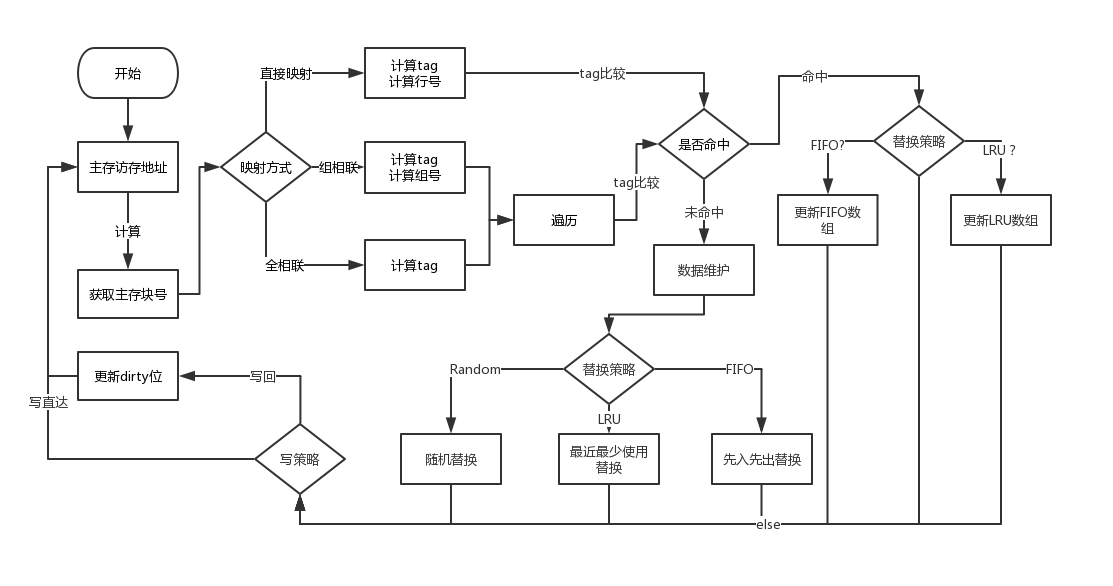
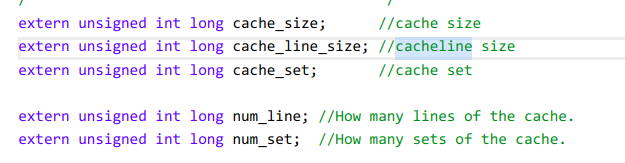
1. **Cache流程图**

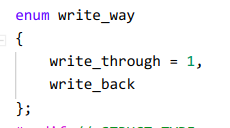
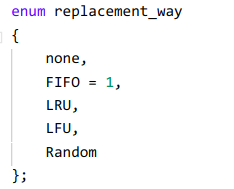
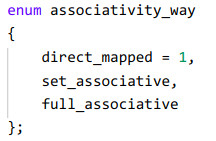


1. **Cache基本配置**

Cache大小和Cache行，定义全局变量表示cache大小（单位KB），cache行大小（单位B），如果是组相联或者全相联，还需要有组数。



定义枚举类型，分别表示映射方式（直接映射，组相联和全相联），替换策略（直接替换，先入先出，最不经常使用，最近最不经常使用，随机），写策略（写直达和写回）。

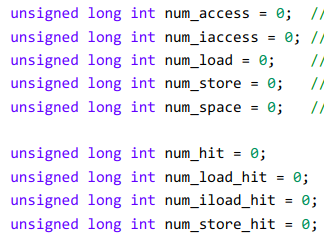


使用bitset来表示cache对象，每个cache行用32位表示（其中31表示是否有效，也就是是否存入，30表示是否命中，29表示dirty位，主要在写回策略中用到，剩余29位表示bit）。

在替换策略的LRU中，使用数组表示每个cache行的优先等级，最低的那个会在替换时被替换掉。



本次试验要求统计L1缓存的命中率，因此使用变量存储访存次数以及命中次数。



1. **Cache映射方式**
2. **Cache基本结构**

Cache行数 = Cache大小 / Cache行大小

块大小 = Cache行大小

块内地址长度 = log2(Cache行大小)

|  |  |
| --- | --- |
| 主存块号 | 块内地址 |

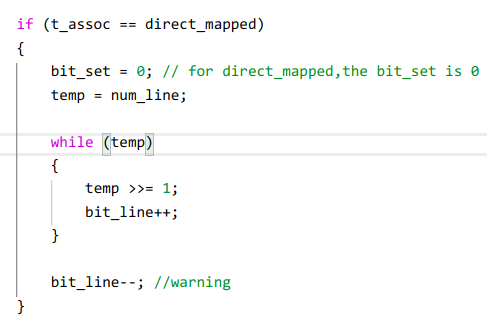
主存块号 = 主存地址 / 块大小 主存地址 >> 块内地址长度

块内地址 = 主存地址 % 块大小

1. **直接映射**

|  |  |  |
| --- | --- | --- |
| 标记 | Cache行号 | 块内地址 |

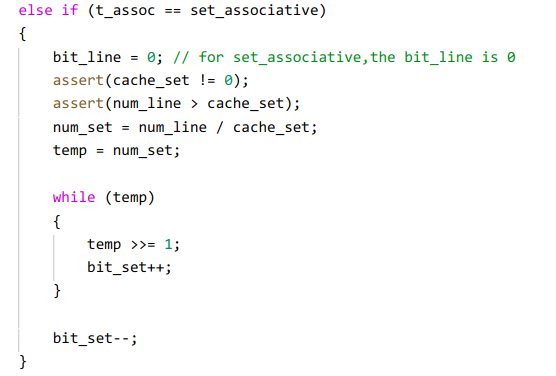
主存块映射到Cache的一行，替换的时候直接替换。如果将数据放入Cache，同时需要将valid位设置为1。



1. **组相联**

|  |  |  |
| --- | --- | --- |
| 标记 | Cache组号 | 块内地址 |

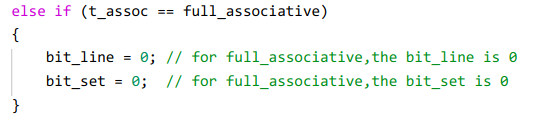
主存映射到Cache某一行的任何位置，这一行为一组（实际上就是组内全相联，组间直接映射）。



