

# 计算机系统结构第一次实验

李雨田 2010012193 计 14

March 29, 2014

## Contents

<b>1</b>	<b>测量数据缓存的大小</b>	<b>2</b>
1.1	实验原理 . . . . .	2
1.2	实验结果和数据分析 . . . . .	2
<b>2</b>	<b>测量数据缓存的块大小</b>	<b>4</b>
2.1	实验原理 . . . . .	4
2.2	实验结果和数据分析 . . . . .	4
<b>3</b>	<b>测量数据缓存的相连度</b>	<b>7</b>
3.1	实验原理 . . . . .	7
3.2	实验结果和数据分析 . . . . .	7
<b>4</b>	<b>对所给程序 <code>matrix_mul.cpp</code> 进行优化</b>	<b>9</b>
<b>5</b>	<b>测量数据缓存的写策略</b>	<b>11</b>
5.1	实验原理 . . . . .	11
5.2	实验结果和数据分析 . . . . .	11
<b>6</b>	<b>测量数据缓存的替换策略</b>	<b>11</b>
6.1	实验原理 . . . . .	11
6.2	实验结果和数据分析 . . . . .	12

## 1 测量数据缓存的大小

### 1.1 实验原理

测量数据缓存大小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.

但是最新的 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.

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.

这里可以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.

### 1.2 实验结果和数据分析

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 `cache-size.c`, 运行时通过命令行提供两个参数, 分别为起始和结束测量的大小. 然后程序会给出相应大小下的吞吐量.

程序输出如图1所示, 将结果画成折线图如图2所示. 注意到横轴对应的是  $2^n$ KB. 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 和 L2 数据缓存的大小分别为 32KB 和 256KB.

```
size: 1KB, throughput: 4723.100180
size: 2KB, throughput: 4498.637010
size: 4KB, throughput: 3501.863357
size: 8KB, throughput: 2790.821903
size: 16KB, throughput: 2286.950924
size: 32KB, throughput: 2200.802842
size: 64KB, throughput: 1297.379616
size: 128KB, throughput: 1298.050199
size: 256KB, throughput: 1282.880066
size: 512KB, throughput: 707.177437
size: 1024KB, throughput: 713.854618
size: 2048KB, throughput: 709.695294
```

Figure 1: 数据缓存大小程序输出

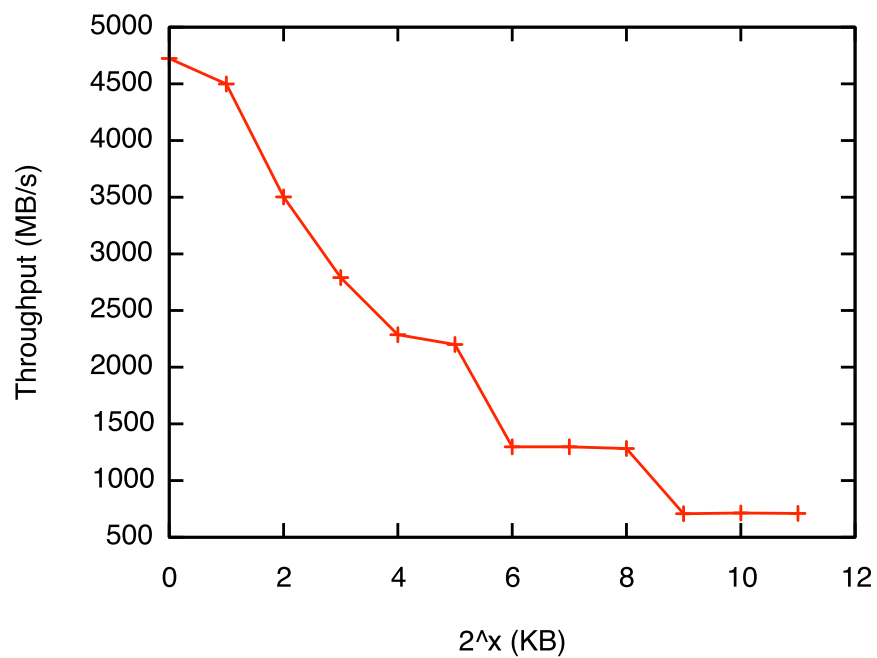


Figure 2: 数据缓存大小数据

## 2 测量数据缓存的块大小

### 2.1 实验原理

同样是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.

所以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.

这里仍然要注意到硬件预读取带来的影响, 同样使用类似链表的数据结构进行访问.

### 2.2 实验结果和数据分析

程序源代码为 `block-size.c`. 运行程序, 对步长从 1 到 32 进行测试. 这里的步长是指 `uint64_t` 的长度, 即 8B.

