2015/04/05

计22

滕爽

2012011270

LAB1 Report

计算机系统结构

目录

[一、测量cache的大小 2](#_Toc416299408)

[1、[实验原理] 2](#_Toc416299409)

[2、 [实验遇到问题] 2](#_Toc416299410)

[3、[实验设计思路] 2](#_Toc416299411)

[4、[实验数据分析] 3](#_Toc416299412)

[5、[实验结果展示] 3](#_Toc416299413)

[二、测量Block大小 4](#_Toc416299414)

[1、[实验原理] 4](#_Toc416299415)

[2、[实验设计思路] 5](#_Toc416299416)

[3、[实验数据分析] 5](#_Toc416299417)

[4、[实验结果展示] 6](#_Toc416299418)

[三、测量cache相连度 7](#_Toc416299419)

[1、[实验原理] 7](#_Toc416299420)

[2、[实验设计思路] 7](#_Toc416299421)

[3、[实验数据分析] 7](#_Toc416299422)

[4、[实验遇到问题] 8](#_Toc416299423)

[5、[实验结果展示] 8](#_Toc416299424)

[四、对所给程序matrix\_mul.cpp进行优化 9](#_Toc416299425)

[1、[实验原理] 9](#_Toc416299426)

[2、[实验结果展示] 10](#_Toc416299427)

[五、测量数据缓存的写策略 11](#_Toc416299428)

[1、[实验原理] 11](#_Toc416299429)

[2、[实验数据分析] 11](#_Toc416299430)

[3、[实验结果展示] 12](#_Toc416299431)

[六、测量数据缓存的替换策略 12](#_Toc416299432)

[1、[实验原理] 12](#_Toc416299433)

[2、[实验设计思路] 12](#_Toc416299434)

[3、[实验数据分析] 13](#_Toc416299435)

[4、[实验结果展示] 13](#_Toc416299436)

[七、实验代码说明 14](#_Toc416299437)

[1、文件说明 14](#_Toc416299438)

[2、作图原始数据 15](#_Toc416299439)

[3、代码运行说明 18](#_Toc416299440)

## 一、测量cache的大小

### 1、[实验原理]

测量cache的基本思想是对一段大小的数组反复读取，并且逐次增加数组的大小。当数组的大小超过缓存的大小时，频繁读取就会导致频繁替换缓存，使得吞吐量下降。所以只要测量吞吐量，观察发生突变的点，即可得到缓存的大小。

### 2、 [实验遇到问题]

最新的Intel处理器自带硬件预读取功能，即会根据程序执行时读取内存的步长预测下一次访问，并且提前读取到缓存里。如果按照顺序访问数组的方法，则会发现吞吐量一直不会下降，或没有很明显的突变点，正是因为硬件预读取提前替换了缓存，没有影响到读取的效率。

### 3、[实验设计思路]

为了不让处理器预测出访问的步长，可以每次产生一个伪随机数作为下标访问。但是产生随机数本身就会影响程序计时，并且调用外部函数时会导致内存访问，使得之前的缓存失效。

仿照链表的实现方式，把下一次访问的地址放在这次访问的地址所对应的变量中。每次读取内存的时候，把读到的值作为下一次访问的地址，防止硬件预读取工作。并且测量缓存大小时要适当增大步长，加大缓存替换的频率，使得结果更加明显。

### 4、[实验数据分析]

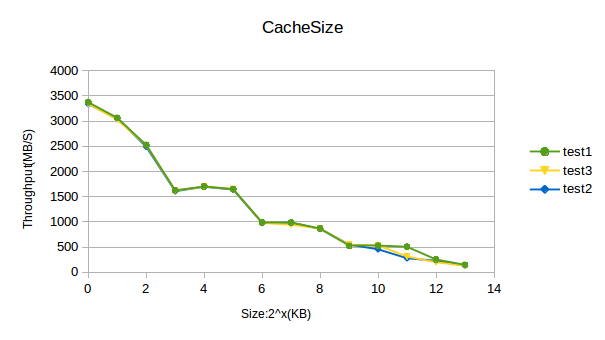
运行程序，对1KB到2048KB之间大小的数组进行测试，并且得出吞吐量。吞吐量的单位是MB/s，但是因为使用clock()函数计时，结果并不是很准确。不过最重要的是吞吐量的相对大小，所以并不影响测量缓存的大小。