1. **全相联**

|  |  |
| --- | --- |
| 标记 | 块内地址 |

主存可以映射到Cache的任意一行，不过这种方式在判断是否命中的时候，需要完全遍历一遍，因此速度比较慢，但是命中率会比较高。



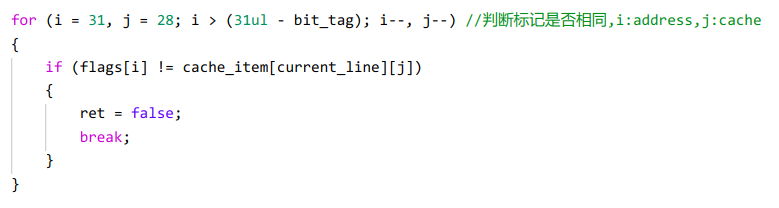
1. **Cache命中检查机制**
2. **处理trace中地址**

将trace中的内存地址转换为32位二进制位串，然后进行处理。

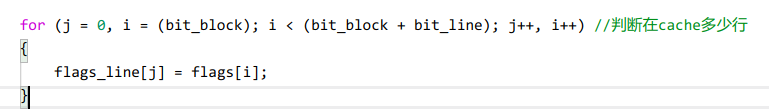


1. **直接映射命中查询**

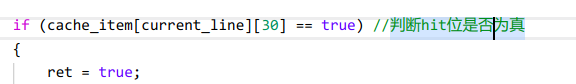
目标标记 T = 主存块号 / Cache行数



目标行号 L = 主存块号 % Cache行数

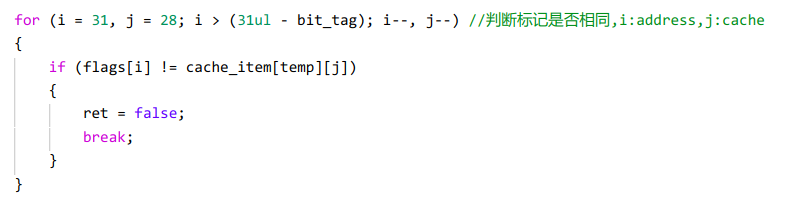


(valid[L] && tag[L] == T) → hit

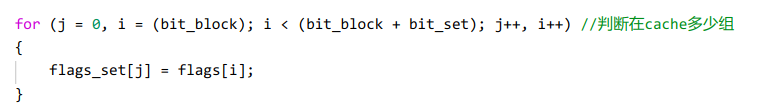


1. **组相联命中查询**

目标标记 T = 主存块号 / G



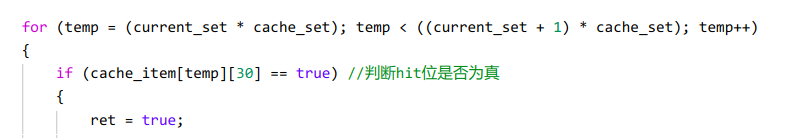
目标组号 g = 主存块号 % G



for(i = g\*S to (g+1)\*S-1)

(valid[i] && tag[i] == T) → hit

　miss



1. **全相联命中查询**

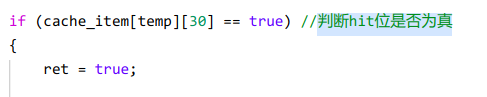
目标标记 T = 主存块号

for(i = 0 to Cache行数-1)

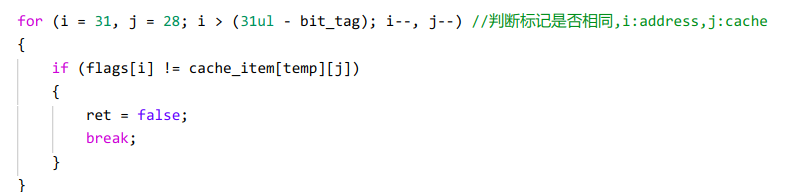
(valid[i] && tag[i] == T) → hit

　Miss

判断hit位是否为真

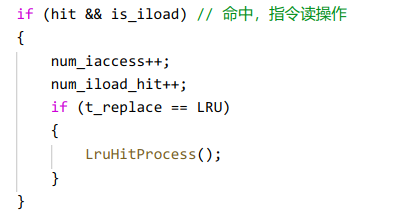


判断tag是否相同

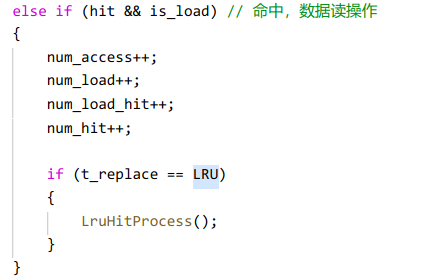


1. **读操作命中处理**

读指令命中，需要将i-cache的访存次数以及命中次数+1，然后判断替换策略，决定是否进行替换。

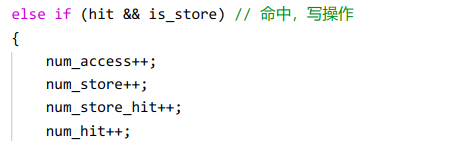


数据读命中，将d-cache的访存次数以及读次数以及相关命中次数都进行+1。然后判断替换策略，决定是否进行替换。

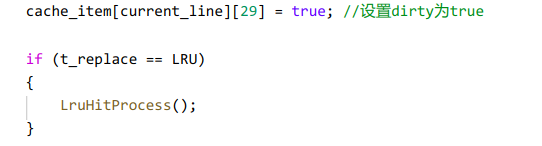


1. **写操作命中处理**

数据写命中，将d-cache的访存次数以及读次数以及相关命中次数都进行+1。

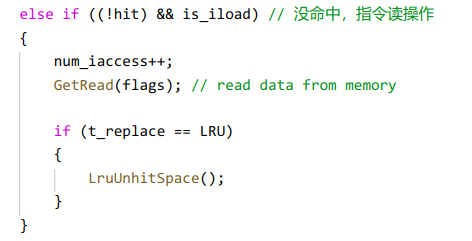


同时如果是写回策略，需要设置dirty位，同时还要考虑LRU替换策略。

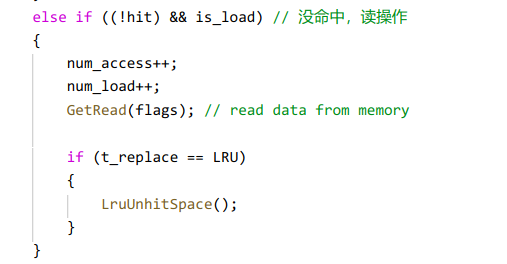


1. **读操作不命中处理**

读指令不命中，需要将i-cache的访存次数+1，然后从内存中读取数据，判断替换策略，决定是否进行替换。



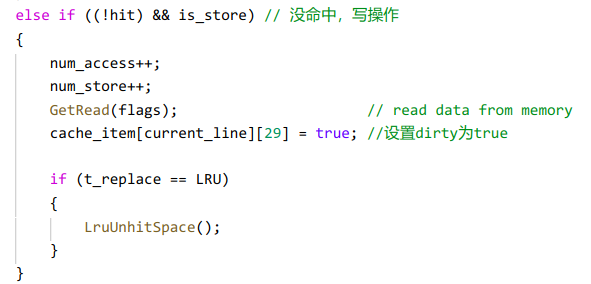
数据读不命中，将d-cache的访存次数以及读次数都进行+1。然后从内存中读取数据，判断替换策略，决定是否进行替换。



1. **写操作不命中处理**

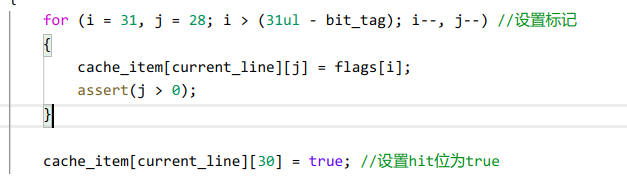
数据写不命中，将d-cache的访存次数以及写次数都进行+1。然后从内存中读取数据，判断替换策略，决定是否进行替换。

同时因为向cache中写数据，需要将脏位置为1。



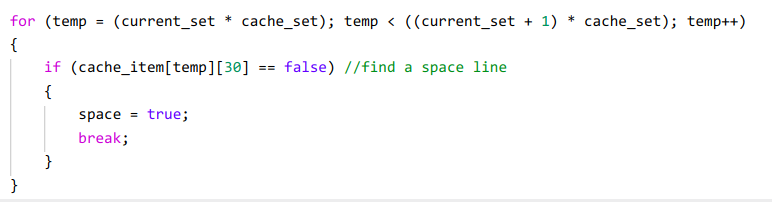
1. **Cache从内存中读取数据**
2. **直接映射**

直接从主存块中读取对应的tag标记放到对应的cache行，然后设置hit位为1。

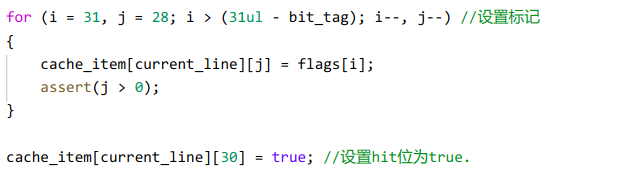


1. **组相联**

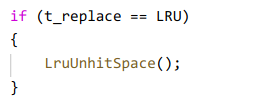
对于组相联，首先找到对应的组号所对应的组，然后遍历这个Cache组，找到空的Cache行。



