

高等计算机体系结构

实验报告



小组成员：高俊龙 1701213594

夏雅娴 1701213643

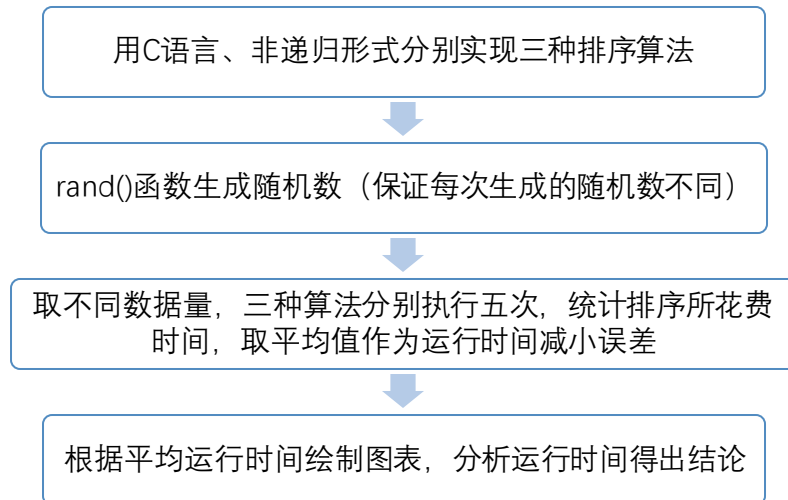
姜 波 1701213603

一、 排序算法性能比较

1.1 实验方案

1.1.1 思路及框架

解决问题思路及框架大致如下：



1.1.2 算法原理及实现

(1) 快速排序

特点：不稳定，时间复杂度 最理想 $O(n \log n)$ 最差时间 $O(n^2)$

原理：确定基准变量，从待排序数组选择大于和小于基准变量的成员，交换两成员位置，同时把基准变量放置于两成员之间，这样既可保证得到一个：小于基准变量，基准变量，大于基准变量的序列；利用递归进行循环，既可快速排序。

代码实现：

(2) 插入排序

特点：稳定，时间复杂度 $O(n^2)$

原理：以数组首端为已排序好的数列（虽然只有一个），从数组首端地址之后的待排序数组成员找到已经排序好的数列的插入位置，将待排序成员插入该位置，位置之后的成员依次往后移一位。

代码实现：

```
void quicksort(int* A, int n)
{
    typedef struct
    {
        int beg;
        int end;
    } sbe;

    sbe* arr = (sbe *)malloc(n * sizeof(sbe));
    int m;
```

```

int p;
int beg, end, i, j;
m = 0;
arr[0].beg = 0;
arr[0].end = n - 1;
for (;;)
{
    if (m < 0)
        break;
    beg = arr[m].beg;
    end = arr[m--].end;
    if (beg >= end)
        continue;
    // partition
    p = A[beg];
    i = beg;
    j = end + 1;
}

for (;;)
{
    while (++i <= end && A[i] <
p); // ! while( A[++i] < p ); error
    while (A[--j] > p);
    if (i > j)
        break;
    swap(&A[i], &A[j]);
}
swap(&A[j], &A[beg]);
arr[++m].beg = beg;
arr[m].end = j - 1;
arr[++m].beg = j + 1;
arr[m].end = end;
}
free(arr);
}

```

(3) 归并排序

特点：稳定，时间复杂度 $O(n \log n)$

原理：归，递归之意，并，合并之意，归：利用递归思路，分隔数组为 2 部分数组，不断递归，最终实际上得到只含有 1 个成员的数组，完成排序；并：假如两个数组均排好序，采用两指针对比大小，将两数组合并。