程序源代码为**cacheSize.cpp**，程序会给出相应的吞吐量。

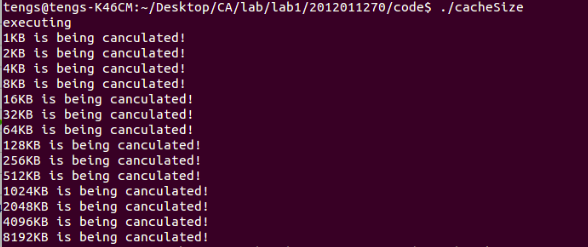
程序输出如图fig2所示，将结果画成折线图如图fig1所示。注意到横轴对应的是2^xKB。

从图中可以看出在32KB和256KB处有明显的吞吐量的突降，于是判断L1、L2、L3数据缓存的大小分别为32KB、256KB、2048KB。

### 5、[实验结果展示]



**Figure 1**



**Figure 2**

## 二、测量Block大小

### 1、[实验原理]

对内存进行顺序访问，但是只要读到块里的某一个字节，整个块都会被缓存进来。所以如果按顺序每字节均访问，那么仅仅会在访问该块的第一个字节的时候访问更低级存储。接下来在该块内的访问会直接命中。如果不是每个字节都依次访问，加大访问的步长，可以预见当步长等于块大小的时候，每一次读取就会要访问更低级存储，将整个块都加载进来，这样的吞吐量是最低的。

采取每次加大步长的方法，观察吞吐量将会出现先降后升的现象，并且最低点对应的步长即正好是块大小。

### 2、[实验设计思路]

注意到硬件预读取带来的影响，同样使用与第一题类似链表的数据结构进行访问。

### 3、[实验数据分析]

程序源代码为**blockSize.cpp**运行程序，对步长从1到64进行测试。这里的步长是指uint64的长度，即8B。

得到程序输出如图fig3，将结果画成折线图如图fig4。程序前面在波动中下降，并当步长等于8，即64B的时候达到吞吐量的最低值。

所以可以判断L1和L2数据缓存的块大小是64B。

### 4、[实验结果展示]

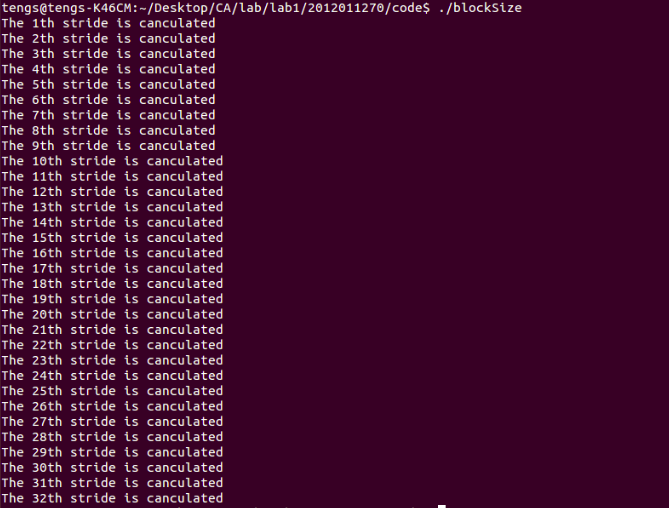
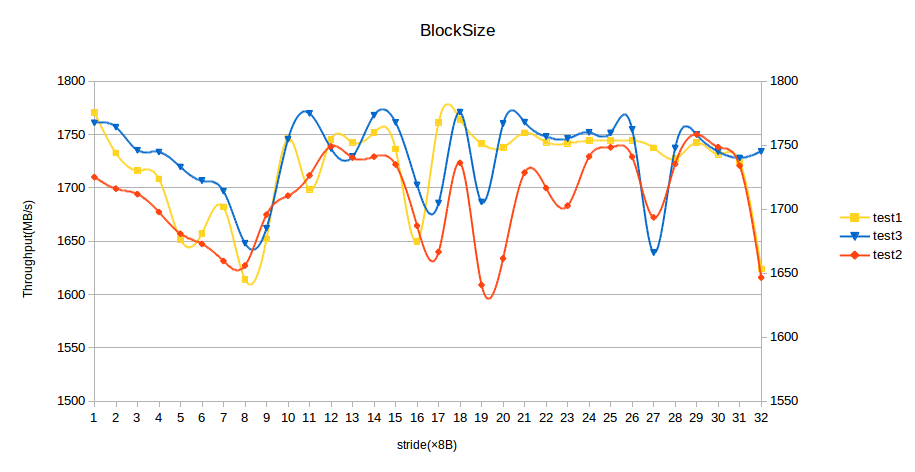


