# Cache算法设计、实现与比较

姓名：莫海 学号：2019211356

## 1.实验条件以及环境

### 1.1 实验条件

假定已知：

* Cache size: 128 KB.
* Block size: 8 Bytes
* For a 64-bit memory address.
* Cache replacement algorithm: LRU

### 1.2 实验环境

* 操作系统：Linux
* 语言：python

### 1.3文件说明

* Trace文件夹为源数据所在文件
* Cache.py为实验源码
* Log文件夹为实验结果，其中DirectingMaping文件夹为直接映射实验结果，GroupConnection为组关联映射实验结果，MRUGroupConnection为使用MRU方法优化后的组关联映射结果，MultiColums优化后实验结果。他们中的result.log文件保存的时每组数据的准确率等。文件夹中对应的trace.log为缺失的log文件。
* cache\_2019211356\_莫海.xlsx为程序跑出的数据整理
* cache\_2019211356\_莫海.vsdx为报告中所划的矢量流程图

## 2. 实验设计

### 2.1 cache设计

Cache的大小为128KB，块大小为8Bytes，可由此得出cache系统一共块，所以可以得到cache的index为14位，offset为3位，tag为64-14-3=47位。Cache的结构可由表2-1所示。

|  |  |  |
| --- | --- | --- |
| Tag | Index | Offset |
| 47 | 14 | 3 |

表2-1 cache结构

### 2.2 直接映射DM方法组织Cache设计

直接映射可以看成路数为1的组关联映射，主存储器中一块只能映射到Cache的一个特定的块中，程序设计的流程图如图2-1所示。



图2-1直接映射程序流程图

主存与缓存分成相同大小的数据块，对应的主存块映射到cache中，由于路数为1，每次cache不命中时没有额外的缓存可用，直接将缓存中的数据替换出cache，因此不用设计LRU算法用于cache的块替换策略。

### 2.3 组关联cache设计

跟直接映射不同，直接映射相当于每一块相当于一个组，因此每个组中只能保存一个数据，数据替换频繁，为了解决这个问题，自然的想到增加每个组的数据块的个数，使得数据可以在cache中驻留一段时间，减少数据的替换。组关联程序设计如何2-2所示。



图2-2组关联映射cache流程图

在进行映射之前需要对cache在逻辑上进行分组，每组的大小为对应关联路数个块，对地址划分时与直接映射是一样的，唯一不同的是在得到地址的index之后不直接映射到对应行号为index的cache上，而是映射到对应的组上，再循环遍历组内的所有块，要是有对应的块的tag与该地址的tag相等，则命中，否则miss。其映射到对应组的起始块为(index//group\_size)\*group\_size。

该过程中涉及到LRU算法对组内块进行淘汰，LRU算法的设计与之后的MRU和MultiColums中的LRU算法设计完全一致，在之后就不在重述，LRU算法的设计流程图如图2-3所示。



图2-3 LRU模块流程图

使用LRU算法对组内块进行状态的跟新主要分为两个分支，当为hit分支时，简单的将命中块的LRU值设置为1，其他的块整体加1即可，当为miss分支时，首先判断组内是否还有空块，要是存在空块直接填上并跟新LRU的值即可，要是没有空块需要找到最大的LUR的值对应的块之后，替换该块为新的数据块，并跟新相应的LRU值。

### 2.4 MRU算法对cache的优化设计

在组关联的基础上，使用MRU算法对cache进行优化，主要是想增加一次命中的概率，以提升cache的命中时间，主要设计与组关联一致，不同点在于，在循环遍历组内所有块之前，首先访问一个长度为group\_size的数组，这个数组保存的是，上一次映射到这组的数据访问的是该组内的第几块数据，从这个数组中得到改组对应的元素，并根据这个元素去访问组内的块，要是没有命中，再根据组关联映射的方法循环访问该组。该算法主要是利用了以局部性原理以及空间换时间的思想。

### 2.5 MultiColum算法对cache的优化设计

MultiColum算法使用与MRU算法不同，它使用的是tag的后几位作为一次命中时访问的组内块号，要是没有命中，再访问该块的后选位置，其算法流程如图2-4所示。



图2-4 MultiCloum算法流程图

首先该算法申请的cache比之前的cache多申请一个数据位用于保存候选位置的编码，比如组内第0块的候选位为0100，则编码成8保存再该数据位中，需要使用候选位置的时候再根据8编码成二进制的数进行访问。

在算法一次命中的情况下，只需要使用LRU算法跟新对应的LRU值即可。在候选位置命中的情况下，首先使用LRU算法跟新组内的LRU值，再交换命中块与主块的tag，offset，与LRU，主块上的候选位不需要修改。