找到空的cache行以后，设置好标记位，并把hit位设置为1。



如果找不到空的Cache块，就先判断替换策略。

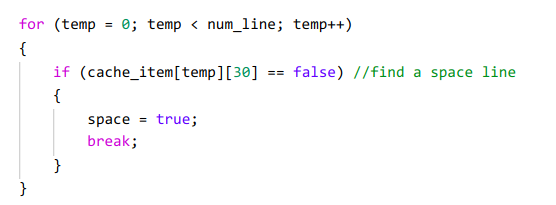


找到组中需要被替换的Cache块，然后进行替换。

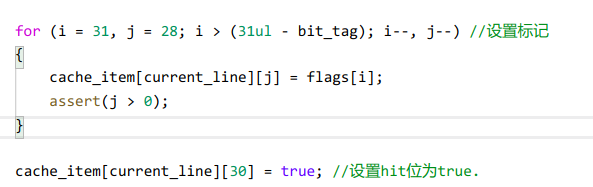


1. **全相联**

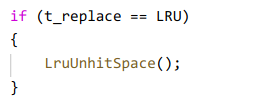
首先需要遍历整个Cache，然后找到一个空行。



找到空行以后，设置好标记位，然后设置hit位为1即可。



如果找不到空行，就先判断替换策略。



找到需要替换的行，然后进行替换。



1. **Cache替换策略**

替换策略只适用于全相联和组相联，直接映射当中就是直接进行替换。。

1. **Random替换**

srand(seed)、srand((unsigned)time(NULL))

0~N-1的随机数：rand() % N

组相联：



全相联：



1. **LRU替换策略**

每次访问时，被访问的那行计数清零，其他行加1

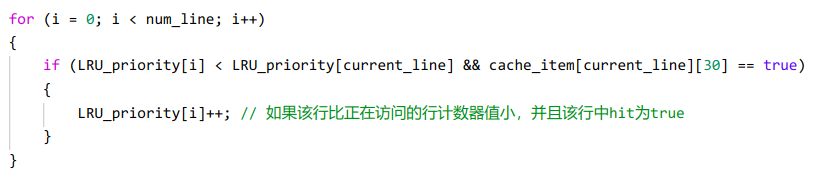
需要替换时，将计数最大的行换出

* 如果在访存时，使用的替换策略为LRU，而且**命中**

对于全相联，将当前行的优先级设置为0



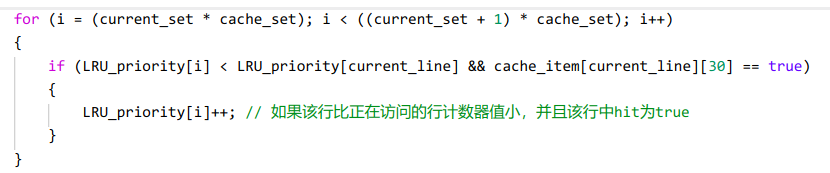
其他行LRU优先级增加1。



对于组相联，组内当前块的优先级设置为0。

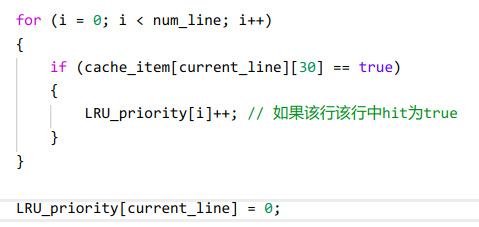


其他块LRU优先级＋1。

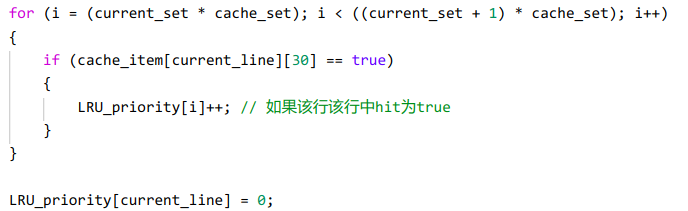


* 如果在访存时，使用的替换策略为LRU，但是没有命中，不过**还有**多余的空的Cache块（对于组相联的组中或者全相联的Cache中）

对于全相联，当前行优先级为1，其他非空行优先级增加。

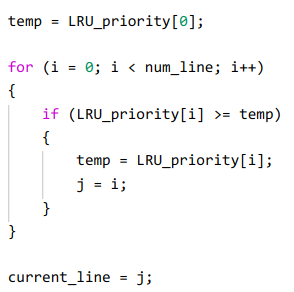


对于组相联，当前块优先级为1，组内其他非空块优先级增加。

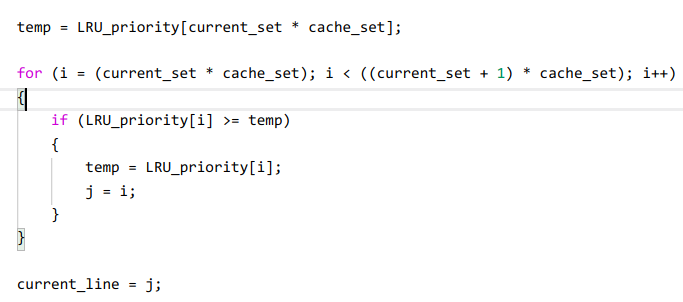


* 如果在访存时，使用的替换策略为LRU，但是没有命中，而且**没有**多余的空的Cache块

如果为全相联，遍历整个LRU数组，找出优先级最大的Cache行，进行替换。



如果为组相联，遍历LRU数组中当前组的所有块，找出优先级最大的Cache行，进行替换。



1. **FIFO**

新开一个FIF0数组，记录块的添加时间。

当需要替换时，遍历全相联的每个cache块，找出添加时间最小且有效的块。

如果为组相联，就遍历组内，完成相关替换。

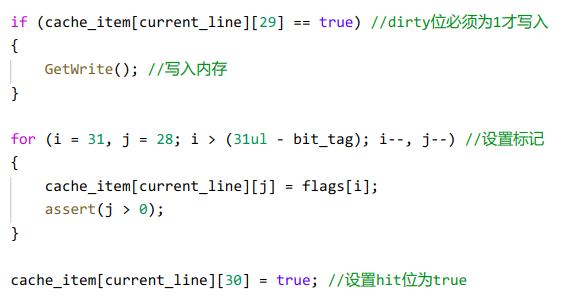
1. **直接替换**

对于直接映射，每次发生冲突都将cache行进行直接替换。

1. **完成替换后策略**

对于写回法，只有dirty位为1才进行写入，然后需要进行tag位的设置，以及将当前行的hit位设置为true。

对于写直达法，无需判断dirty位，直接写即可。同时也无需设置hit位。



1. **Cache写策略**
2. **写直达法**

写cache时，cache与主存同时发生写修改。写直达法用来保护cache与主存的一致性，但是当有多个cache都拷贝这个数据时，更新这个cache和主存时会造成其它的cache中的数据的不一致。

1. **写回法**

模拟cache的写回策略，只有dirty位为1的才会写回。这时候将dirty位设为0，hit位也设置为0。

写回法的cache中的数据会与主存的不一致。为了识别cache中的数据是否与主存中的一致，cache中的每一块要增加一个记录信息位，以反映此行是否被CPU修改过。修改cache中某一块时设置这个位为浊（dirty）。根据这个位的值，cache中每一块都有两个状态：清（clean）和浊（dirty），在将新的值放入浊的块的时候，将原值写回到主存，否则，直接将新值存入这个块。