Figure 3



**Figure 4**

## 三、测量cache相连度

### 1、[实验原理]

已知块大小是64B，第1级缓存为32KB，2级缓存为256KB。下面只要测出来一共有多少个组，就能知道每个组的大小和相连度了。

### 2、[实验设计思路]

已知块大小是64B，占了地址最低的6位。这里可以取一掩码，用来分割地址前面的标签和后面的索引和偏移量部分。实际上在程序实现时可以取这个掩码加1之后的值，每次往地址上累加。如果掩码没有盖住索引和偏移量部分，那么往地址上加的时候会改变索引，会从一个组跳到另一个组。

如果掩码正好盖住或者超过了索引和偏移量，那么往地址上累加的时候就只会改变标签，而仍然还在同一个组内。

通过枚举掩码长度，当掩码正好盖住索引和偏移量的时候，所有读取的内容都属于同一个缓存组，缓存冲突频率最大，吞吐量最低。通过观察

吞吐量下降到极值的这个点，即可算出相连度。

### 3、[实验数据分析]

程序源代码为**associativity.cpp**依次左移掩码，得到程序输出如图fig5。这里的mask实际上是掩码加1之后的值，即往地址上累加的值。

数据如图fig6可以看出当掩码为12位的时候吞吐量最低，即可判断地址的低12位是索引和偏移量。已知偏移量为6位，所以索引为6位，共有64个不同的组。一级缓存共有32KB，而块大小之前已经算出来是64B，计算得到一组里有8个块，即是8相连的。

考虑到一级缓存和二级缓存有一定的一致性，猜测二级缓存也是8相连的。之前已经得到二级缓存共256KB，算得一共应有512组。对应的掩码有15位，可以看到在图fig6中当掩码为15位时，吞吐率的增长趋势得到抑制，验证猜测是正确的，即二级缓存也是8相连的。

### 4、[实验遇到问题]

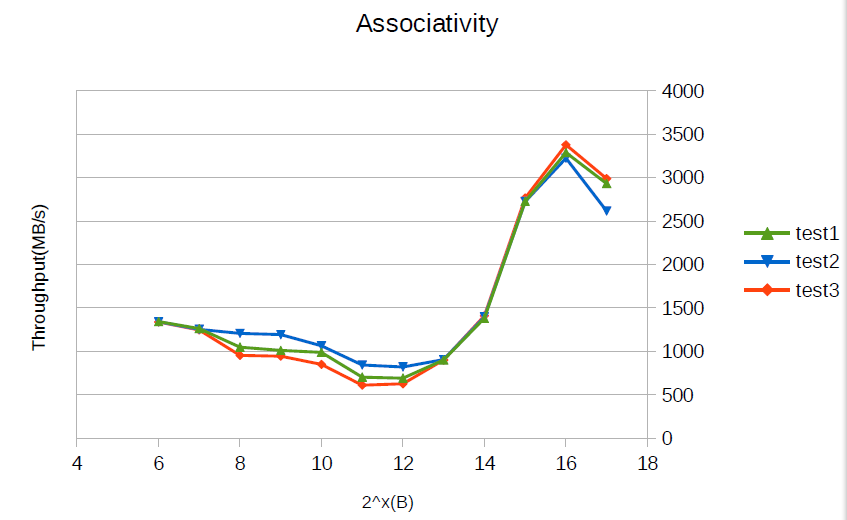
图 fig6后面吞吐量开始上升，猜测是因为程序采用固定读取量，测量时间得到吞吐量的方法。当掩码变大时，实际上每一步跳跃距离变大，但数组大小不变，

为了达到同样的读取量，只能多次重复读取。跳跃距离变大导致不同的地址的数量太少，可能会无法使缓存某一组填满，更不用说替换了，所以吞吐量会变高。要解决这个问题可以扩大数组的大小。

### 5、[实验结果展示]



**Figure 5**



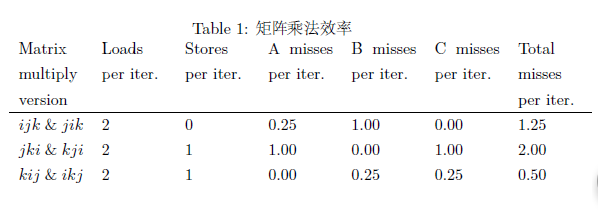
**Figure 6**

## 四、对所给程序matrix\_mul.cpp进行优化

### 1、[实验原理]

对于A×B=C的矩阵乘法。对于指数的顺序，一共有三种方式。其中ijk和jik等效， jki和kji等效， kij和ikj等效。

根据Computer Systems A Programmer's Perspective上的结论，有如表tab1所示结论。



**Figure 7**

所以只要采用kij或者ikj方式，就能大幅利用空间局部性提高效率。

为了进一步提高程序的运行效率，可以采用矩阵分块的思想，增大空间局部性。

A为1000×1000的矩阵，每个元素占4B，一行为4000B。

考虑到一级缓存为32KB，分到A， B， C三个矩阵上，每个矩阵大概可以利用10KB。所以在ikj的方式上进一步改进，每次对i， i+1， i+2，i+3同时操作，降低B的重复读取的次数。

