ICS 课程报告——cache 机制分析

姓名：陈锐林 学号：21307130148

一、总览

本次课程报告我选择选题 2，分析不同类型 cache 机制设计。接下来将按顺序分析高速缓存的 3 种类型、当不命中时的不同替换策略、写策略和缓存友好的编程建议。

二、高速缓存的 3 种类型

这个部分会先分别简单回顾直接映射高速缓存、组相联高速缓存、全相联高速缓存的特点，说明各自的优缺点；最后会综合起来谈这三种高速缓存的适用场景。

接下来采用的定义与课本一致。一个地址为 m 位，包括 t 位的标记位、s 位的组索引和 b 位的块偏移。而高速缓存包括 $S = 2^s$ 组，每组 E 行；每行为 1 个有效位，t 个标记位，一个 $B = 2^b$ 位的缓存块。高速缓存的大小为 $C = B \times E \times S$ 的数据字节。

首先是直接映射高速缓存，此时即 $E = 1$ 的情况。抽取请求的字的三个过程：组选择、行匹配、字抽取。组选择由组索引确定；行匹配只需要考虑一行，将标记位相互对比、检查有效位；最后根据位偏移得到目标字。直接映射的优点在于实现很简单，找到了组就不用再找行了；电路实现简单。直接映射的缺点是可能无法对空间进行充分的利用，即使在空间局部性良好前提下。因为是只根据组进行索引，多次查找可能遇到同一组；在课本上就有一例，因为在来回调用数组 x 和 y 的过程中总是找同一组，导致有空的位置却不利用，来回抖动。书上的解决方法是在定义数组大小时下功夫，让 x 和 y 两个数组命中到不同组上；但我认为这是不够合理的，在编程过程中考虑到这样的调整是很难的，这种解决方法要我们对 cache 的各项数据了解得很透彻。

其次是全相联高速缓存。即 S=1 的情况，此时 m 位的地址不再包含组索引，而只需要考虑标记位和块偏移。行匹配需要考察所有行，因为每一行都有可能包含所需要的标记位。全相联高速缓存只有 1 组，所以优点在于空间利用率较高，不会因为有空余组不被映射到而浪费空间，只要有空行就能插入内存块。但是全相联高速缓存的缺点在于因为每一行的排列是无序的，逐次考察每一行会很费时间（与 cache 的设计初衷相悖），并行考察又需要很强的硬件支持，又大又快的全相联高速缓存很难实现；而且若我们假设在高速缓存的大小一样的前提下，显然每一行的 t 标记位占比更大，意味着储存时的无效信息（缓存块为有效的目标信息）更多，是对空间的不友善利用。

最后是组相联高速缓存，称为 E 路组相联高速缓存。组相联高速缓存的三个过程中只有行匹配更为复杂。在根据组索引找到组后，需要考察每一行的有效位+标记位，看是否能找到目标。组相联高速缓存是前两者的折衷方案，中和了两者的优点，在实现上比全相联高速缓存简单些；因为每个组中有更多的空间，比起直接映射高速缓存，造成 cache 冲突而每次都不命中、空间利用率低的问题能得到有效的改善。如此的实现方式也对我们利用 cache 的机制去编写程序是提出了较高的要求的；同时在对 cache 机制的设计上，如何考虑 E S 的选择也是更

复杂的。

接下来是这三种高速缓存的适用场景。可以用图书馆的例子来解释，一个图书馆可以先分为多个门类，如科幻作品、童书、专业书籍等。每一个门类又可以分为多个小类，如专业书籍又可分为人文社科、理工科等。紧接着又可以向下分为具体的专业。现在要从全部的书籍，可能是几十万本中的不同大门类中各选出一种，就可以采用直接映射高速缓存，不同书种类的书映射到不同组内，一本书一行；而如果是从最小的分类抽取，比如数学专业的书内有百来本，就可以采用全相联高速缓存，因为只要容量小、成本低，用全相联方式就较划算；而如果是要从 x 个类中各抽出 y 本书，就适用于组相联高速缓存，达到前两者的折中。总结一下，当高速缓存大小一定的前提下（C 相同）；C 较小时可以一股脑塞地采用全相联方式，若 C 很大时，采取直接映射实现简单，且因数量大，空间利用率低的问题也能较好解决；若要达到二者的平衡，采用组相联是较好的。但是组相联的 E S 怎么选择，需要根据具体情况分析，如课本关于直接映射抖动的例子，只要让 x 和 y 映射到不同的组（就不必人为更改数组大小），就不会遇到类似的情况。