代码实现：

```

void merge_sort(int *list, int length)
{
    int i, left_min, left_max, right_min,
right_max, next;
    //int tmp[BUF_SIZE];
    int *tmp = (int*)malloc(sizeof(int) *
length);

    if (tmp == NULL)
    {
        fputs("Error: out of memory\n",
stderr);
        abort();
    }

    for (i = 1; i < length; i *= 2) // i为步长,
1,2,4,8.....
    {
        for (left_min = 0; left_min < length
- i; left_min = right_max)
        {
            right_min = left_max =
left_min + i;
            right_max = left_max + i;

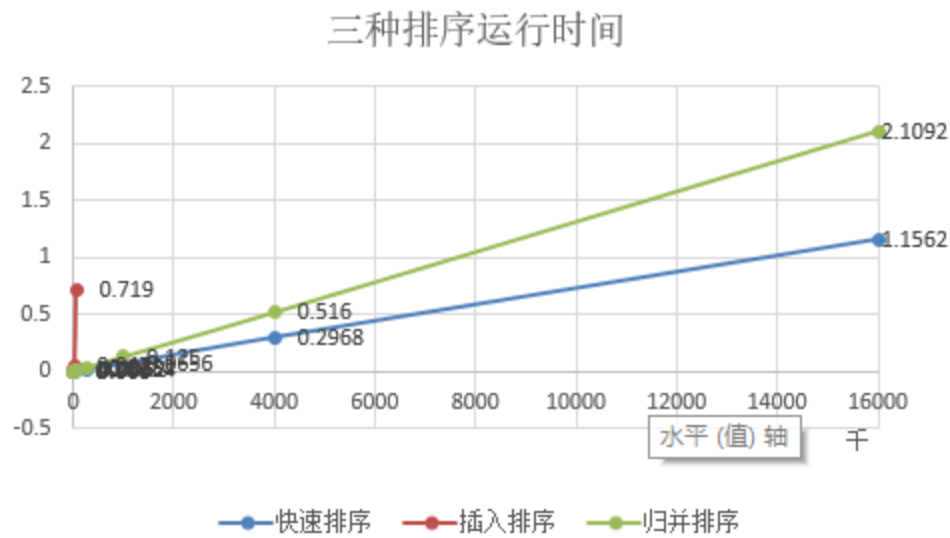
            if (right_max > length)
                right_max = length;

            next = 0;
            while (left_min < left_max &&
right_min < right_max)
                tmp[next++] =
list[left_min] > list[right_min] ?
list[right_min++] : list[left_min++];

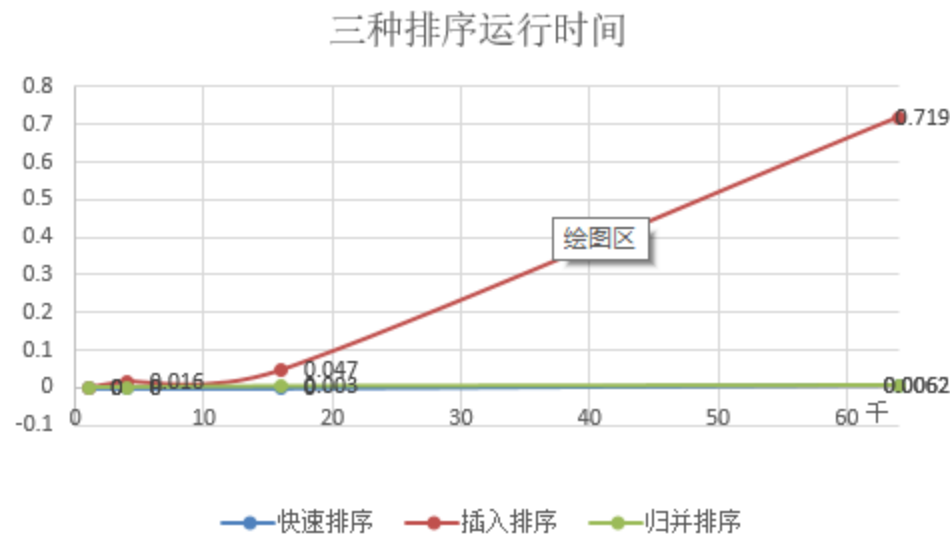
```