这时的程序源代码为**matrix\_mul.cpp**，未分块时时间减少55%，分块后程序输出如图fig9所示，时间减少88%。

如果还想进一步优化，可以使用编译器的优化参数-O3，得到程序输出如图fig9所示，时间减少达到96%，性能提升巨大。可能因为循环次数是常数，编译器直接进行循环展开，再加上其它一些优化，才能得到这样的结果。

### 2、[实验结果展示]



**Figure 8**



**Figure 9**

## 五、测量数据缓存的写策略

### 1、[实验原理]

数组大小小于缓存大小时，如果是采用的写回法，则没有对低级存储的访问，而如果是采用的写直达法，则每次都会访问低级存储。但是当数组大小大于缓存大小时，无论采用哪种方法，都会对低级存储进行写操作。于是可以通过比较这两种情况下的吞吐量的差别，来测量数据缓存的写策略。

为了增大可能存在的性能上的差别，每次访问的步长正好是一个块大小。这样就能在最短时间内写脏整个数据缓存，扩大吞吐量的差距。

### 2、[实验数据分析]

程序源代码为**writeBack.cpp**。运行得到程序输出如图fig10当数组大小正好为一级缓存大小时，吞吐量有1362.03MB/s，而当数组大小超过一级缓存大小时，吞吐量下降到1297.74MB/s，下降了4.7%。多次测量均可观测到这个差距，说明数据缓存采用写回法。

### 3、[实验结果展示]

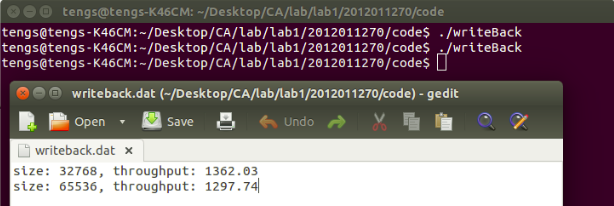


Figure 10

## 六、测量数据缓存的替换策略

### 1、[实验原理]

通过巧妙地构造访问序列，来测试缓存的替换策略。这里构造两个序列，其中一个对FIFO有特别优化，另一个对LRU有特别优化，但是两者的总访问量保持相当。这样如果一个序列访问时间短，另一个访问时间长，就可以以此来判断数据缓存的替换策略。

### 2、[实验设计思路]

一个缓存组里面有8个块，所以一共需要9个块，编号为0到8。

注意到它们的地址的后12位是相同的，即都属于同一个组。

构造出对FIFO有特别优化的序列是081102213324435546657768870，对LRU有特别优化的序列是080212434656878101323545767。

可以看出FIFO序列中有n，n-1，n+1类型元素，如果采用LRU会打乱n和n+1的顺序，使得缓存替换增多。同样LRU序列中有n，n-1，n类型元素，如果采用FIFO则会使得第二次访问n的时候发生缓存替换。

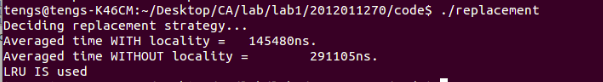
最后需要注意在执行完一个序列之后缓存要恢复成原样，这里设为01234567。只有这样才能反复测量，得到比较明显的结果。

### 3、[实验数据分析]

程序源代码为 **replacement.cpp**。

运行程序输出如图fig11看出运行LRU序列的时间比FIFO的少50%，得到缓存采用的替换策略是LRU。

### 4、[实验结果展示]



**Figure 11**

## 七、实验代码说明

### 1、文件说明

如下表所示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **文件名** | **说明** | **生成中间文件** | **可执行文件** | **数据文件** |
| ele.cpp | 被以下文件调用的基类 | ele.o |  |  |
| ele.h |  |  |  |  |
| cacheSize.cpp | 测量cache的大小 | cacheSize.o | cacheSize | cacheSize.dat |
| blockSize.cpp | 测量block的大小 | blockSize.o | blockSize | blockSize.dat |
| associativity.cpp | 测量相连度 | associativity.o | associativity | associativity.dat |
| writeback.cpp | 测量写策略 | writeback.o | writeback | writeback.dat |
| replacement.cpp | 测量替换策略 | replacement.o | replacement | replacement.dat |
| matrix\_mul.cpp | 根据存储器结构优化代码 | matrix\_mul.o | matrix\_mul | matrix\_mul.dat |
| makefile | 用于linux/g++运行的文件 |  |  |  |
| picture | 文件夹放置图片 |  |  |  |

