CacheLab 报告

软 73 沈冠霖 2017013569

1.PartA

可以把代码分成三部分：第一部分是 Cache，Set，Block 的定义，构造和析构；第二部分是指令拆分和处理；第三部分是 main 函数中的读入指令和文件。

第一部分包括类型 Block（存储一个 Block 是否有效，其 Tag 和其最近使用时间），Set（包含一个 Block 类型数组）和 Cache（包含一个 Set 类型数组），以及这些类型的构造，析构。在读入完毕 s,E,b 后，程序会自动递归构造一个完整的缓存。

第二部分是整个程序的核心。包括函数 Find_Set,Find_Block,Compare_Tag,OperateCache,OperateInstruction.读入一条指令后，OperateInstruction 函数会拆分这个指令的地址，得到 t，s，b。然后会调用 Find_Set 在缓存里找到其对应的 Set，之后会调用 Find_Block 来找到其对应的 Block。

Find_Block 函数的逻辑可以说是这一部分的核心：程序遍历所有的 Block，并且调用 Compare_Tag 来判断是否命中，同时，我会记录当前遇到的第一个 invalid 的 Block，和遇到的上次调用最早的 Block。遍历完毕之后，我就能得到这次访问的三种可能情况了：命中，未命中但不替换，命中且替换。并且，我也能找到对应的 Block 地址（命中就是命中的 Block，未命中就是第一个遇到的 invalid 的 Block，命中且替换就是待替换的 Block）。我会把情况和地址传递给函数 OperateCache。如果未命中或者要替换，就修改 Block 的 valid 或者 Tag。否则只需要修改访问时间。因为这个模拟器输入的地址并不是真实的内存地址，所以并不需要真的修改缓存的值，只需要修改 Tag，Valid 和访问时间就可以了。

第三部分包括命令行指令读取和输入文件。命令行指令读取需要用到 getopt 函数，我参考了这个函数的百度百科。输入文件则需要读取指令类型，地址和长度。需要注意，地址是 16 进制的数，需要用%llx 读取。值得一提的是，程序中许多地方需要用到地址，在对地址进行拆分，比较（主要用位运算），输入的时候，我定义地址为 long long int 类型。在需要访问地址的时候，我定义地址是 bool*类型。这两者可以强制类型转换，因为都是 64 位。

2.PartB

2.1 核心思想

缓存的 s=5，E=1，b=5，这意味着一个 Block 里能存 8 个 int，连续的 32 个 block 互不冲突，而每隔 32 个 block 就会发生冲突，以及只要冲突就必然替换。为了最大限度利用 Block，我这个程序的核心思路就是一下子读取 8 个数，然后对这 8 个数进行写入。

2.2 32*32

这类数组每行有 4 个连续的 block，而且每隔 8 列的对应位置数组会冲突。我的设计思路就是以一个 8*8 矩阵为基本单位，每次读取 A 矩阵某个基本单位的一行，也就是连续的 8 个数，然后写到 B 矩阵中对应的一列，然后再读这一个基本单位的下一行，以此类推。示意图如下。