排序时间(s)		数据量							
		1	4k	16k	64k	256k	1M	4M	16M
快速排序	1	0	0	0	0.015	0.016	0.078	0.297	1.172
	2	0	0	0	0	0.015	0.063	0.281	1.156
	3	0	0	0	0	0.015	0.062	0.328	1.156
	4	0	0	0	0	0.016	0.063	0.281	1.156
	5	0	0	0	0.016	0.015	0.062	0.297	1.141
	av	0	0	0	0.006	0.015	0.065	0.296	1.156
	插入排序	1	0	0	0.04	0.735	—	—	—
2		0	0	0.04	0.719	—	—	—	—
3		0	0	0.04	0.703	—	—	—	—
4		0	0	0.04	0.719	—	—	—	—
5		0	0.01	0.04	0.719	—	—	—	—
av		0	0	0.04	0.719				
归并排序		1	0	0	0.01	0.016	0.031	0.125	0.516
	2	0	0	0	0	0.031	0.125	0.516	2.109
	3	0	0	0	0.015	0.016	0.125	0.516	2.109
	4	0	0	0	0	0.031	0.125	0.516	2.125
	5	0	0	0	0	0.032	0.125	0.516	2.094
	av	0	0	0.00	0.006	0.028	0.125	0.516	2.109

图 1 三种排序方法在不同数据量下的排序时间



放大图表中数据量为 1K，4K，16K，64K，256K 时的部分结果如下



结论：

- (1) 现象：插入排序所花费时间远多于归并排序和快速排序，并且随着数据量的增加迅速上升。

表 2 三种排序算法时间复杂度对比

排序法	平均时间	最差情形
快速排序	$O(n \log n)$	$O(n^2)$
插入排序	$O(n^2)$	$O(n^2)$
归并排序	$O(n \log n)$	$O(n \log n)$

原因：插入排序时间复杂度为 $O(n^2)$ ，归并排序与快速排序的时间复杂度均为 $O(n\log n)$ ，所以插入排序的运行时间远高于另外两种方法，数据量不大时，快速排序与归并排序的速度相近。

- (2) 现象：当数据量相对小时，归并排序与快速排序所用时间相近；而当数据量越来越大时，快速排序的优势越来越明显。

结论：虽然快速排序和归并排序平均时间复杂度相同，但是当数据量越来越大且数据随机时，尽管归并排序的比较次数较少，但是归并排序后期的合并操作所花费的时间便越来越大，合并操作对整体的效率影响越来越明显，包括后面大量数据的赋值操作等。所以当数据量变大时，不需要专门合并的快速排序的优势就变得越发明显，当数据失去随机性时，结果会有不同。