三、不同替换策略

当不命中时需要替换行，组相联和全相联高速缓存一组内有多行，要替换哪一行值得深思，书上提到了三种替换策略：随机替换、LRU 策略、LFU 策略……接下来会给出这三种的简单实现，讨论它们的优缺点以及适用场景；最后会拓展性地讨论一些在 LRU 和 LFU 基础上搭建的策略。

首先是随机替换的策略，这是最简单的。优点在于实现很简单，随机选择其中一行替换，不需要额外的空间/硬件；缺点就在于随机是盲目的，可能会替换掉最近经常使用到的内容。

其次是 LRU 策略，所谓的最近最少使用策略。LRU 策略可以从生活找到很多类似的例子，比如微信界面，界面大小有限，只能展示最近的信息，而当有一条新信息时，就会提到界面顶部，会挤掉最久远的信息。写到这里的时候我想到之前做力扣时遇到的题目：146. LRU 缓存 - 力扣（LeetCode）。当时了解不够充分，不大理解缓存的概念，回头来看，通过简单的数据结构，我们就能模拟出 LRU 策略。这里要求我们以 o(1)的时间复杂度完成 cache 的 get 和 put 操作（put 包含了我们这里讨论的“替换”）。我们只需要利用双向链表和哈希表就能实现，哈希表指示在双向链表的位置，双向链表越靠近头部表示上一次使用时间越近；每当 cache 储存位置满了以后就删除尾部的节点（表示这是和上次使用时间间隔最久的一行）。每次命中一行之后需要及时把这个节点提到链表头部。LRU 替换策略的优点就在于，如果有热点数据，经常时不时就被引用，那么 LRU 策略能保留该数据，避免不命中。但是 LRU 策略下，高速缓存很容易被“污染”：可以试想一个场景，当要打开 WPS 继续写课程报告的时候，不小心点到了紧挨着的游戏，游戏很好玩，不由得玩了几分钟，但是在这几分钟内，之前保存在 cache 里的 WPS 文件已经按照 LRU 策略被替换殆尽。由此可见，LRU 策略在有较多重复引用的数据时是较好用的，且适用于干扰较少的场景。

再次是 LFU 策略，即最不常使用策略。LFU 策略同样可以从生活找到很多类似的例子，比如淘宝京东有很多的“人气榜”，当有新的人气商品出现，会优先替换掉排在榜单末尾的商品(购买次数较少的)。在力扣中同样有一题 460. LFU

缓存 - 力扣（LeetCode），通过这题我也简单地掌握了 LFU 策略的实现。只需要两个哈希表就能实现 o(1)时间内的操作；一个储存最近一段时间的使用频率，一个储存索引对应的在前一个哈希表内的地址。使用频率最低的成员也始终在哈希表尾，如果冲突了没空间，就需要删除表尾元素。LFU 策略是统计了近一段时间内的使用频率，较少用到的成员自然就被认为是不大可能再被调用的。但所谓的近一段时间，到底是多久，其实很值得斟酌；极端的情况就是从开机了就开始统计，那么如果前期的计数次数都比较多，堆积在那里了，新的元素进来就会不断被淘汰（A 存进来，次数积 1，下一次 A 最少又被淘汰）；另一个极端就是考察的时间很短，那可能有访问次数很多的数据可能会被淘汰。LFU 策略的优点在于效率会优于 LRU 策略，并且不会因为 LRU 策略而导致的内存污染；但也是因为 LFU 策略的缓存机制，LFU 的缺点之一在于如果切换访问对象，那么前者残留的次数信息会对后者造成影响，如果需要额外处理的话，又是其他的代价了。LFU 策略的另一个缺点就是，对访问次数的额外记录也是需要付出代价的，如果要用到排序的话耗费更大。在适用场景上，对于热点数据的频繁使用仍然是我们希望看到的；而在访问时，频繁更改访问对象、访问方式也是不被建议的（有别于 LRU 的易受到污染，这里的意思更倾向于如 A B C 三件事都是要做的，那么根据三者访问数据的交集（如 A 和 B 都用到了数组 x y（C 没有），而 B 和 C 又用到了数组 a b（A 没有），那么按 ABC 的顺序实现可能好些，而不是 A 做一点，B 做一点，C 做一点，最后又回到 A），等可能会找到较好的实现顺序。）