1. **测试样例以及效果**
2. **用时统计（因为cache用数组表示，因此全相联用时非常长）**

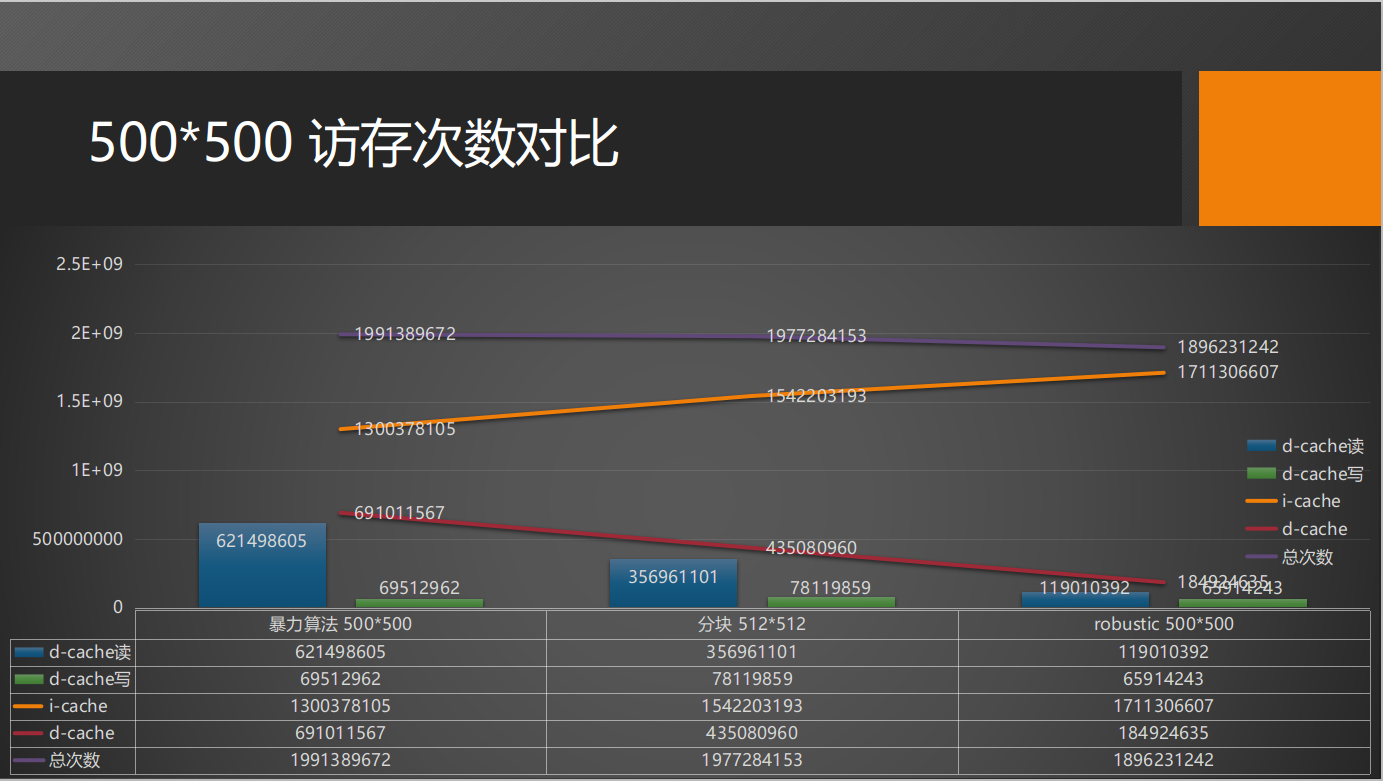
|  |  |  |
| --- | --- | --- |
| **策略 矩阵** | **500\*500** | **100\*100** |
| 直接映射 | 78min | 6min |
| 2-way 组相联 | 84min | 7min |
| 4-way 组相联 | 86min | 7min |
| 全相联 | 几天 | 15min |