1.3 分工情况：

团队分工如下：

高俊龙：确立代码框架及思路，负责最终的文档整理、图表生成工作

姜波：用非递归方式完成三种排序方式的编写和调试

夏雅娴：考虑排序次数、时间度量等要素，整合完善代码，分析运行结果。

二、 Cache 配置参数测试

2.1 实验方案及结果

2.1.1 层级数与 Cache 容量

2.1.1.1 原理及代码实现

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");

```

```

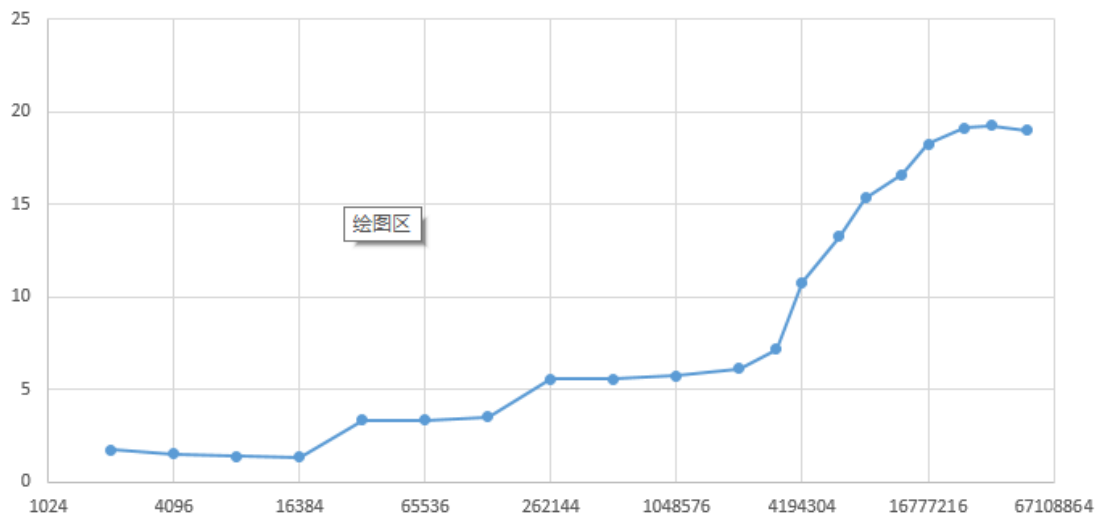
double times[20], sizes[20];
double sizeFactor, k,
notedPercentDrop, timeforFiftyLoops;
int threshold, nextCacheLikely = 1;
int i, j;
timeforFiftyLoops = loop(50, dataSize,
64);
loopFactor = (int)1000 /
timeforFiftyLoops;
sizeFactor = 3.0 / 4; k = 8.0 / 9;
i = 0;
numBytesToMove = dataSize;
while (numBytesToMove >= 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: %8d\n", (int)numBytesToMove,
timeTaken);
    fprintf(fp, "%8d\t%8d\n",
(int)numBytesToMove, timeTaken);
}
fclose(fp);
return 0;

```

2.1.1.2 运行结果及分析

```
numBytesToMove:50331648, time: 19.007000s
numBytesToMove:33554432, time: 19.264000s
numBytesToMove:25165824, time: 19.141000s
numBytesToMove:16777216, time: 18.251000s
numBytesToMove:12582912, time: 16.596000s
numBytesToMove: 8388608, time: 15.372000s
numBytesToMove: 6291456, time: 13.281000s
numBytesToMove: 4194304, time: 10.800000s
numBytesToMove: 3145728, time: 7.184000s
numBytesToMove: 2097152, time: 6.140000s
numBytesToMove: 1048576, time: 5.731000s
numBytesToMove:  524288, time: 5.539000s
numBytesToMove:  262144, time: 5.540000s
numBytesToMove:  131072, time: 3.502000s
numBytesToMove:   65536, time: 3.354000s
numBytesToMove:   32768, time: 3.329000s
numBytesToMove:   16384, time: 1.350000s
numBytesToMove:    8192, time: 1.395000s
numBytesToMove:   4096, time: 1.503000s
numBytesToMove:    2048, time: 1.726000s
请按任意键继续. . .
```

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。

2.1.2 块大小

2.1.2.1 原理及代码实现

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:%f\n", currentBlock, timeTaken);
        sizes[i++] = currentBlock;
        currentBlock *= 2;
        numOfLoop *= 2;
        blockSize = currentBlock;
    }
    return 0;
}
```

2.1.2.2 运行结果及分析

```
当步长为1时，耗时为:0.633000s
当步长为2时，耗时为:0.650000s
当步长为4时，耗时为:0.726000s
当步长为8时，耗时为:1.099000s
当步长为16时，耗时为:1.986000s
当步长为32时，耗时为:3.755000s
当步长为64时，耗时为:7.721000s
当步长为128时，耗时为:11.732000s
当步长为256时，耗时为:15.460000s
请按任意键继续. . . |
```

通过上面的分析我们知道，本机 Cache 的块大小为 64Byte。

2.1.3 组相连度

2.1.3.1 原理及代码实现

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>                                     //32j L1 cache, 16 int/blocj
#include <time.h>                                       end = clock();
#include <stdlib.h>                                     int total_time = end - start;
#define Lenght (1<<8<<10<<10)                       start = clock();    //减去移位运算
int loop = (1 << 27);    //循环次数，为了更          时间，以得到更准确的结果
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;
        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;
}

```

2.1.3.2 运行结果及分析