LRU 和 LFU 是有一定缺陷的，并且可以提升的，就在网上能找到的几种基于 LRU 和 LFU 策略拓展的策略进行讨论，力求能看到这些策略的改进之处、适用场景。LRU-k 策略，是对 LRU 策略的改进。主要是解决了上面的缓存污染的问题，只有当信息被访问了 k 次才会真正放入缓存，避免了因为偶发性的数据读写导致之后的不命中数上升。LRU-k 策略的好处是显然的，但是要达成这样的效果，需要足够多的储存空间，k 越大，所需的额外空间越大；并且要维护这样的额外空间，需要排序、需要计数，是较复杂的。LFRU 策略，是基于两者共同做出的改进。LFRU 策略将缓存分为 privileged partition 和 unprivileged partition，特权区域表示内容很受欢迎，是受保护的；在特权区域内采用 LRU 策略，在非特权区域采用 LFU 策略。在特权区域内，以时间为顺序对热点内容进行更替，符合实际；在非特权区域，只有当这段时间内的访问次数够多，表示这是新潮的、流行的，才能进入特权区域。这个策略适用于我们访问一些学习网站时，看到一个帖子好，反复观看，但忘记要收藏时，如果能即使推送到缓存的“特权区域”，那么下次打开也很流畅。

总结一下，在组相联和全相联高速缓存中，对行的替换是很重要的；选择合适的行替换可以提高命中率，能更快速地完成目标。常见的替换策略中：随机选取替换，实现简单，但不确定性大，不是好的选择；LRU 策略实现复杂一些，但 LRU 在有反复、间隔短地被引用时表现很好，但容易受到其他数据的污染；LFU 策略实现复杂，但可以避免 LRU 策略的数据污染，能对多次引用的数据达成很好的储存，也仍会因考察时间的长短、数据访问对象的影响；在 LRU 和 LFU 策略上有很多拓展策略，LRU-k 策略基于 LRU 策略，损失复杂度/硬件，避免缓存污染；LFRU 策略集两者之长，达成了分区、记忆、保护的效果，但实现也较为复杂。

四、写策略

在课本上，cache 的写策略根据命中与否，可分为直写/写回、写分配/非写分配；接下来会两两配对，按照 cache aside（在选题 2 中提到的）、直写/非写分配、写回/写分配的顺序简要介绍这三组的特点，重点阐述它们各自的优缺点和适用场景。

首先是 cache aside 策略。这时的写策略是先更新内存，然后将缓存中的有效位置 0。这个策略的优点在于，实现的思路简单，不需要区分为写命中和写不命中，并且统一将缓存数据失效，不用考虑更新的操作（不需要额外记录信息，只要删除）。这时的缺点是，频繁的写操作，导致缓存中的内容不断被删除，导致读的命中率很低。而且，读和写的时序安排上可能会导致缓存和内存中的数据不一致，导致错误的、无法预料到的缓存污染。从上面的叙述可以看出，cache aside 策略应尽量避免“写”操作，即在读操作占了大多数时，该策略能发挥较好地用处；并且最好在对缓存和内存数据一致性要求不高的情景下使用。