1. **针对不同算法以及不同映射方式的矩阵乘法比较**

**矩阵大小分别为500\*500以及512\*512，生成的trace文件大约15GB（大约20亿行）多一点。**

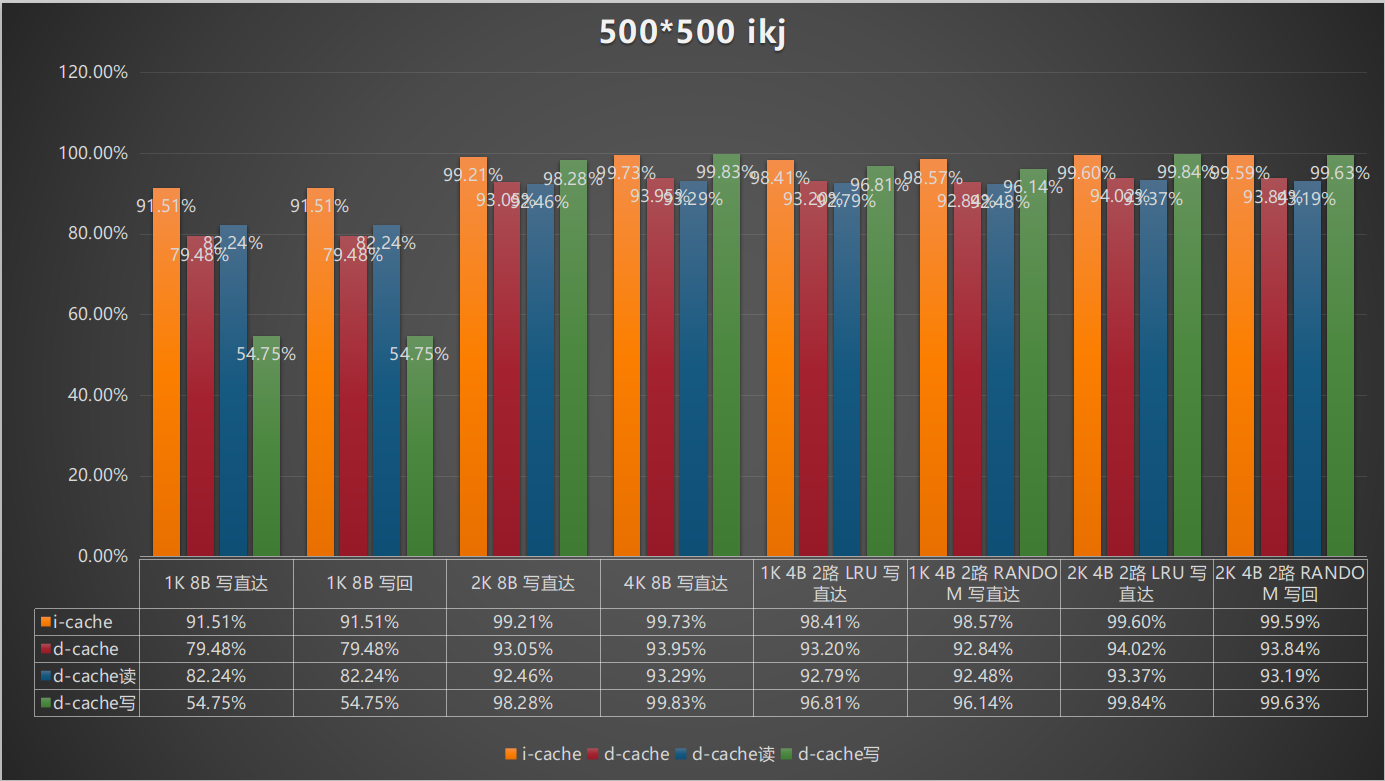


**这三种策略的访存次数**

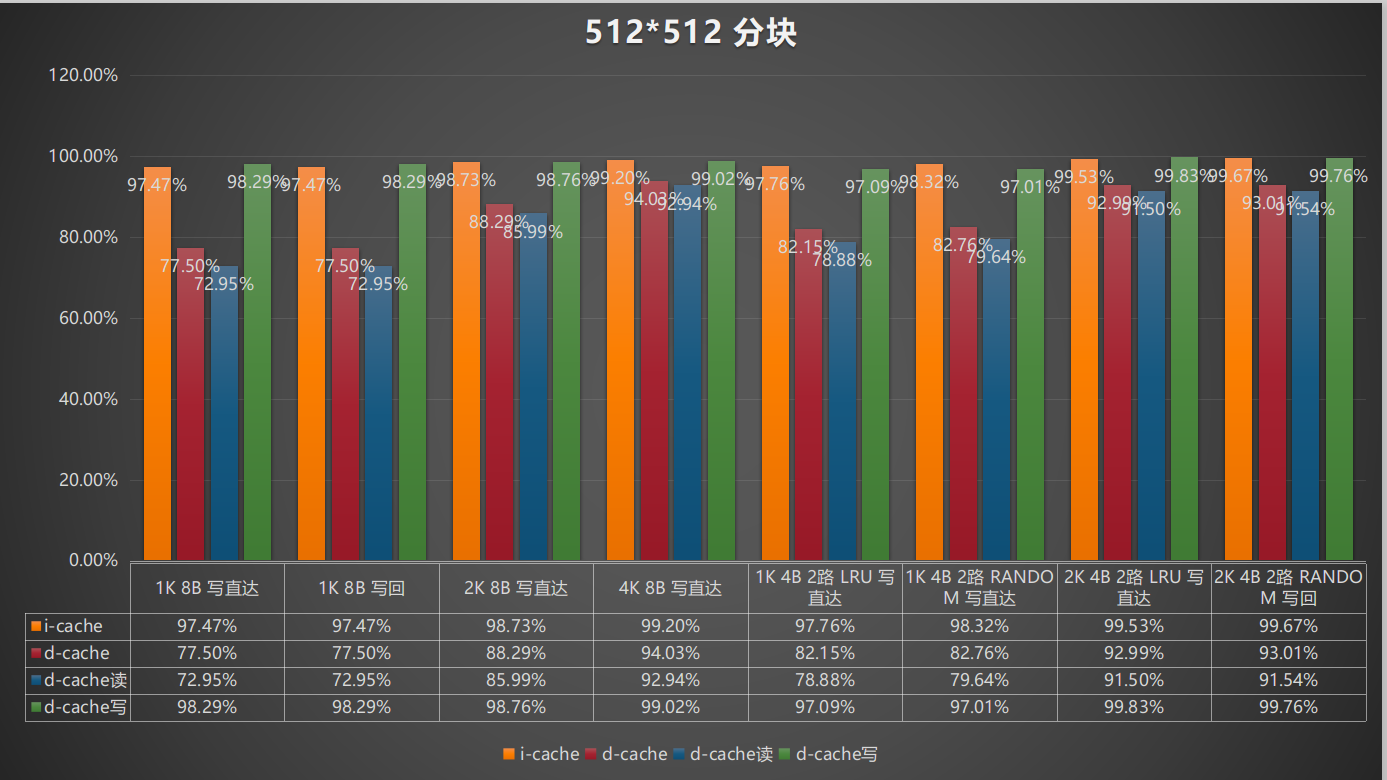


**直接映射与组相联对比**

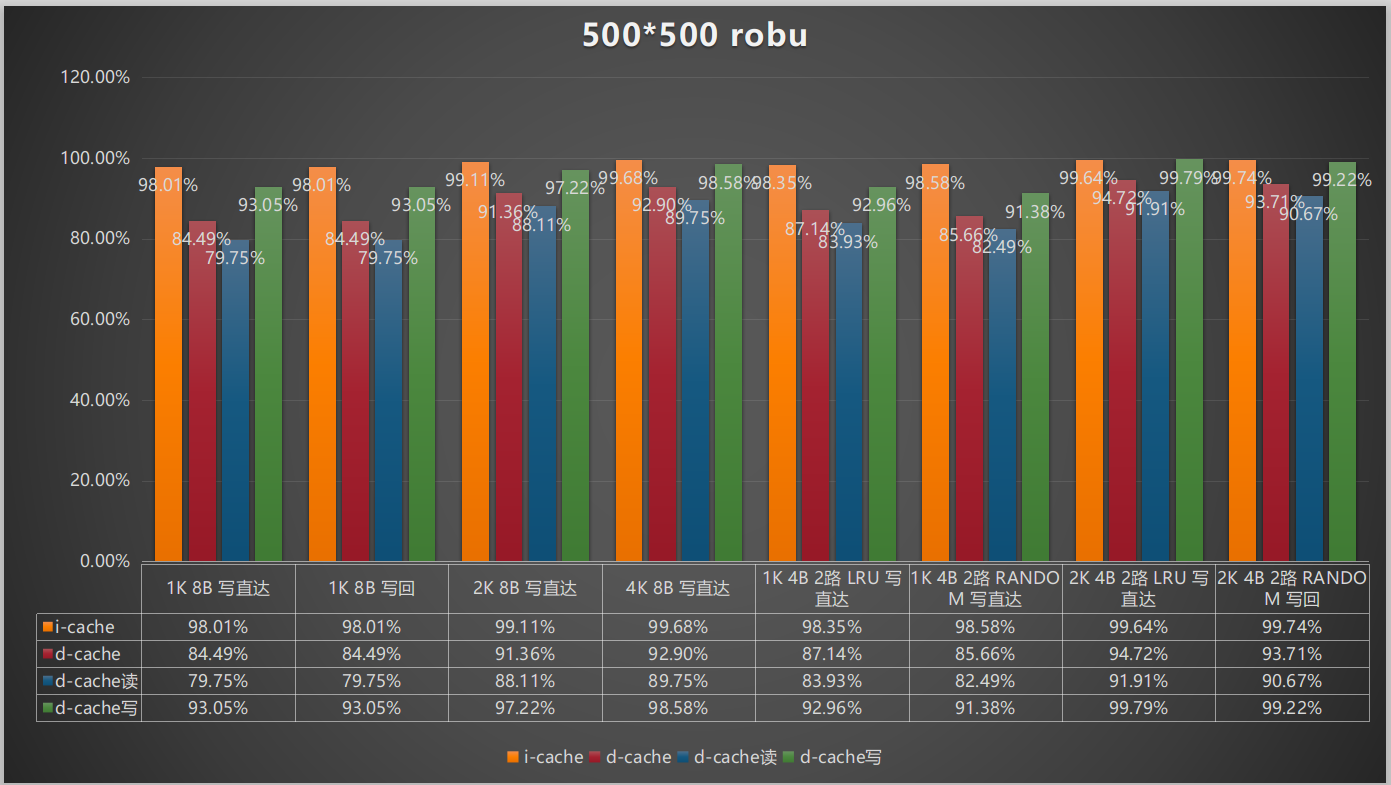
一种暴力算法



分块算法



Robustic



通过统计图对比分析可以得出，直接映射的命中率并不比组相联好，当然两种方式的i-cache命中率都比较高。

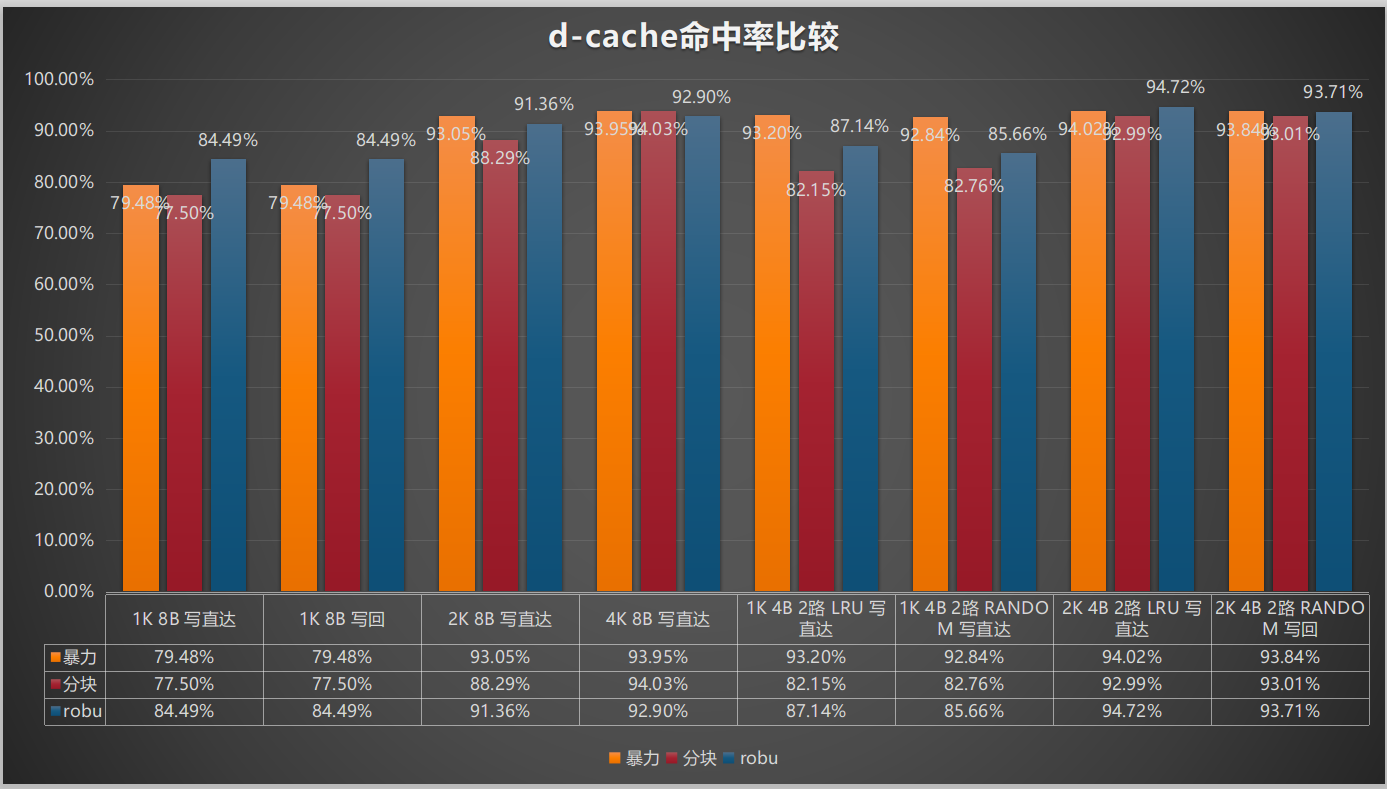