```
2路组相连测试总耗时 89.000000ms
4路组相连测试总耗时 84.250000ms
6路组相连测试总耗时 75.333333ms
8路组相连测试总耗时 70.625000ms
10路组相连测试总耗时 398.100000ms
12路组相连测试总耗时 648.500000ms
14路组相连测试总耗时 516.571429ms
16路组相连测试总耗时 603.875000ms
请按任意键继续. . .
```

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。

2.1.4 命中时间

2.1.4.1 原理及代码实现

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
linesizeof(int) = 16
    }
    ticks_2 =

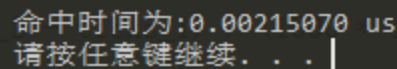
```

```

(double)read_time_stamp_counter();
    printf("命中时间为:%.8f us\n",
(double)(ticks_2 - ticks_1) / perus /
SIZE_1MB);
    return 0

```

2.1.4.2 运行结果及分析



```

命中时间为:0.00215070 us
请按任意键继续. . . |

```

由此可得命中时间为 2.15ns。

2.1.5 缺失代价

2.1.5.1 原理及代码实现

缺失代价即指需要访问的数据不再 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();

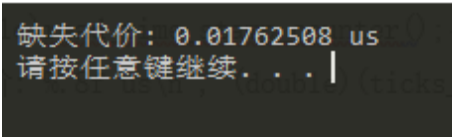
```

```

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;
}

```

2.1.5.2 运行结果及分析



由此可得缺失时间为 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

2.1.6 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。

2.2 实验分工情况

经团队讨论之后，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，具体分工如下：

夏雅娴：查阅资料提供代码编写思路，分析程序运行结果及结论并整理文档。

姜波：主要 cache 容量、级数及块大小部分负责代码的编写，并分析结果。

高俊龙：主要 cache 组相连度、命中时间及确实代价部分负责代码的编写，并对结果进行分析。

三、 矩阵运算优化

3.1 实验方案

矩阵乘法 $dest = src1 * src2$ ，矩阵的尺寸都是 $N \times N$ 。

3.1.1 M0 及对应代码实现

方法：基本的矩阵乘法，即 dest 的每个元素是 src1 的每一行元素和 src2 的每一列元素的乘积之和。

评价：

1. 当 N 很小时，矩阵大部分内容可以存在 cache 里，cache 缺失率较低；当 N 很大时，由于 cache 空间极其有限，无法将矩阵的每一行、每一列存在 cache 里，因此容易导致 cache 缺失。
2. 由于在内存中，矩阵是按行存放数据的，同一行的数据存放在相邻位置，逐列访问将会每次跳 $N \times 4\text{Bytes}$ ，当 N 很大时，cache 会存储矩阵的同一行数据，当 N 很小时，cache 才会存放矩阵很多行数据。当 src2 在做乘法时，是要访问同一列每一行的数据，因此当 N 很大时，容易造成 cache 缺失，并且频繁进行 cache 替换。

```
void M0(int *dest, int *src1, int *src2, int N,
int *temp)
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                dest[i * N + j] += src1[i *
N + k] * src2[k * N + j];
```

3.1.2 M1 及对应代码实现

方法：将 src2 先转置，然后在计算 dest 某一元素时 src1 仍然按行读取元素，src2 变为按行读取元素，更好地利用了 cache 的空间局部性原理，降低了 cache 缺失率。本方法无法解决 M0 中的第 1 个缺点。

```
void M1(int *dest, int *src1, int *src2, int N,
int *temp)
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                dest[i * N + j] += src1[i *
N + k] * temp[j * N + k];
    i];
}
```

3.1.3 M2 及对应代码实现

方法：将 src2 矩阵先转置，将 src1、src2 同一行的数据分块。这个改进看似毫无意义，实则指定了从内存按块取数据到 cache 的方式，降低了 cache 缺失率。但是当块过小或者过大时，性能都会下降。经多次实验。Block 取 16。

```
void M2(int *dest, int *src1, int *src2, int N,
int *temp)
{
    for (int i = 0; i < N; i++)
```

```

        for (int j = 0; j < N; j++)
            temp[i * N + j] = src2[j * N +
i];