其次是直写和非写分配策略。直写即更新了高一层的缓存后，立即将该缓存块立即写回到紧接着的下一层中。而非写分配也类似，在写不命中之后，不管这层的 cache 了，而是直接向低一层发起修改，主存内容也不会被载入 cache。直写同时修改 cache 和主存，频繁访问主存，访问次数很多；任何一点风吹草动就动辄进行很大的开销。其缺点就在于此，如果我们按储存结构一层一层从高到低修改，时间开销太大（从 L1 到 L2，再到 L3，最后到主存），每次写都引起总线流量；如果我们要考虑多层并行，硬件设计的难度较大（而且因为主存的大小是比前几级 cache 大得多的，所以如果每层搜索速率一样，关键的约束仍是在主存修改；并行可能不会有很大的裨益）。但直写也是有好处的，直写保证了数据的一致性，从 cache 往下修改，对于这个固定的块的内容，因为每次查找从 cache 开始，一定找到的是最新内容。非写分配也是类似，牺牲效率保证一致性。而直写和非写分配的使用场景，即对于数据的准确率、一致性、实时性要求高的场景。如这次的课程报告，A 同学（看作 cache）将“ICS 课程报告.pdf”从 elearning 的众多文件（主存）拷了副本，但发现其中有错误；因为如果只有 A 同学改了错，其他同学是不知道的，所以应该保持数据的一致性和实时性，及时修改主存（elearning）上的版本。而直写之所以和非写分配搭配，也是因为在多次对同一个块的修改中，如果采用写分配，（第一次未命中）下一次会进入直写，要多一步对于 cache 的修改；而非写分配只要考虑主存，只对主存进行修改就可以了，这样操作更简单。

最后是写回和写分配。写回即尽可能避免对下一层 cache/主存的更改，先对现在这层 cache 进行信息的更改，当且仅当设定的替换策略要修改这行时，才会向下一层传递修改后的信息。写分配即写不命中时既改变主存又把该信息从主存传输到 cache 中，将 cache 中的一块替换为该信息。写回和写分配搭配也类似于直写和非写分配的意义；若写回采用非写分配，（第一次不命中后采取非写分配），下一次仍然会不命中，然后还是要到主存去查看；而若是采用写分配，下一次时就会在 cache 中命中，写回的“推迟更新”才能得到实施。写回的优点是显著的，只要修改的这个块没有被驱逐，无论写多少次都只要在 cache 操作，显著减少了开销、总线流量，是对局部性的很好利用；写回的缺点也是有的，如何标识该块是被修改的，与主存版本不同的，就需要额外的空间开销（修改位），但这点开销随着写命中的次数增大（即比起直写省下的开销变大）就变得很实惠，唯一吃亏的情况就是该块 cache 被标记为“修改过”之后，就被替换策略驱逐，往下层

更新。写分配，作为与写回搭配的策略，能很好的保证一次不写回（若不被驱逐），下一次就是写回；但在利用了写的空间局部性的同时，每次不命中从主存到 cache 的传递、更新，都需要额外的开销。在真实的 cache 机制中，cache 是多层的，若要从主存写到最快的 L1 cache 要经历多次的驱逐一个块、写入新的块；开销就更大了。基于写回和写分配的开销小、效率高，我们应该在写密集、写操作占大多数的场景使用写回和写分配；并且在查询数据时，要注意 cache 和主存数据的不一致性。

总结一下，cache 的写策略从实现的难易程度来说，cache aside 最为简单，其次是直写和非写分配，写回和写分配则困难很多。而从效率/开销角度来说，一般而言，写回和写分配是显著高效的，cache aside 和直写/非写分配相距甚远。而从 cache 和主存数据的一致性而言，cache aside（cache 失效）和直写/非写分配（cache 一致）保证了数据的一致性。

五、缓存友好的编程建议

评价对缓存是否友好，我们可以根据程序运行的平均未命中数来判断，未命中较少，则意味着对于缓存是更友好的。

根据前面几个部分的分析，如果我们对 cache 的大小、储存模式、替换策略、读写策略都有较好的了解，我们在写程序时（或者是后期修改时），就可以根据 cache 的各项机制进行程序的优化，以期达到更快的速度。