通过图中也可以看出，随着cache容量的增加，以及cache块的增加，命中率会有所上升。

LRU替换算法的命中率比随机替换好一些。

写回法和写直达法对于cache命中率没有影响，只会影响写的速度。

在这个例子中，暴力乘法的命中率竟然略微高一些，可能是分块算法设计的对于512\*512的矩阵并不是特别好。

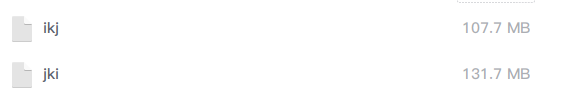
**单独拿出d-cache进行对比**



可以看出，在我所使用的这三种方式中，最后一种的效果最好。而最优化的暴力算法命中率与分块的那种相当，只是略微好一点点。

1. **针对两种暴力算法的比较（ikj以及jki）**

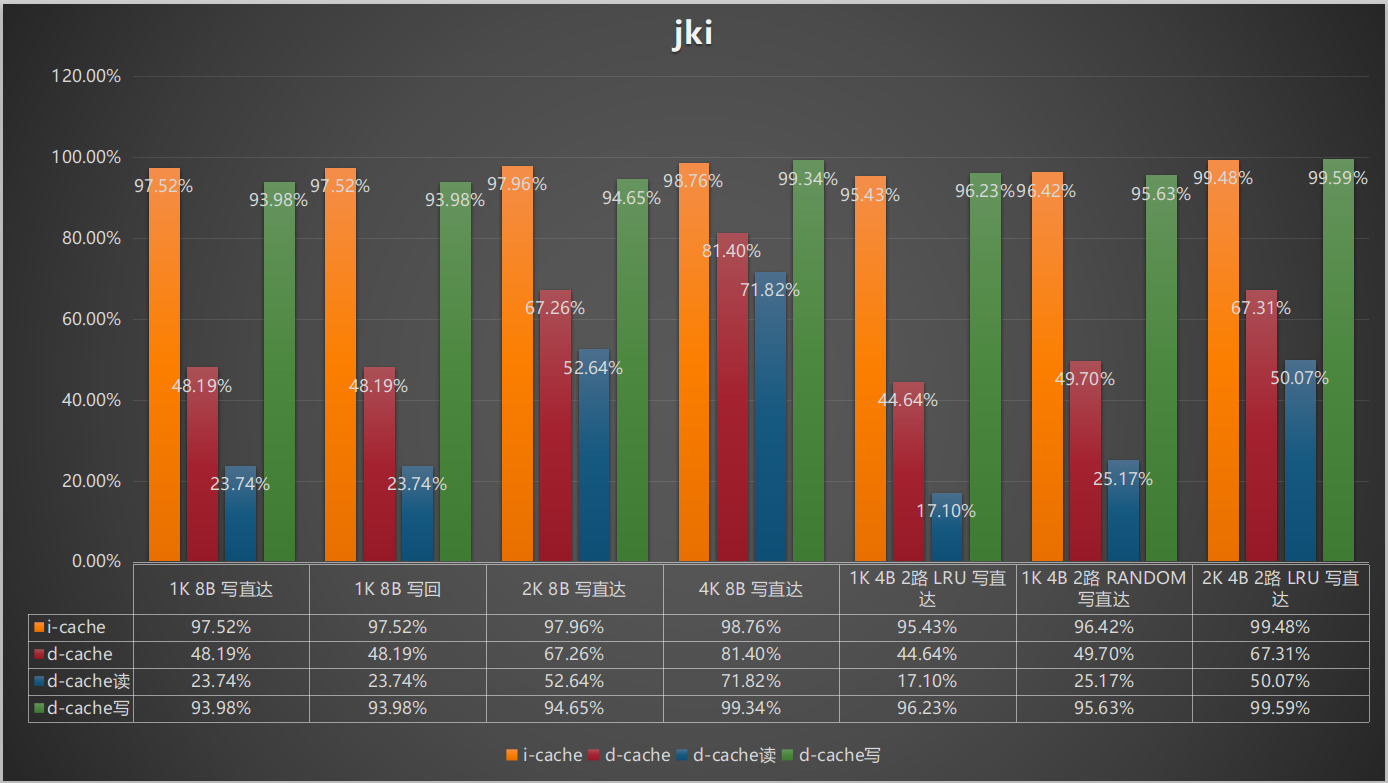
**矩阵大小均为100\*100，生成的trace文件大约100MB（大约一千万行）多一点。**



