = 1;

timeTaken = 0, prevTime = 0;

int numOfLoop = 20;

i = 0;

while (currentBlock <= 256) {

prevTime = timeTaken;

timeTaken = loop(numOfLoop, dataSize, currentBlock);

printf("当步长为%d时，耗时为:%lfs\n", currentBlock, timeTaken);

sizes[i++] = currentBlock;

currentBlock \*= 2;

numOfLoop \*= 2;

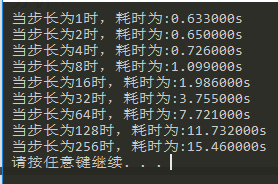
blockSize = currentBlock;

}

return 0;

}

运行结果如下：



通过上面的分析我们知道，本机Cache的块大小为64Byte。

**2.3 组相连度**

首先我们假设每个缓存有16个块，则对于直接映射的情况（其中0为未命中，1为命中）：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 块号 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 块 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 映射情况 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 48 | … |  |  |  |  |  |  |  |  |  |  |  |  |  |

对于2路组相连映射的情况:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 块号 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 块 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 映射情况 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | … |  |  |  |  |  |

对于4路组相连映射的情况:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 块号 | 0 | 1 | 2 | 3 |
| 块 | 1 | 1 | 1 | 1 |
| 映射情况 | 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |
| 32 | 33 | … |  |

由上图我们可以看出，对于编号为0和16的块，不停访问，在直接映射中会产生冲突，但2路、4路、8路……则不会冲突；对于编号为0、16、32、48的块，不停访问，在1路、2路中会产生冲突，但对于4路、8路、16路……则不会冲突，以此类推，我们让程序每次都出现可能导致块缺失的情况，即访问编号为0、16的多次，计算平均访问时间；访问0、16、32、48的块多次，计算平均访问时间；访问编号为0、16、32、48、64、80的块多次，计算平均访问时间……假定CPU为4路组相联，则不停访问0、16、32、48不会出现缺失，而再增大为0、16、32、48、64，就会出现不停缺失的情况，据此思路，写出代码如下：

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#define Lenght (1<<8<<10<<10)

int loop = (1 << 27); //循环次数，为了更准确

int main()

{

clock\_t start, end;

int \*array = (int \*)malloc(Lenght \* sizeof(int));

for (int ways = 2; ways <= 16; ways += 2) {

start = clock();

for (int j = 0; j < loop; ++j)

for (int i = 0; i < ways; ++i)

array[(i << 9) << 6] = 0; //32j L1 cache, 16 int/blocj

end = clock();

int total\_time = end - start;

start = clock(); //减去移位运算时间，以得到更准确的结果

for (int j = 0; j < loop; ++j)

for (int i = 0; i < ways; ++i)

(i << 9) << 6;

end = clock();

printf("%2d路组相连测试总耗时 %fms\n",

ways, (double)(total\_time - (end - start)) / (ways));

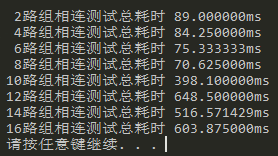
}

free(array);

return 0;

}

运行结果如下：



通过上面的分析我们知道，耗时最短的即为本机L1 Cache的组相连度，既8路组相连。

**2.4 命中时间**

命中时间就是当需要访问的数据在cache中时，程序访问所需要的时间。因此，我们先选取适当大小的数组，将其从内存取到 Cache 中，选取的数据长度不能太大，也不能太小，太长了会超出 Cache 大小,太短了就会只访问了 L1 Cache,而 L2、L3 Cache的访问时间比 L1 Cache长，会导致很大的测试误差。因此，我们选取了4M/4Byte=1M的数组，以16为步长进行跳读，循环1024\*1024次求得总时间后除以1024\*1024和时钟频率就得到命中时间。

实现代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

\_\_int64 \_\_declspec(naked) read\_time\_stamp\_counter()

{

\_\_asm cpuid;//使用汇编指令测试时间精度更高

\_\_asm rdtsc;

\_\_asm ret;

}

#define SIZE\_1KB (1024)

#define SIZE\_1MB (1024\*1024)

#define NUMBER 128\*SIZE\_1MB

int main()

{

long long ticks\_1, ticks\_2;

long perus;

int i, step, s;;

int \* a;

ticks\_1 = (double)read\_time\_stamp\_counter();