具体的建议如下：（1）从变量的数量来说，我们应该尽量重复地引用局部变量，这样局部变量会被储存到寄存器中，速度更高；而冗余的、重复的局部变量不是好的选择。（2）从循环角度来说，内循环是最重要的，是主要发生不命中的地方，我们要聚焦于循环内部，以此减少不命中数。（3）从空间的局部性来说，数据总是储存为连续的块，那么访问连续的数据是好的选择；这导致步长为 1 的引用模式是好的，很多时候循环更新的条件都是 i++，即步长为 1；而在程序优化的章节中我们提到了 2x1a 展开，虽然每次更新为 i+=2，但是一次循环会覆盖两个数据。

在书上有矩阵乘法的六个版本，不命中数最低的 kij/ikj 版本，比起其他版本，一是同样很好地利用了建议（1），多次重复使用一个局部变量；并且聚焦于内循环，充分利用了建议（3），利用空间局部性，每次都是在矩阵中一行一行（连续的内存地址）地读写数据……接下来会以一具体的例子再来例证这几项高速缓存友好的建议。

接下来以用 LU 分解求解矩阵的逆阵这一算法，再具体阐述（就是把上一段讲具体一些）这几条建议。因为矩阵是很好体现空间局部性和多层循环的结构，但又想找点新东西（不想再复述一遍矩阵的乘法），就选择了这个例子。首先是一个在 csdn 的链接：代码实现矩阵求逆的三种方式（超详细、已实现）_爱学习的杨子的博客-CSDN 博客_矩阵求逆代码。从这个链接的 LU 分解求解逆矩阵部分的代码出发，来说明。

首先简述一下 LU 分解的思路，将矩阵 A 化为矩阵 L*U，L 为下三角矩阵，U 为上三角矩阵；A 的逆就等于 U 的逆乘 L 的逆。上三角和下三角矩阵的逆就更容易求出。具体的代码分为三个部分：（1）构造 L 和 U；（2）求出 L 和 U 的逆；（3）将二者相乘。

从这个文章的代码第一部分开始。即求 U 逆阵这里：

```
for(i=1;i<N;i++)
{
    for(j=i;j<N;j++) // 求U
    {
        s = 0;
        for(k=0;k<i;k++)
        {
            s += L[i][k] * U[k][j];
        }
        U[i][j] = W[i][j] - s;
    }
```

这里可以看到，他的编写应该是出于好理解的角度，从 LU 分解的数学定义编写；我们可以发现，与矩阵乘法的 ijk/jik 版本有很大的相似性。因为每次计算都是考察 L 的一行（的前 i 个），和 U 的某一列（的前 i 个），连续考察步长为 1 的行是没有错的，因为是连续的内存地址，会被整块拿到 cache；但是对行的步长为 1 的考察是不合理的，特别是矩阵大小太大的时候，可能每次都不命中，都需要替换等操作。我们可以仿照书上的高效 ikj/kij 版本；不是一次性计算出最后的结果，而是分多次计算。大概的代码如下：

```
for(i = 0;i != N;i++)
    for(j = 0;j != i;j++){
        temp = L[i][j];
        for(int r = i;r != N;r++)
            U[i][r] -= temp * U[j][r];
    }
```

（这里 U 先与 W 等，指上三角部分，且非首行首列）
和矩阵的乘法类似，虽然这里没有用一个 s 来减少对内存的访问，但是通过对 cache 机制的利用，会让这个版本明显快得多。

当然，这里的问题比起矩阵的乘法会复杂一些，因为并不是遍历一整个矩阵。我们关注下最重要的内层循环，假设 N 足够大，并且 N 能够整除 cache 一个块的大小。在矩阵的乘法版本中，在内层循环的考察数是一致的，而在这里，随着 i 的增大而减小（一开始的一段时间内是不变的），到最后每次替换的元素并不都是有用的。比起具体地计算不命中数，有两点值得注意，一是当 N >> 块的大小，这种情况是占少数的；二是通过 C++自带的计时函数也发现，当 N 足够大，按照我们修改之后的版本运行仍然是显著更优的。