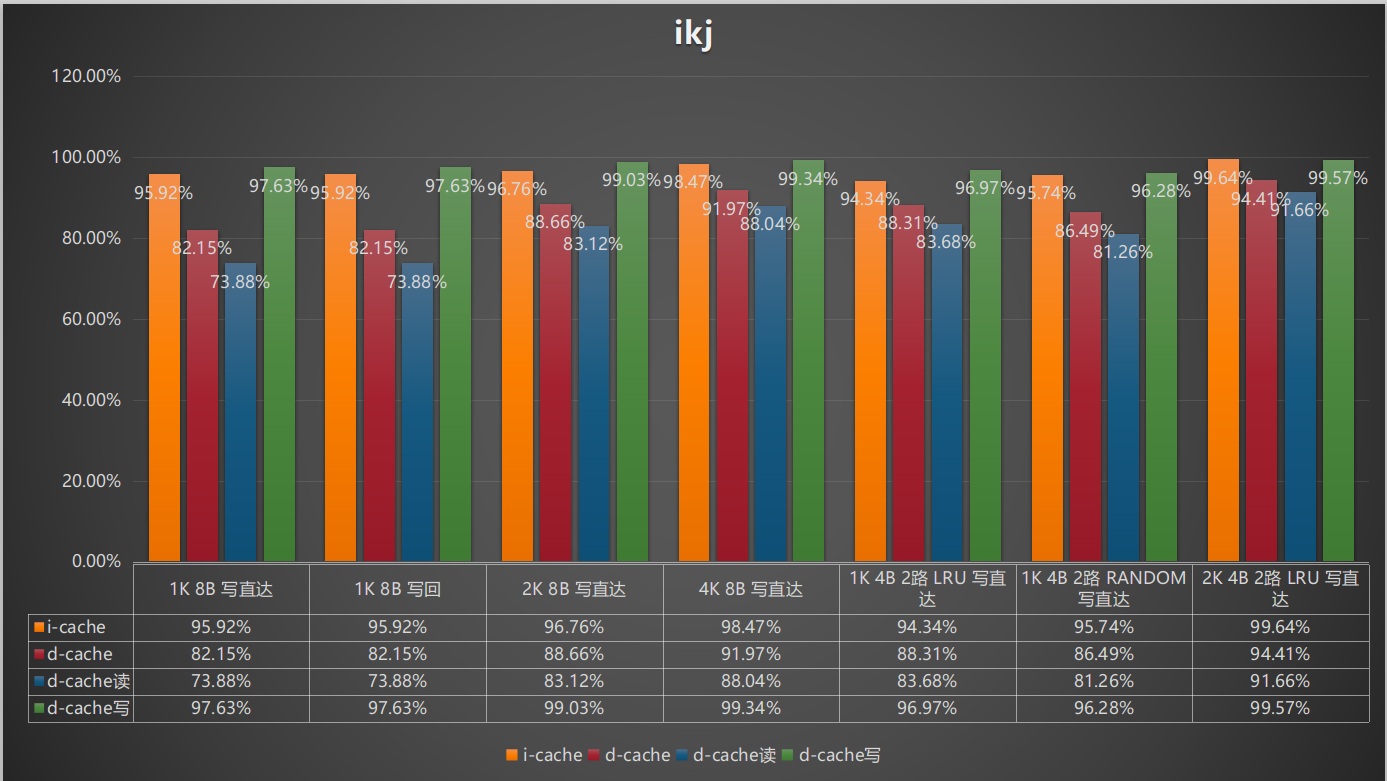
这两种的访存次数分别1200W次和1500W次

**直接映射和组相联对比**

jki



ikj

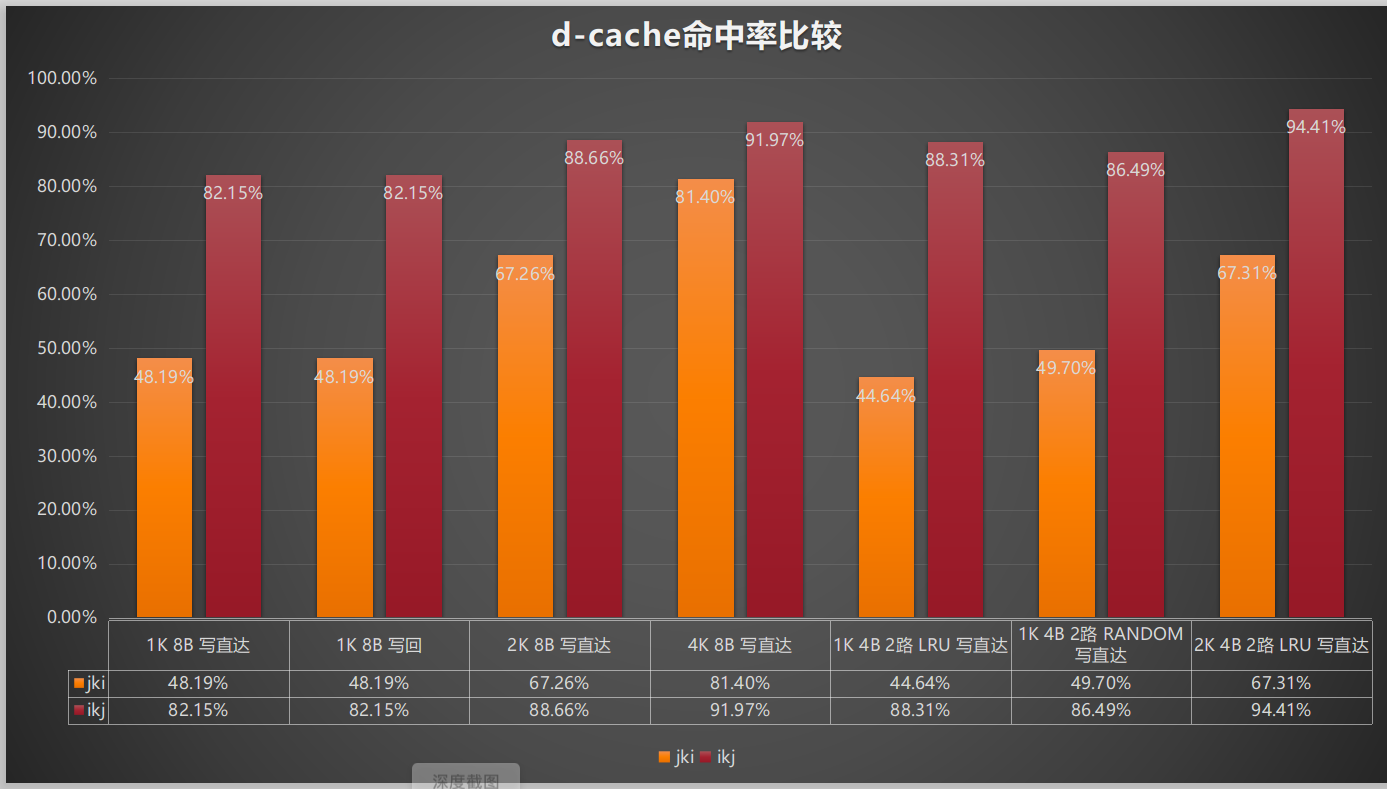


两个图的对比可以很明显看到，ikj这种算法明显更优。不过d-cache的读操作命中率并不高。

对于jki这种矩阵乘法，虽然看上去命中率可能有50%左右，但是抛去数组初始化时候的命中，在计算乘法时几乎没有什么命中，所以说这个程序的局部性是非常差的。

即便是使用组相联，对于第一种暴力算法来说，cache读数据的命中率仍然很低，所以仍然局部性非常差。

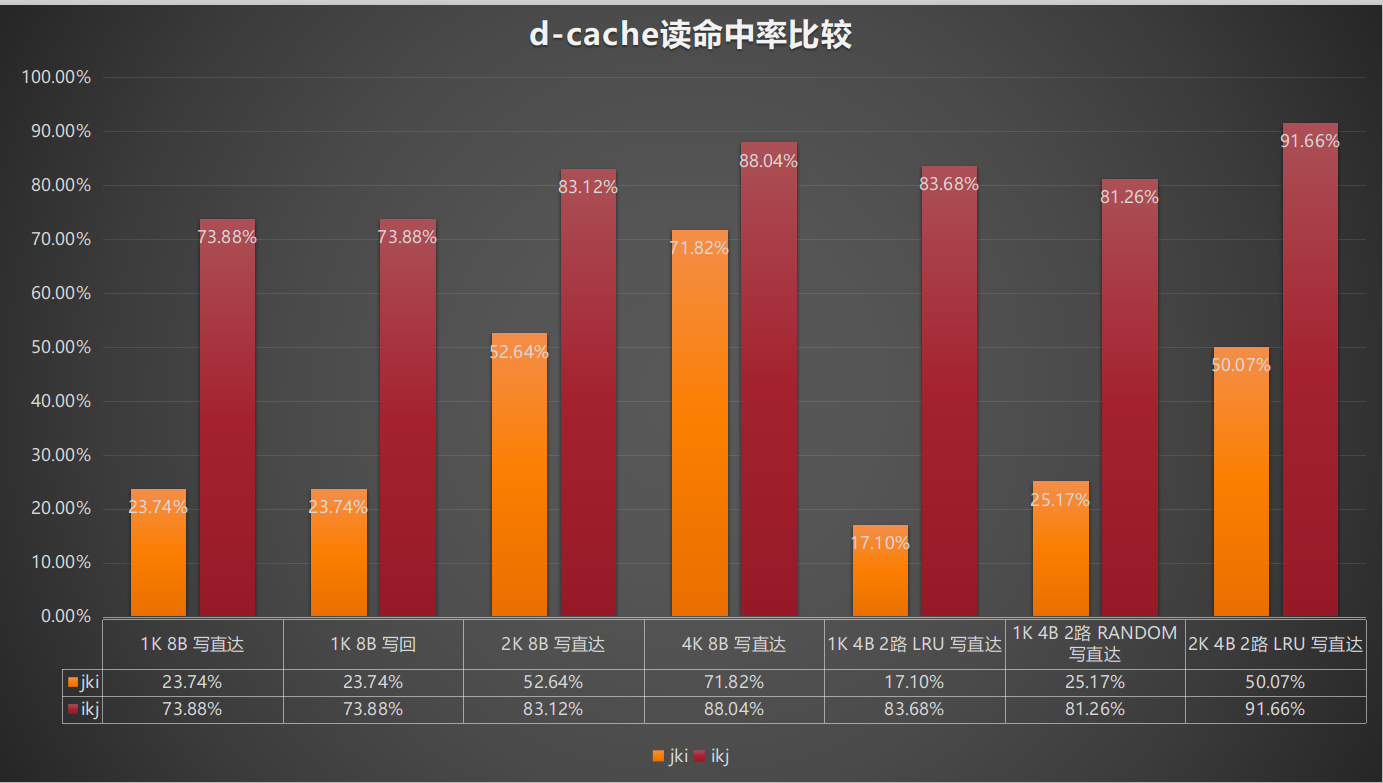
**两者d-cache命中率直接对比**



可以看到，ikj这种矩阵乘法远远好于jki。