得到程序输出如图3, 将结果画成折线图如图4. 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.

```
stride: 1, throughput: 2416.590460
stride: 2, throughput: 2322.337760
stride: 3, throughput: 2347.899559
stride: 4, throughput: 2275.441783
stride: 5, throughput: 2314.489120
stride: 6, throughput: 2258.253569
stride: 7, throughput: 2272.907618
stride: 8, throughput: 2249.660711
stride: 9, throughput: 2264.554851
stride: 10, throughput: 2327.188102
stride: 11, throughput: 2320.436668
stride: 12, throughput: 2371.438066
stride: 13, throughput: 2337.786123
stride: 14, throughput: 2350.830453
stride: 15, throughput: 2304.621960
stride: 16, throughput: 2222.381377
stride: 17, throughput: 2351.009913
stride: 18, throughput: 2390.341061
stride: 19, throughput: 2246.196519
stride: 20, throughput: 2330.975800
stride: 21, throughput: 2352.668176
stride: 22, throughput: 2346.830089
stride: 23, throughput: 2334.714630
stride: 24, throughput: 2336.905825
stride: 25, throughput: 2363.474169
stride: 26, throughput: 2371.992879
stride: 27, throughput: 2253.409547
stride: 28, throughput: 2377.032389
stride: 29, throughput: 2375.976189
stride: 30, throughput: 2229.154467
stride: 31, throughput: 2350.108842
stride: 32, throughput: 2168.643825
```

Figure 3: 数据缓存块大小程序输出

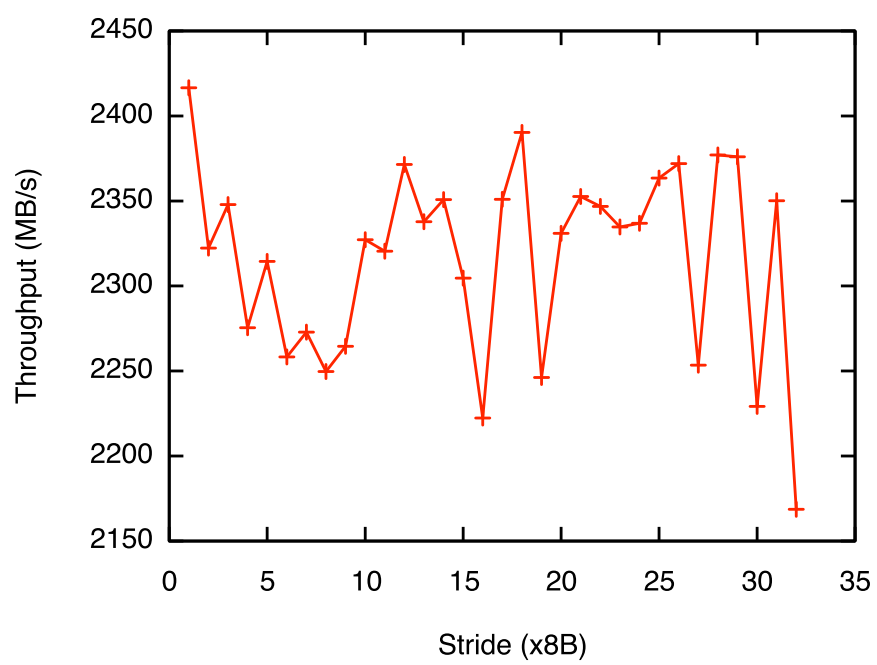


Figure 4: 数据缓存块大小数据

## 3 测量数据缓存的相连度

### 3.1 实验原理

到现在已经知道块大小是 64B, 缓存的大小也已经测出来了. 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.

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.

所以通过枚举掩码长度, 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.

### 3.2 实验结果和数据分析

程序源代码为 `associativity.c`. 依次左移掩码, 得到程序输出如图5. 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6. 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6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.

对于图6后面吞吐量开始上升, 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, 但是实际上目前已经可以得到缓存的相连度, 并没有这个必要.

```
mask: 64, throughput: 1792.765735
mask: 128, throughput: 1713.085742
mask: 256, throughput: 1735.105773
mask: 512, throughput: 1675.176147
mask: 1024, throughput: 1563.110590
mask: 2048, throughput: 1187.185469
mask: 4096, throughput: 1083.229929
mask: 8192, throughput: 1490.857035
mask: 16384, throughput: 2022.253677
mask: 32768, throughput: 4039.332208
mask: 65536, throughput: 5217.116626
mask: 131072, throughput: 3767.756796
```

