

**计算机系统结构实验报告**

|  |
| --- |
| 姓 名： 宁嘉 |
| 学 院： 计算机科学与技术 |
| 专 业： 计算机科学与技术 |
| 班 级： ACM1601 |
| 学 号： U201614787 |
| 指导教师： 吴非、曹强 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2019 年. 4月. 28日

**目 录**

[1. 第一部分：Cache模拟器 3](#_Toc5302128)

[**1.1.** **实验目的** 3](#_Toc5302129)

[**1.2.** **实验环境** 3](#_Toc5302130)

[**1.3.** **实验思路** 3](#_Toc5302131)

[**1.4.** **实验结果和分析** 3](#_Toc5302132)

[2. 第二部分：矩阵转置优化 3](#_Toc5302133)

[**2.1.** **实验目的** 3](#_Toc5302134)

[**2.2.** **实验环境** 3](#_Toc5302135)

[**2.3.** **实验思路** 3](#_Toc5302136)

[**2.4.** **实验结果和分析** 3](#_Toc5302137)

[3. 总结和体会 4](#_Toc5302138)

[4. 对实验课程的改进建议 4](#_Toc5302139)

# 第一部分：Cache模拟器

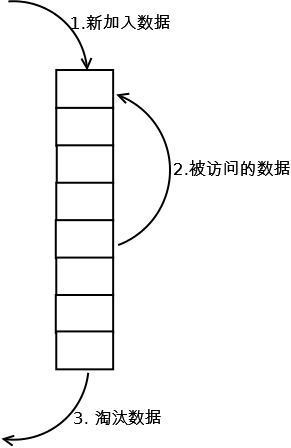
* 1. **实验目的**

1. 理解cache的工作原理
2. 用软件实现组相联cache
3. 用软件实现LRU替换算法
   1. **实验环境**

Linux 64-bit, gcc, python2

* 1. **实验思路**

由于提供的代码片段是采用计数器实现的LRU，这在硬件层面是合适的。但是从软件层面看，可能发生整数溢出、实现效率过低等问题，于是重构代码，采用（前向）链表的形式实现。



为了使得效率尽可能的高，这里采用的数据结构是在组间用hash的方式，在组内用链表的方式实现全相联LRU算法。

于是对于每个cache组类型cache\_set\_t我们可以定义如下结构：

|  |
| --- |
| typedef struct cache\_set {  cache\_line\_t \* line;  int size;  } cache\_set\_t; |

其中cache\_line\_t \* line为指向组内首元素的指针，size为组内已存放cache行的个数。

对于每个cache行类型，可以利用将其串接在一起构成链表，其中链表首的是最近被访问的元素。cache行结构定义如下：

|  |
| --- |
| typedef struct cache\_line {  mem\_addr\_t tag;  struct cache\_line \*next;  } cache\_line\_t; |

其中tag用来区分标识其映射的区域。

初始化cache的代码如下，在这里，直接动态分配了cache组数量大小的数组。并且初始化每个cache\_set\_t类元素。

|  |
| --- |
| void initCache()  {  int i;  cache = (cache\_set\_t\*) malloc(S \* sizeof(cache\_set\_t));  for (i=0; i<S; i++){  cache[i].line = NULL;  cache[i].size = 0;  }  /\* Computes set index mask \*/  set\_index\_mask = (mem\_addr\_t) (S - 1);  } |

其中初始化采用将组大小设置为0,并将指向组中行的指针设置为空。

为了防止内存泄漏，需要定义释放cache函数，该函数将分配的cache行释放，同时将分配的cache组释放。

|  |
| --- |
| void freeCache()  {  int i;  for(i = 0; i < S; ++i) {  if(cache[i].line) free(cache[i].line);  }  free(cache);  } |

Cache的访问操作可以分为两部分，首先根据地址计算出其所在的cache组，沿着相应的cache组链表进行查找，有两种情况——在查找链表途中找到或到达链表尾部仍未找到与地址对应的cache行；其次根据是否找到进行修改替换。