        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                for (int kk = 0; kk < N; kk +=
Block)
                {
                    int r = 0;
                    int end_k = (kk +
Block)>N ? N : (kk + Block);
                    for (int k = kk; k < end_k;
k++)
                    {
                        r += src1[i * N + k] *
temp[j * N + k];
                    }
                    dest[i * N + j] += r;
                }
            }
        }
    }
}

```

3.1.4 M3 及对应代码实现

方法：将 src2 矩阵先转置，先将 src1、src2 同一行的数据分块，再将 src2 转置后的数据按行分块。在 M2 的基础上，降低 src2 的 cache 缺失率。

```

void M3(int *dest, int *src1, int *src2, int N,
        j++)
int *temp)
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            temp[i * N + j] = src2[j * N +
i];

    for (int i = 0; i < N; i++)
    {
        for (int jj = 0; jj < N; jj += Block)
        {
            for (int kk = 0; kk < N; kk +=
Block)
            {
                int end_j = (jj +
Block)>N ? N : (jj + Block);
                for (int j = jj; j < end_j;
j++)
                {
                    int r = 0;
                    int end_k = (kk +
Block)>N ? N : (kk + Block);
                    for (int k = kk; k <
end_k; k++)
                    {
                        r += src1[i * N
+ k] * temp[j * N + k];
                    }
                    dest[i * N + j] += r;
                }
            }
        }
    }
}

```

3.1.5 M4 及对应代码实现

方法：将 src2 矩阵先转置，先将 src1、src2 同一行的数据分块，再将 src2 转置后的数据按行分块，最后将 src1 的数据按行分块。在 M2 的基础上，降低 src1、src2 的 cache 缺失率。

```

void M4(int *dest, int *src1, int *src2, int N,
        {
        int *temp)
        for (int i = 0; i < N; i++)

```



```

        for (int j = 0; j < N; j++)
        {
            temp[i * N + j] = src2[j * N +
            i];
            for (int ii = 0; ii < N; ii += Block)
            {
                for (int jj = 0; jj < N; jj += Block)
                {
                    for (int kk = 0; kk < N; kk +=
                    Block)
                    {
                        int end_i = (ii +
                        Block)>N ? N : (ii + Block);
                        for (int i = ii; i<end_i;
                        i++)
                        {
                            int end_j = (jj +
                            Block)>N ? N : (jj + Block);
                            for (int j = jj; j <
                            end_j; j++)
                                {
                                    int r = 0;
                                    int end_k = (kk
                                    + Block)>N ? N : (kk + Block);
                                    for (int k = kk; k
                                    < end_k; k++)
                                    {
                                        r += src1[i
                                        * N + k] * temp[j * N + k];
                                    }
                                    dest[i * N + j]
                                    += r;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