假设一种较完美的情况，cache 有 2 组，采用全相联，每组只有一个块，一个块可以容纳几百个 double 元素；数组 U 映射到组 0，数组 W 映射到组 1，交替填充即可。

对矩阵 L 的求解的修改也是类似的，不再赘述。

但这里有一个很重要的问题/误区，就是矩阵的乘法，我们由 A B 得 C，这里对于 U 的求解却是用到了它自身的，要确定这个结果是不是正确的，如果结果不是正确的，再 cache 友好也是没有意义的。我们再回顾一遍原题中的算法，按照行列的顺序求解问题，求解一个元素时，需要确保该元素在行/列方向上次序在前的元素已经是有序的。在我们改写的算法中，因为同样不是完全遍历整个矩阵，对 j 进行了限制，可以发现，每次调用 U[j][r]时，他一定是已经被计算出来的。

接下来是代码的第二部分，求上/下三角矩阵的逆。可以发现，上下三角矩阵求逆，步骤简单，类似于矩阵的乘法。最后是代码的第三部分，求两个矩阵的乘积。这里显然也没有采用最好策略，修改为课本上的 kij/ikj 版本即可。

总结一下，根据前面已有的建议，进行程序的修改是高效的；但所有程序的优化，对 cache 的适应，一定要建立在不会改变算法原意的基础上。算法可以慢一些，但一定要是正确的。而对 cache 友好的追求，值得我们花较多时间去研究和琢磨。

最后，从 cache 不同类型的高速缓存、替换策略、写策略的角度而言，这些是比较微观的，写程序时往往很难具体顾及；但是我们只要对这些要点有一定的了解，就能理解课本上为什么会给出那样的高速缓存友好的建议，并且在涉及到内存读写的时候充分利用 cache 的特点。

至此，课程报告的内容已经结束，下一页有一些对本学期课程的感想。

对本学期课程的感想

（1）首先先说一下我自己的情况，上了大学才接触计算机相关的东西；所以没有什么前置经验;在暑假预习的时候并没有 get 这门课的量大和重要的特点，也没有提前预习（翻了翻，感觉书好大本，遂止）。

（2）首先这门课，和其他专业课从学习的模式上来说，都是很不一样的。没有很多的笔头作业（写课后习题这类的），然后就是大部分时间在听老师讲。lab 作业总体上来说，还是感觉好难……（可能最后一次那种查资料的比较简单些）。对我来说，可能 lab4 是最简单的、lab2 其次、然后是 lab1，最后是 lab3（太难了）。回归正题，lab 作业还是和课上的知识相关联的，但很多是延申了不少的；做完确实很有成就感，也学到了很多新的知识。TA 也很负责，虽然我很少去问 TA 问题，但是在有限的几次问答中能感觉到。

（3）因为正课放在周五下午，所以真的就，不知不觉就困了。老师讲得其实真挺好，并不是很枯燥的；但是正如（2）中提到的，没有笔头作业，就上课很懒散，偶尔听一听，听也听不认真等现象层出不穷（仅针对我）。然后课后也就把优先级放得很低，有几次是，已经开到下一章了还是很懵，自己感觉不行了，一定要先看懂之前的，就在周五上午“复习”了一遍；或者是 lab 布置下来几天了，深感一定要开始努力了，然后去看一遍书。在此，对金老师表示非常非常的抱歉；所幸课本真的很详细，能自己看懂。

（4）关于课程报告：我原先是对课程报告这种形式存疑的，怕再给我来一个 lab5 或者 pj plus，但现在我感觉是很好的决定。根据有 lab，课本内容就掌握得比较仔细的规律，程序优化、存储结构这两方面也是我之前掌握地比较水的，这次为了课程报告，我能较有时间地把这俩部分好好过了一遍，最终写下课程报告，收获不仅仅在于这次的课程报告这些内容。更不用提，在十七、十八周我阳了，所以如果是闭卷考试，估计现在还在抱头痛哭哈哈。