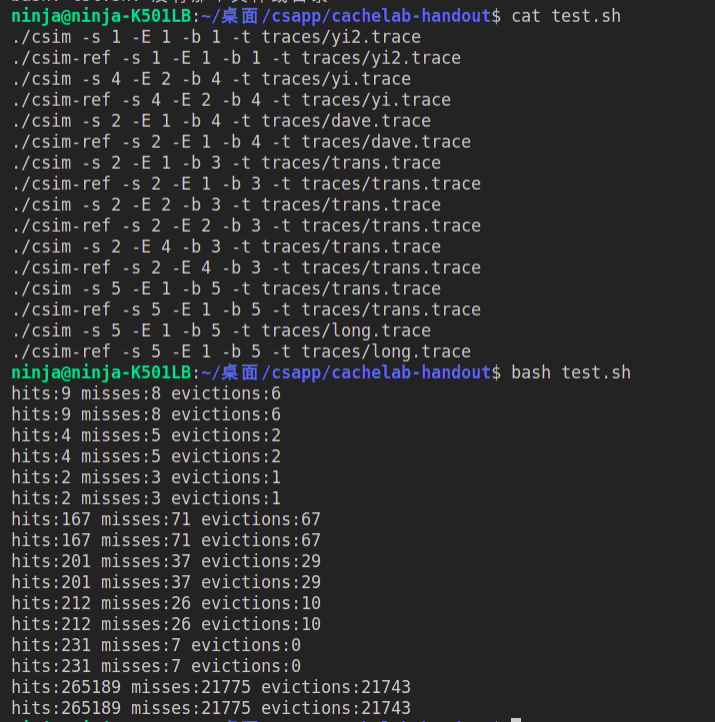
只需要注意的是这里不需要用valid位来进行存取，因为进入链表中的cache行都默认有效。在这个实验中，为了简化操作，不需要考虑写回操作。

|  |
| --- |
| void accessData(mem\_addr\_t addr)  {  mem\_addr\_t block\_addr = (addr >> (b+s));  mem\_addr\_t set\_index = (addr >> b) & (S-1);  int flag = 0;  for(cache\_line\_t \*\*p = &cache[set\_index].line; \*p; p = &((\*p)->next)){  // printf("\n%pd\n", \*p);  if((\*p)->tag == block\_addr){  flag = 1 | flag;  if(verbosity) printf(" hit");  hit\_count++;  if(\*p == cache[set\_index].line) break;//already the head  cache\_line\_t \*tmp = (\*p)->next;  (\*p)->next = cache[set\_index].line;  cache[set\_index].line = \*p;  \*p = tmp;  break;  } else if((\*p)->next == NULL && cache[set\_index].size == E) {  free(\*p);  \*p = NULL;  break;  }  }  if(!flag){//not find  if(verbosity) printf(" miss");  miss\_count++;  cache\_line\_t \*newline = (cache\_line\_t\*)malloc(sizeof(cache\_line\_t) + B);  newline->tag = block\_addr;  newline->next = cache[set\_index].line;  cache[set\_index].line = newline;  if(cache[set\_index].size == E){//already overflow  if(verbosity) printf(" eviction");  eviction\_count++;  } else {  cache[set\_index].size++;  }  }  } |

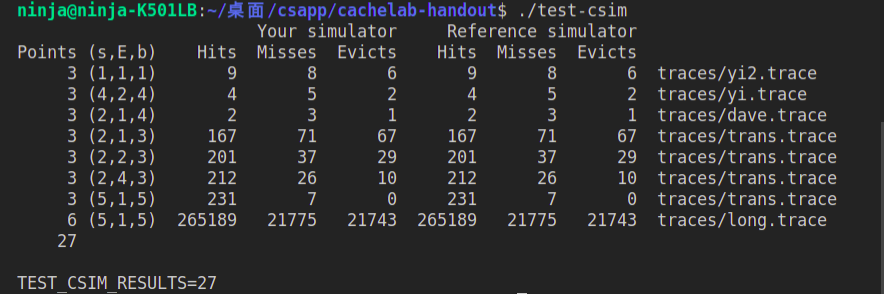
其中，为了在遍历前向链表的时候进行删除操作方便，需要指向删除块的指针域，于是可以通过指针的指针来修改上一个块的指针域。

* 1. **实验结果和分析**

通过对比自己写的csim与提供的参考程序csim-ref，可以发现两者结果完全一致。



使用test-csim测试程序，所有测试样例皆通过，得到满分27分



# 第二部分：矩阵转置优化

* 1. **实验目的**

1. 加深cache缓存结构对程序性能的影响和理解

2. 在cache层面对软件进行优化

* 1. **实验环境**

Linux 64bit + valgrind + gcc + python2

* 1. **实验思路**

这题乍一看无从下手，但是当仔细分析了块与块之间如何发生冲突之后再看问题其实并没有那么困难。所以在实验之中，先针对每个类型的矩阵观察哪些块交换可能发生冲突，然后再制定相应的策略去解决冲突。

在分析前，先观察到有

static int A[256][256];