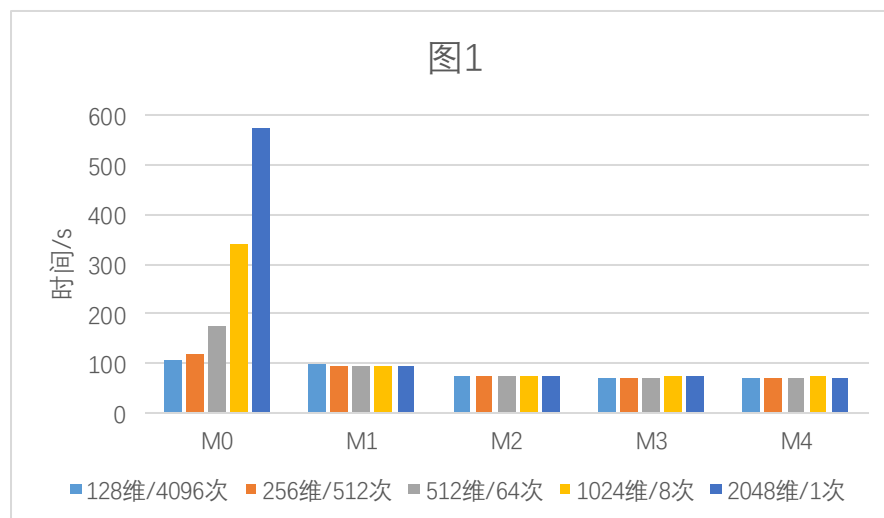
```

3.2 实验结果及分析

一共进行 5 次相同环境下的计算，并取平均值，减少误差。N 维/n 次，表示为矩阵为 N x N 维，计算次数为 n 次。纵坐标为运行的时间，单位为秒(s)。

分析一下每种维度的乘法和加法的运算次数。128 维的矩阵需要计算 4096 次，则乘法运算次数和加法运算次数都为 4096x128x128x128，256 维的矩阵需要计算 512 次，则乘法和加法运算次数都为 512x256x256x256，即 128 维和 256 维的乘法和加法的次数一样，同理，五种维度的矩阵乘法的乘法和加法的次数一样。

(1) 同一种方法在不同维度下的内部比较（同一方法的比较）



结论:由图 1 可知, M0 的计算时间随维度增大, 增加的幅度越大, 呈指数形式增长, 而其余四种方法则变化情况幅度比较微弱, 故 M0 受矩阵维度影响程度相较于其余四种方法更大。

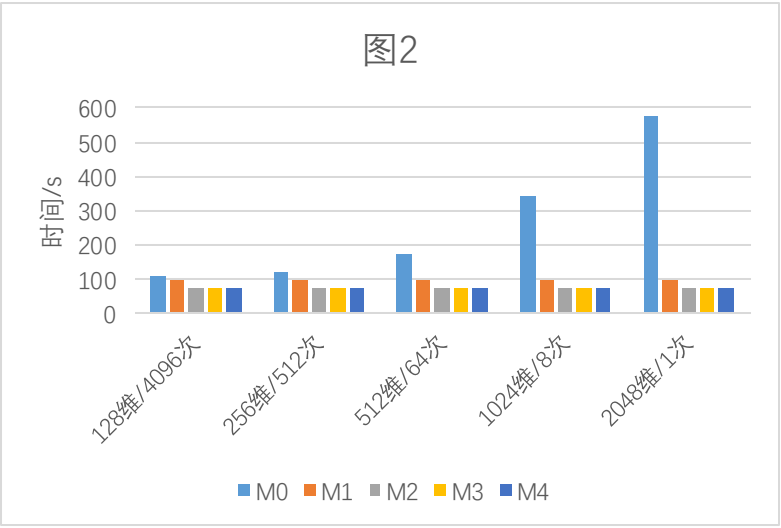
原因:

- a) c 语言中矩阵在内存中是以行优先存储的, 即先把数组中第一行的数组元素存完紧接着存下一行的, 直到存完。
- b) 因为 cache 的空间极其有限, 当 cache 缺失时由于 cache 会利用空间局部性把相同行的周围的元素一起取到 cache 中。
- c) M0 在每次最内层循环中取 src1 的同一行的元素、src2 的同一列的元素, 其余四种方法取 src1、src2 同一行的数。

分析:如果矩阵维度小, src2 数组的相邻行数据可以放进 cache 中, 则 M0 的 src2 缺失率较低; 当矩阵维度变大, src2 数组相邻行数据无法放进 cache 中, 又因为矩阵的维度大小呈指数形式增长, 则 M0 的 src2 缺失率就会急剧增加, 且呈指数形式增加。

其余四种方法则在五种维度下运算时间非常接近, 总体趋势是随维度增加运算时间微弱减少。当维度增加, 但矩阵乘法的外部循环次数减少, 而且由于行越来越大, 同一行的数据容易一起放在 cache 中, 从而减少 cache 的 src1 和 src2 的缺失率。

(2) 同一维度下的不同方法的外部比较 (方法之间的比较)



结论:在五组同一维度下, 从 M0 到 M4 的时间全都是降低的趋势。其中随维度的增长而 M0 与 M1 的时间差呈指数型增长, M1、M2、M3、M4 两两之间的时间

差几乎不变，且 M1 与 M2、M3、M4 的时间差较大，M2、M3、M4 时间差比较微弱。