**d-cache读命中率**

根据之前所统计的数据，d-cache的写命中率已经非常高，限制d-cache命中率的主要因素还是读数据。因此对读命中率也单独作了统计。



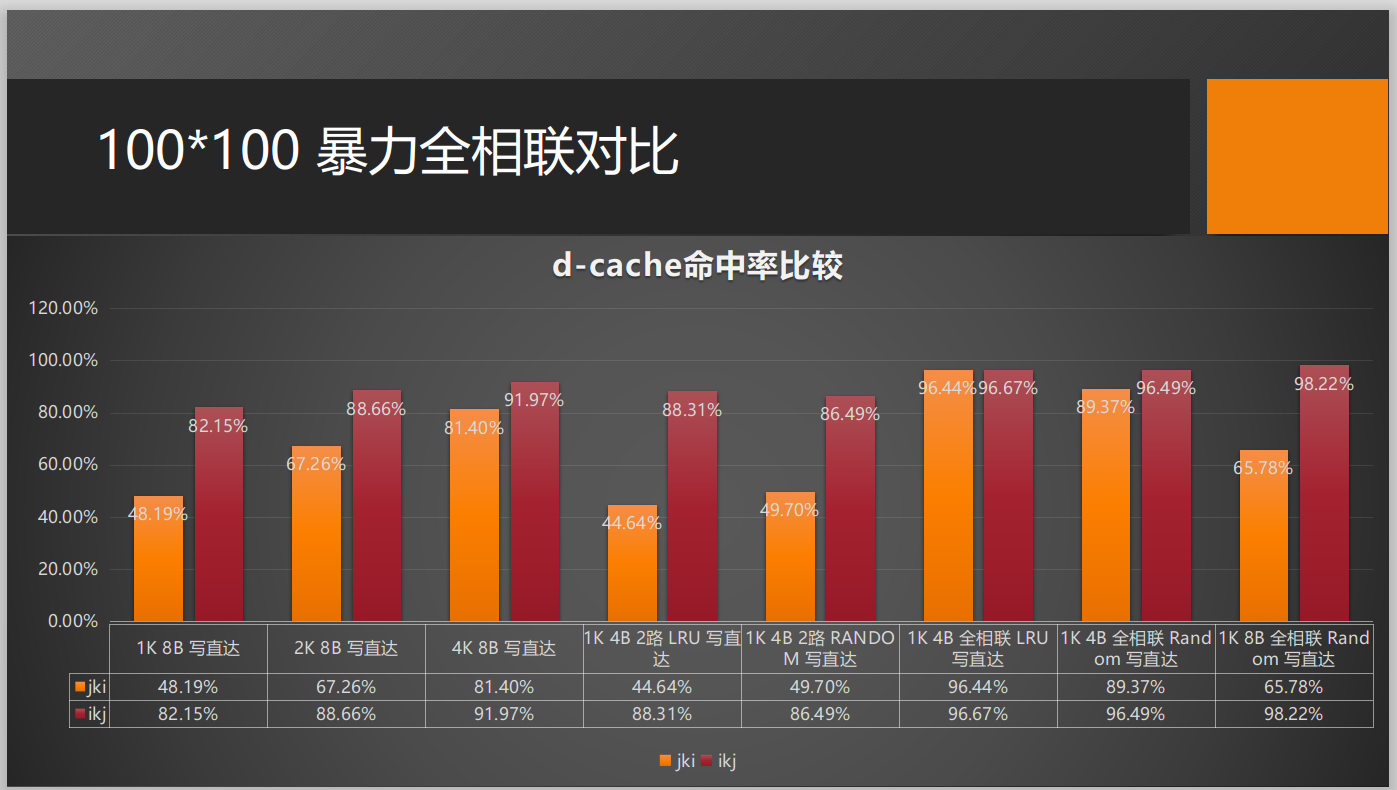
无论是直接相联还是组相联，jki这种暴力算法始终不理想（全相联肯定没有这方面的问题）。这跟乘法的时候的对于数组访问的无序性密不可分，局部性的不理想造成即便使用多路组相联仍然不能有效改善速度。

**两种算法全相联数据对比**

**左图为jki，右图为ikj。可以看到全相联情况下命中率都非常高。**



与直接映射和组相联的d-cache对比图



不过对于jki这种算法来说，使用LRU替换算法明显好于Random替换。

而对于ikj，这种现象并没有那么明显。

而且，如果cache块数量不那么足够的情况下，random替换方法的劣势会被放大。