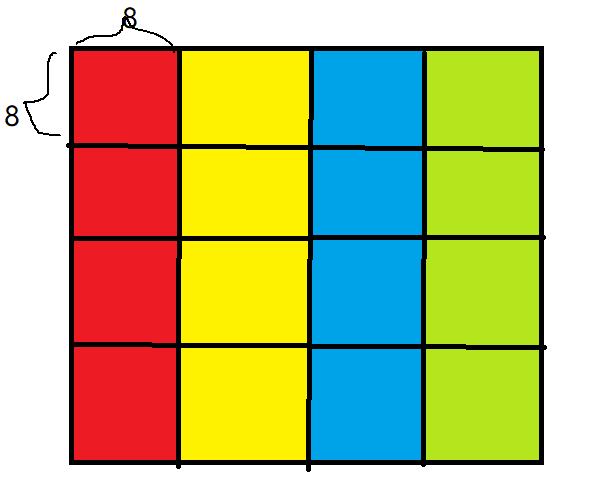
static int B[256][256];

并且注意到对于这个part2的实验，采取的是直接相联的方式，直接相联的块大小为32bytes，cache行数为32行。这表明在题目的三种用例中，尽管其所选取矩阵的大小不同，但是都有A与B相应位置发生冲突。

接下来对三种类型的矩阵单独进行分析

* + 1. **32 x 32 matrix**

32 x 32的矩阵一行有32个int类型即128个字节，所以每8行发生冲突，其对应的冲突部位如图所示：



于是可以很容易想到进行8x8分块，这样块内不会冲突，只有块与块之间相应的位置才会发生冲突。在图中相同颜色块的对应的位置会发生冲突。

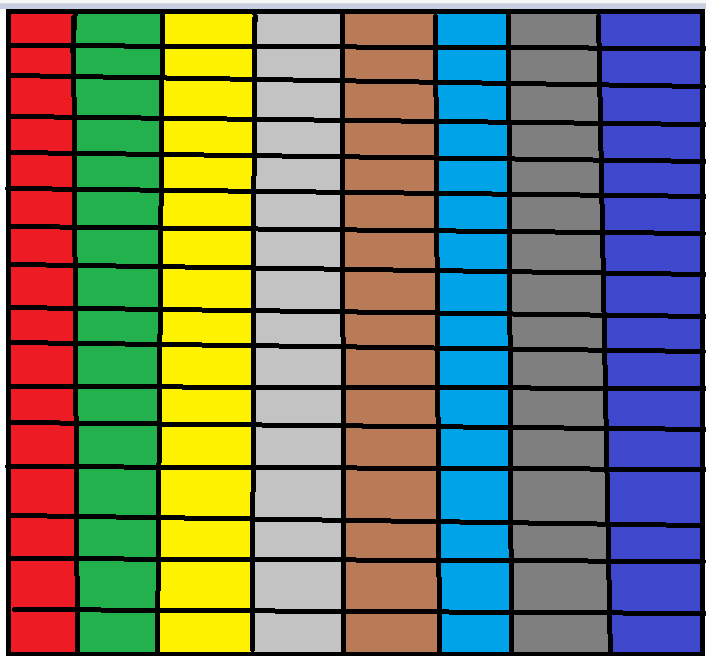
从A矩阵往B矩阵移动块，就要想到一个算法，使得B矩阵不容易把A矩阵对应的块给挤下来，这可以通过分段来实现。

|  |
| --- |
| int i, j, m, n;  for(m = 0; m < M; m += blk32){  for(n = 0; n < N; n += blk32){  for(i = 0; i < blk32; ++i){  for(j = 0; j <= i; ++j){  B[j+m][i+n] = A[i+n][j+m];  }  }  for(j = 0; j < blk32; ++j){  for(i = 0; i < j; ++i){  B[j+m][i+n] = A[i+n][j+m];  }  }  }  } |

在这里，巧妙的利用循环for(i = 0; i < blk32; ++i) for(j = 0; j <= i; ++j)，将使用过的i的块不断挤下来，而不是在外层循环的某一次中挤下来，可以减少块被替换的次数。于是通过没有使用辅助变量的方式，将cache被替换的风险得以降低。

* + 1. **64 x 64 matrix**

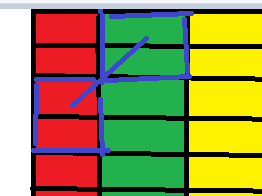
64 x 64的矩阵一行有64个int类型即256个字节，所以每4行发生冲突，其对应的冲突部位如图所示：