原因：

- a) 由第 1 个结论可知，由于 M0 受矩阵维度影响，时间呈指数型增长，其余四种方法受矩阵维度较小。
- b) 本实验呈递进式，M1-M4，都对 src2 进行转置；M2-M4，都对转置后的 src2 的每一行分块；M3-M4，都对转置后的 src2 的每一列也分块；M4 在前面的所有基础上，再对 src1 的每一行进行分块。

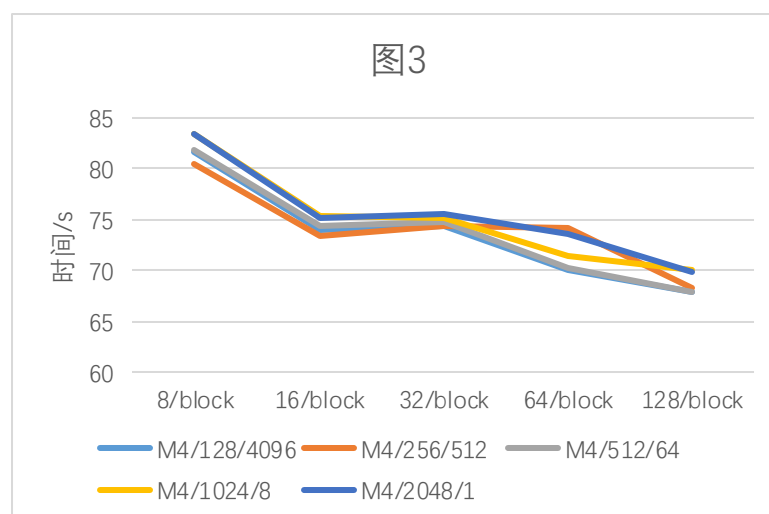
分析：从 M1 开始进行转置操作，使得 M1-M4 的 cache 的缺失率不再像 M0 的缺失率随维度增大而指数型增大，而是保持相对稳定的状态。

M2-M4，都对 src2 的每一行分块，在一个块的内部循环中块的元素可以放在 cache 中，块元素的缺失率就比较低；而 M1 由于没有分块，所以在行的内部循环中由于 cache 无法存放整整一行的元素，行元素容易发生 cache 缺失。

M3-M4，都对转置后的 src2 的每一列也分块，以及 M4 对 src1 的每一行进行分块，实际上也如同上面所讲，在列内部循环会降低缺失率，所以有降低。但是降低的能力极其有限，因为块大小为 16x16 时，cache 可能无法存储完整的矩阵块，所以也会发生一定的缺失，只是缺失率有微弱的降低，故 M2-M4 的时间差只有微弱的减少趋势。

(3) 探究不同块大小的性能比较

由于 M4 方法在前面 M0-M4 中用时最短，因此只用 M4 来探究不同分块的大小在不同维度下的使用时间，作为 M4 的递进方案。



结论：

在块为 128 时，用时最短。M2-M4 的块均为 16，当块为 128 时，减少了循环的次数，同时又利用了块的空间局部性，因此性能最好。

但是，最意外的是，当块为 128 的时候，M1 与 M4 在进行 128 维的矩阵运算看似等价，然而前者用时 100s 左右，后者用时 70s 左右，相差 30%左右，多次实验结果不变。推测原因可能是，后者的由外循环进入内层循环时，大部分矩阵元素由内存取至 cache，从而降低内循环的 cache 缺失率，而前者没有这个外部循环来提示取数据到内存中，因此循环内部 cache 缺失率高。

3.3 实验分工情况

经团队讨论之后，想出包括基本方法在内的 5 种矩阵相乘方案，然后分工如下：

夏雅娴：写基本的代码框架，并做了不同块大小的探究实验及其分析工作，并整理文档工作。

高俊龙：实现了 M0、M1、M2 的代码，并做了第一个分析。

姜波：实现了 M3、M4 的代码，并做了第二个分析。