Figure 5: 数据缓存相连度程序输出

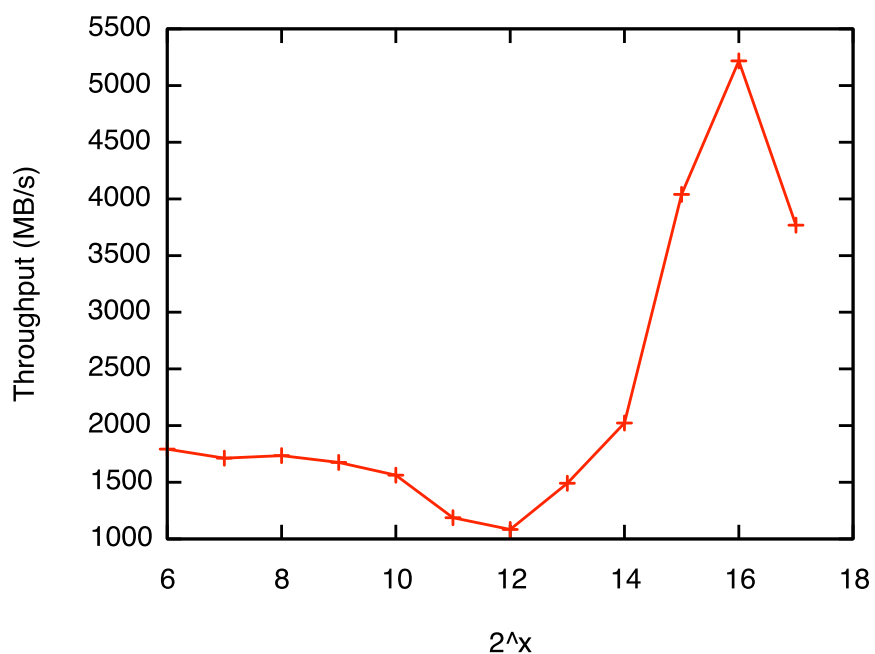


Figure 6: 数据缓存相连度数据



## 4 对所给程序 `matrix_mul.cpp` 进行优化

对于  $A \times B = C$  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1所示结论.

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.

改进后程序输出如图7所示. 经测试使用原来的方法耗时8 292 664 ms, 使用  $ikj$  方式只需2 781 312 ms, 时间减少 66.46%.

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  $1000 \times 1000$  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`, 程序输出如图8所示, 时间减少 71.77%.

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9所示, 时间减少达到 95.98%, 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.

Table 1: 矩阵乘法效率

Matrix	Loads	Stores	A misses	B misses	C misses	Total
multiply	per iter.	per iter.	per iter.	per iter.	per iter.	misses
version						per iter.
$ijk$ & $jik$	2	0	0.25	1.00	0.00	1.25
$jki$ & $kji$	2	1	1.00	0.00	1.00	2.00
$kij$ & $ikj$	2	1	0.00	0.25	0.25	0.50

```
time spent for original method : 8292664 ms
time spent for new method : 2781312 ms
```

Figure 7: 优化矩阵乘法程序输出一

```
time spent for original method : 8928669 ms  
time spent for new method : 2520279 ms
```

Figure 8: 优化矩阵乘法程序输出二

```
time spent for original method : 6769329 ms  
time spent for new method : 272354 ms
```

Figure 9: 优化矩阵乘法程序输出三

## 5 测量数据缓存的写策略

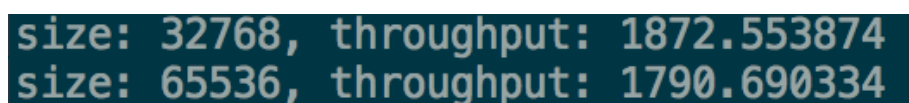
### 5.1 实验原理

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.

### 5.2 实验结果和数据分析

程序源代码为 `write-back.c`. 运行得到程序输出如图10. 当数组大小正好为一级缓存大小时, 吞吐量有 1872.55MB/s, 而当数组大小超过一级缓存大小时, 吞吐量下降到 1790.69MB/s, 下降了 4.4%. 多次测量均可观测到这个差距, 说明数据缓存采用写回法.



```
size: 32768, throughput: 1872.553874
size: 65536, throughput: 1790.690334
```

Figure 10: 数据缓存写策略程序输出

## 6 测量数据缓存的替换策略

### 6.1 实验原理

此时已经知道缓存的基本大小信息, 可以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.

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.

## 6.2 实验结果和数据分析

程序源代码为 `replacement.c`. 运行程序输出如图11. 看出运行 LRU 序列的时间比 FIFO 的少 11.55%, 得到缓存采用的替换策略是 LRU.

实际上通过搜索知道, 此 CPU 的缓存采用的是 LRU 的变种 PLRU. 并不是严格地按照最近访问先后顺序替换缓存, 但是大致上是和最近访问相关, 最近访问越久远的缓存更有可能被替换出去, 符合测量结果的判断.

A terminal window with a dark blue background and light blue text. The text displays two lines: 'FIFO: 3351' and 'LRU: 2964'. The numbers are in a monospaced font.

Figure 11: 数据缓存替换策略程序输出