### 2、作图原始数据

如下表所示

1. cacheSize

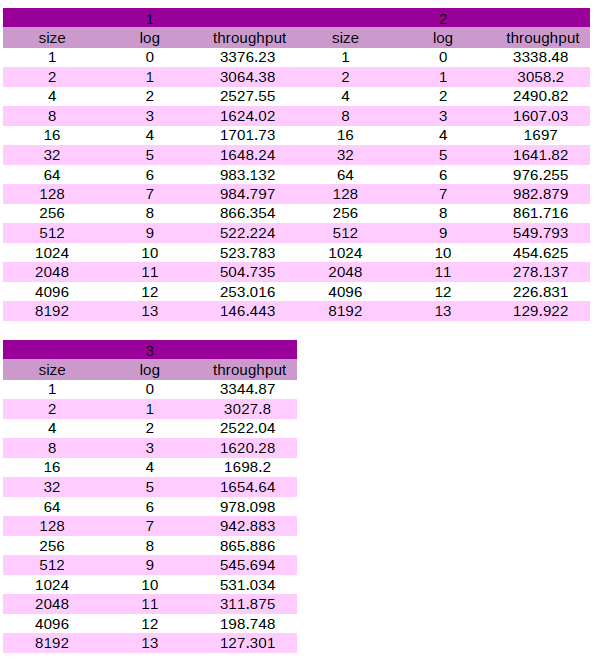


Figure 12

1. blockSize

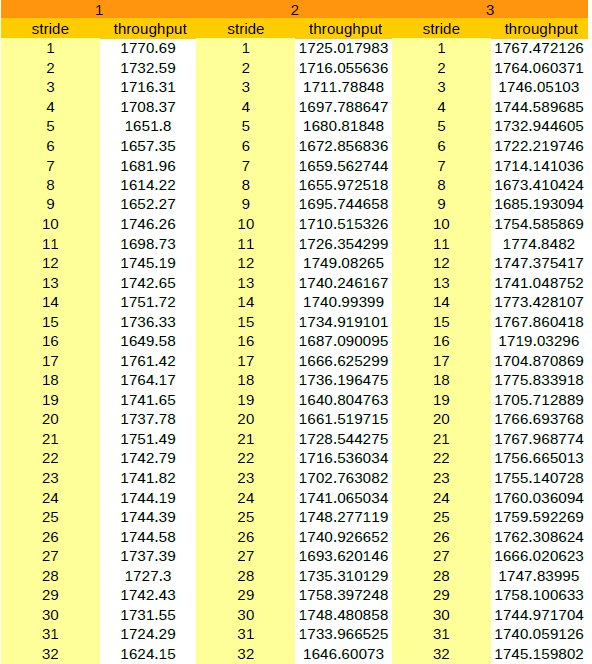


Figure 13

1. associativity

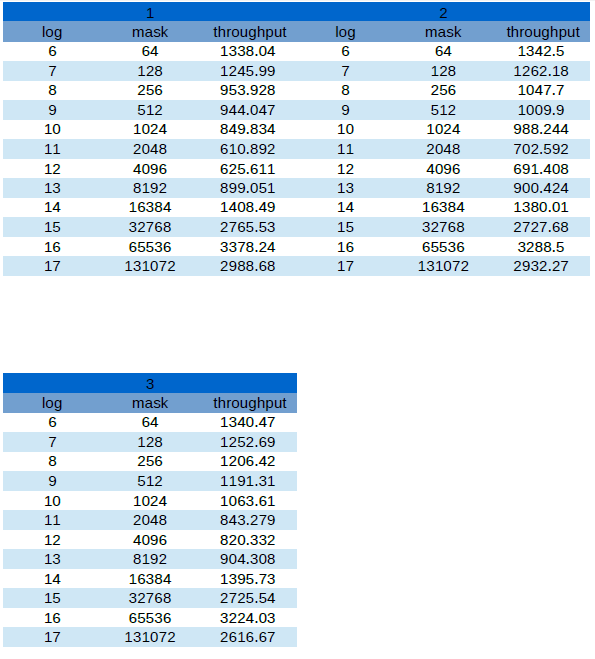


Figure 14

### 3、代码运行说明

运行环境为Ubuntu 14.10 + g++

shell指令：

make：进行编译运行所有cpp文件

make cacheSize

make blockSize

make asso

make matrix\_mul

make writeBack

make replacement

分别编译运行每个部分。