Sleep(1000);

ticks\_2 = (double)read\_time\_stamp\_counter();

perus = (ticks\_2 - ticks\_1) / (1000 \* 1000);//测试cpu当前时钟频率

//测试命中时间

a = (int \*)malloc(NUMBER\*sizeof(int));

int len = SIZE\_1MB;//访问的数组大小

//先将数组取到cache中

for (i = 0; i < len / 16; i++) {

a[(i \* 16) & (len)]++;//linesize 是64B，16 means linesizesizeof(int) = 16

}

ticks\_1 = (double)read\_time\_stamp\_counter();

for (i = 0; i < SIZE\_1MB; i++) {

a[(i \* 16) & (len)]++;//\*16 means linesizesizeof(int) = 16

}

ticks\_2 = (double)read\_time\_stamp\_counter();

printf("命中时间为:%.8f us\n", (double)(ticks\_2 - ticks\_1) / perus / SIZE\_1MB);

return 0;

}

运行结果如下：



由此可得命中时间为 2.15ns。

**2.5 缺失代价**

缺失代价即指需要要访问的数据不再cache中，程序访问这些数据时所需要的时间，因此可以通过循环地访问不同位置的数据来计算缺失代价，另外，我们设置数据读取步长为1024，这样可以排除硬件预取带来的影响。

实现代码如下：

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

\_\_int64 \_\_declspec(naked) read\_time\_stamp\_counter()

{

\_\_asm cpuid;//使用汇编指令测试时间精度更高

\_\_asm rdtsc;

\_\_asm ret;

}

#define SIZE\_1KB (1024)

#define SIZE\_1MB (1024\*1024)

#define NUMBER 128\*SIZE\_1MB

int main()

{

long long ticks\_1, ticks\_2;

long perus;

int i, step, s;;

int \* a;

ticks\_1 = (double)read\_time\_stamp\_counter();

Sleep(1000);

ticks\_2 = (double)read\_time\_stamp\_counter();

perus = (ticks\_2 - ticks\_1) / (1000 \* 1000);//测试cpu当前时钟频率

//测试缺失代价

a = (int \*)malloc(NUMBER\*sizeof(int));

step = SIZE\_1KB \* 4;

s = 1;

ticks\_1 = (double)read\_time\_stamp\_counter();

for (i = 1; i < SIZE\_1KB; i++) {

a[s] = 1;

s = s + step;

}

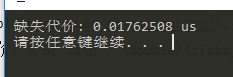
ticks\_2 = (double)read\_time\_stamp\_counter();

printf("缺失代价: %.8f us\n", (double)(ticks\_2 - ticks\_1) / perus / SIZE\_1KB);

return 0;

}

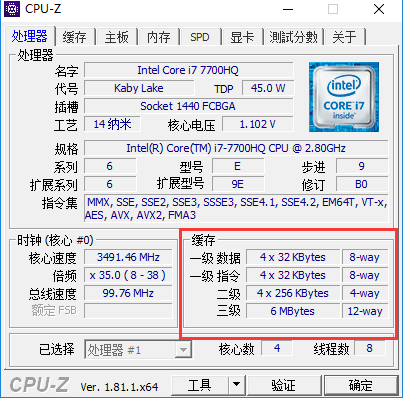
运行结果如下：

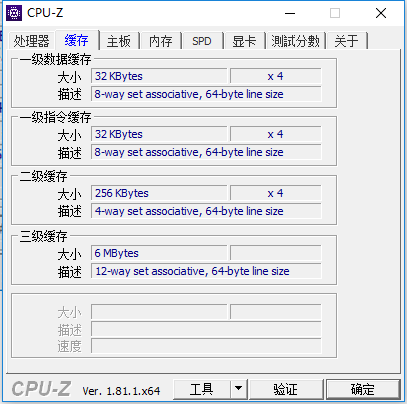


由此可得命中时间为 17.6ns，大约命中时间的8倍左右，所以一但发生缺失，将会产生很大的代价

**2.5 真实参数对比**

我们采用软件CPU-Z对电脑cache参数进行了获取，结果如下图所示：





通过分析对比可以看出，通过设计并运行一组数据密集型程序得出的Cache参数与真实的Cache参数基本一致。