在miss的情况下，要是有空位，则直接把数据放入空位，并把数据放到主块，修改之前主块上数据对应的主块候选位的值。如果没有空位，则根据LRU算法找出淘汰块的位置，并修改淘汰块上数据对应主块的候选位，把正在访问数据填入淘汰块，并交换淘汰块与主块上的数据，修改主块上数据对应的主块的后选位。自此算法结束。

## 3. 实验结果分析

### 3.1 使用直接映射结果分析

使用直接映射得到四个文件的命中率如图3-1所示。

图3-1 直接映射截图

由上图可知，不同的文件cache的命中率会有一定的差异

### 3.2 使用组相联映射结果分析

使用组相联映射得到的准确率结果如便3-2所示。



表3-2 不同路数组相联命中率

为了直观的观察，将表3-2的结果绘制成折线图，具体如图3-2所示。

图3-2 组相联映射折线图

由上可知，不同的文件，随着cache的路数增加，其命中率会相应增加

### 3.3 使用MRU优化结果分析

由于使用的淘汰算法没有改变，所以使用MRU算法对cache进行优化的时候，其命中率并没有改变，一次命中率的结果如表3-3所示，值观的表示位图3-3。



表3-3 MRU算法一次命中率

图3-3 MRU算法一次命中率

MRU算法的非一次命中率数据如表3-4所示，趋势如图3-4所示。



表3-4 MRU非一次命中率

图3-4 MUR非一次命中率

由上述图标可知，在使用MRU算法对cache优化时，其一次命中率会随着路数的增大而减小；而非一次命中率，会随着路数的增加而增加

### 3.4 使用Multi-Column优化结果分析

与MRU优化类似，由于没有淘汰算法，使用Multi-Column算法进行优化并不会影响算法的命中率，而与MRU算法不同的时，Multi-Column算法的一次命中率与非一次命中率会有所不同，其一次命中率数据如表3-5所示，趋势如图3-5所示



表3-5 Multi-Column算法一次命中率

图3-5 Multi-Column算法一次命中率

Multi-Column算法的非一次命中率数据如表3-6所示，趋势如图3-6所示。



表3-6 Multi-Column算法非一次命中率

图3-6 Multi-Column算法非一次命中率

由上图可知，Multi-Column算法在cache路数增加时一次命中率始终保持在一个较高的水平，其非一次命中率波动也不大。

### 3.5 比较说明MRU路预测方法、Multi-Column预测方法

使用MRU路预测时，在路数较小的时候，可以得到较高的一次命中率，但是当路数增加的时候，由于预测数组每次只能保存上次访问的组内块号，预测的命中率概率上等于1/路数，因此，一次命中的准确率会随着路数的增加而减少。

使用Multi-Column路预测方法时，每次命中之后，都会交换数据，使得最近主路上的数据始终是相对来说最新的，由于局部性原理，每次访问主路直接命中的概率依旧很大，相当于使用每次命中交换数据的代价，用于获取较高的一次命中率。

MRU：

* 优点：使用的资源较少，当cache 的路数较小的时候，可以获得较好的一次命中率。
* 缺点：当cache的路数增加时，一次命中率减小，与传统的cache相比，还是需要而外的空间来进行预测。

Multi-Column：

* 优点：不管cache的路数如何改变，其一次命中率始终维持在一个较高的水平，通过候选位的方法，与普通的多路组相联相比，可以更块的找到某地址是否命中。
* 缺点：每次命中后需要进行数据交换与候选位修改，需要花费一定的计算资源，设计也相对来说比较复杂。与普通多路组相联相比，需要额外的空间用于保存候选位。

### 3.6 当其他条件一定，只改变cache大小实验结果分析

当使用4位组相联时，在不改变其他条件的情况下，改变块大小后，其命中率数值为表3-7所示，其趋势为图3-7所示。



表3-7 不同块大小名中率

图3-7 不同块大小准确率

又上可只，在只改变块的大小的情况下，cache的命中率会随着块的大小的增加而增加，开始时，块的大小比较小，存储的数据量较小，比较难命中，由于LRU算法的淘汰效果，会导致之前命中的块很快被淘汰，因此命中率较低，随着块大小的增加而增加，但是增加到一定程度的时候，会导致块的大小过大，块内保存了很多无用的数据，而且由于cache的容量一定，块大小的增加会导致cache的行数减小，当cache的行数太少时，会起到反效果，导致cache的命中率降低，但是实验中给出的块还不够大，不能显示出cache命中率随块大小增加而减小的数据。