其中图中每个块都是4 x 8的块，相应同颜色4x8的块的对应位置发生冲突，而块与块之间不会发生冲突。

由于1个cache块是8个int类型的大小，同样和32x32的矩阵一样进行8x8分块，对于这样的8x8的块，可以分为两类。一类是与转置过去的矩阵不会发生冲突，另外一类是与转置过去的矩阵会发生冲突。

很容易看出第一类是不位于对角线上的8x8矩阵，例如



这类矩阵可以采用缓存局部变量与设计搜索路径的方式减少冲突。具体实现如下：

|  |
| --- |
| for(n = 0; n < N; n += blk64 \* 2){  for(m = 0; m < M; m += blk64 \* 2){  if(m == n) continue;  for(i = n; i < n + blk64; ++i){  for(j = m; j < m + blk64; ++j){  B[j][i] = A[i][j];  }  }  tmp1 = A[n][m+4];  tmp2 = A[n][m+5];  tmp3 = A[n][m+6];  tmp4 = A[n][m+7];  tmp5 = A[n+1][m+4];  tmp6 = A[n+1][m+5];  tmp7 = A[n+1][m+6];  tmp8 = A[n+1][m+7];  for(i = n + blk64; i < n + blk64 \* 2; ++i){  for(j = m; j < m + blk64; ++j){  B[j][i] = A[i][j];  }  }  for(i = n + blk64; i < n + blk64 \* 2; ++i){  for(j = m + blk64; j < m + blk64 \* 2; ++j){  B[j][i] = A[i][j];  }  }    B[m+4][n] = tmp1;  B[m+5][n] = tmp2;  B[m+6][n] = tmp3;  B[m+7][n] = tmp4;  B[m+4][n+1] = tmp5;  B[m+5][n+1] = tmp6;  B[m+6][n+1] = tmp7;  B[m+7][n+1] = tmp8;  for(i = n + blk64 / 2; i < n + blk64; ++i){  for(j = m + blk64; j < m + blk64\*2; ++j){  B[j][i] = A[i][j];  }  }  }  } |

另外一类便是对角线上的8x8矩阵，其转置过去会发生冲突，然而这种情况可以划归到32x32矩阵相应的解决冲突方式进行容易的处理。

|  |
| --- |
| int i, j, m, n;  int tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;  for(n = 0; n < N; n += blk64){//处理对角线处  for(m = n / (2\*blk64) \* (2\*blk64); m < n / (2\*blk64) \* (2\*blk64) + 2\*blk64; m += blk64){  tmp1 = A[n][m+1];  tmp2 = A[n][m+2];  tmp3 = A[n][m+3];  B[m][n] = A[n][m];  tmp4 = A[n+1][m+2];  tmp5 = A[n+1][m+3];  B[m][n+1] = A[n+1][m];  B[m+1][n+1] = A[n+1][m+1];  tmp6 = A[n+2][m+3];  B[m][n+2] = A[n+2][m];  B[m+1][n+2] = A[n+2][m+1];  B[m+2][n+2] = A[n+2][m+2];    B[m][n+3] = A[n+3][m];  B[m+1][n+3] = A[n+3][m+1];  B[m+2][n+3] = A[n+3][m+2];  B[m+3][n+3] = A[n+3][m+3];  B[m+1][n] = tmp1;  B[m+2][n] = tmp2;  B[m+3][n] = tmp3;  B[m+2][n+1] = tmp4;  B[m+3][n+1] = tmp5;  B[m+3][n+2] = tmp6;  }  } |

* + 1. **61 x 67 matrix**

61x67矩阵事实上运用解决32x32的方式就可以解决

|  |
| --- |
| int i, j, m, n;  for(m = 0; m < M; m += blk61){  for(n = 0; n < N; n += blk67){  for(i = n; i < blk67+n && i < N; ++i){  for(j = m; j < blk61+m && j <= i && j < M; ++j){  B[j][i] = A[i][j];  }  }  for(j = m; j < blk61+m && j < M; ++j){  for(i = n; i < blk67+n && i < j && i < N; ++i){  B[j][i] = A[i][j];  }  }  }  } |

* 1. **实验结果和分析**

# 总结和体会

这次实验的第一部分较为简单，用软件模拟cache的实现，主要是实现了cache的LRU替换，用数组hash+链表的结构就可以实现。

第二部分的32x32与61x67都用的同样的技术，通过改变一个块内的转置次序来实现减少冲突。而64x64的部分自己写了比较久，先用32x32的naïve的实现，发现

# 对实验课程的改进建议