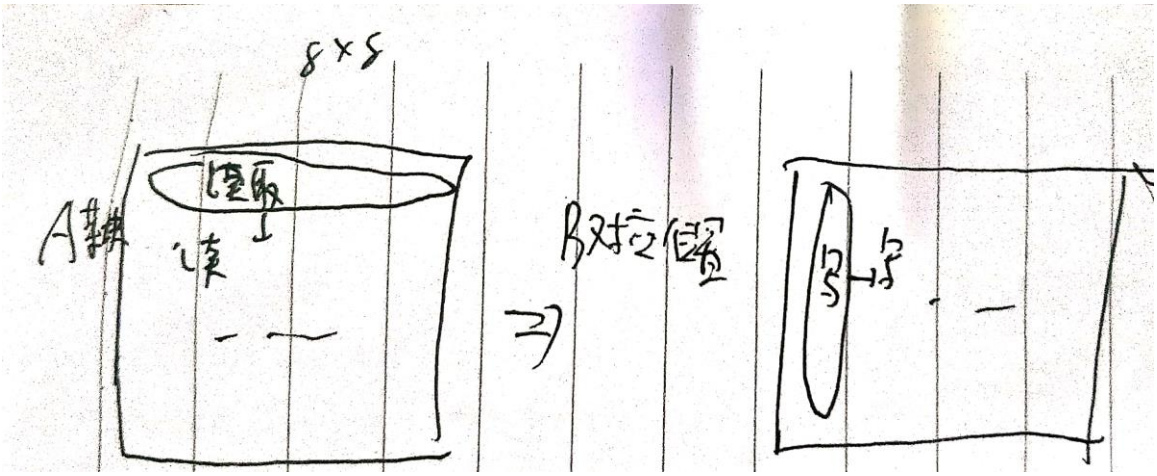

图 1. 32*32 矩阵的思路示意图

估计一下不命中次数：每读取 A 矩阵的 8 个数，只会有一次不命中。而写到 B 矩阵对应位置需要分两种情况。

情况 1：这个基本单位不在矩阵 A 对角线上，这样的话 A 读取的每一行和 B 写入位置的某一行不会有冲突的可能（地址的 set 不同），所以，其不命中次数如下：每次读入一行会产生一次不命中，写入的话，第一次会有 8 个不命中，之后就会全部命中。这样，每操作一个基本单位会产生 16 次不命中。

情况 2：这个基本单位在矩阵 A 对角线上（比如第一个子矩阵），这样的话 A 和 B 对应行的 set 和 block 值一致，tag 不同，是冲突的。这样，每次读入一行会有一次不命中，写入第一次有 8 个不命中，之后每写入一次，因为之前读入一行会修改一个缓存地址，因此会产生 1 个不命中。这样，操作一个基本单位会产生 23 次不命中。

综上，这种思路会产生 23*4+16*12=284 次不命中，而实验结果是 287 次不命中，大致相当。

2.3 61*67 和 48*48

这两种情况较为复杂，因为其两个互相冲突的地址间隔的行不是整数个。我的思路同 2.2，每次处理 8*8 的子矩阵，读入一行写一列，只是在 61*67 矩阵要加入边界判定语句。

这两种复杂情况和 32*32 比较，其缺点有两点：首先，某次写入之后，下一次写入的位置不一定在缓存内，不一定能很好利用之前写入的缓存红利。其次，对于 61*67 矩阵，连续读入的 8 个字符不一定都在一个 Block，可能在 2 个 Block 中。

但是相比 64*64 矩阵，采用这种方法也有两点优势：首先，缓存替换现象不那么明显。64*64 矩阵连续写入 8 个数据的话，缓存会被替换一次，下次写入还是全 miss。而这两种矩阵每隔几整列的数据很可能有不同的 set 值，不一定会发生冲突，下次写入可能还能利用上次的缓存。对于 48*48 矩阵，每隔 5+1/3 行才会发生冲突。61*67 矩阵则因为太不规则，每隔几整行元素出现冲突的概率较小。这意味着我写入某一列元素后，再写入下一列时，之前的缓存难以被替换。这就让我能实现和 32*32 类似的优化效果。其次，对于 61*67 的矩阵，矩阵 A 和 B 在对角线位置的子矩阵不再互相冲突了。

采用这种简单的思路效果还不错。对于 61*67 的矩阵，miss 略低于 2000 次，因为 miss 要求不是很严，因此也能达到要求。对于 48*48 的矩阵，miss 是 600 多次。鉴于 32*32 的矩阵 miss 是 287 次，而 48*48 的矩阵更加不规则，元素也是 32*32 的 2.25 倍，因此出现 600 到 700 次的 miss 可以认为优化效果还不错。

2.4 64*64

对于 64*64,因为每隔四列的元素是冲突的，问题变得复杂了。如果直接按照 2.2 的步骤，每写入一列，后四个元素会替换前四个元素所在的缓存，导致再写入下一列还会出现全部 miss 和替换，大大降低效率。因此，得修改这个方法。

我的修改思路如下：为了表示方便，令 B 矩阵中某个 8*8 子矩阵等于四个 4*4 子矩阵（B11,B12,B21,B22）的分块。首先，对于读入的前四行 32 个数据，我把每行前半部分正常存到 B 的对应位置 B11，然后把后半部分存到 B12 位置。因为 B11 和 B12 都在同样的 4 个 Block 里，所以这种操作写入的时候只 miss 四次。

其次，对于读入的后四行数据，我需要修改 B12 位置，提取 B12 位置的值来修改 B21，然后正确赋值 B22。我的方法是按列读取后四行数据。举个例子，我先读第一列，然后用读到的四个数据和 B12 第一行的四个数据交换。此时，B12 第一行的四个数据对应的应该是 B21 第一行的四个数据。之后，我再读取第五列作为后四个数。这样我就得到了 8 个数，他们分别对应 B21 和 B22 的第一列。之后我去读取第二列，替换 B12 第二行，以此类推。这个方法的核心思路在于，第一，无论怎么读取数据，miss 数都是 4 不变。其次，按行写入可以大大降低 miss 数。因为每隔四行冲突，因此我就先修改完第 i 行，再修改第 i+4 行，以此类推，这样就能减少 miss 数。具体的步骤如下图。

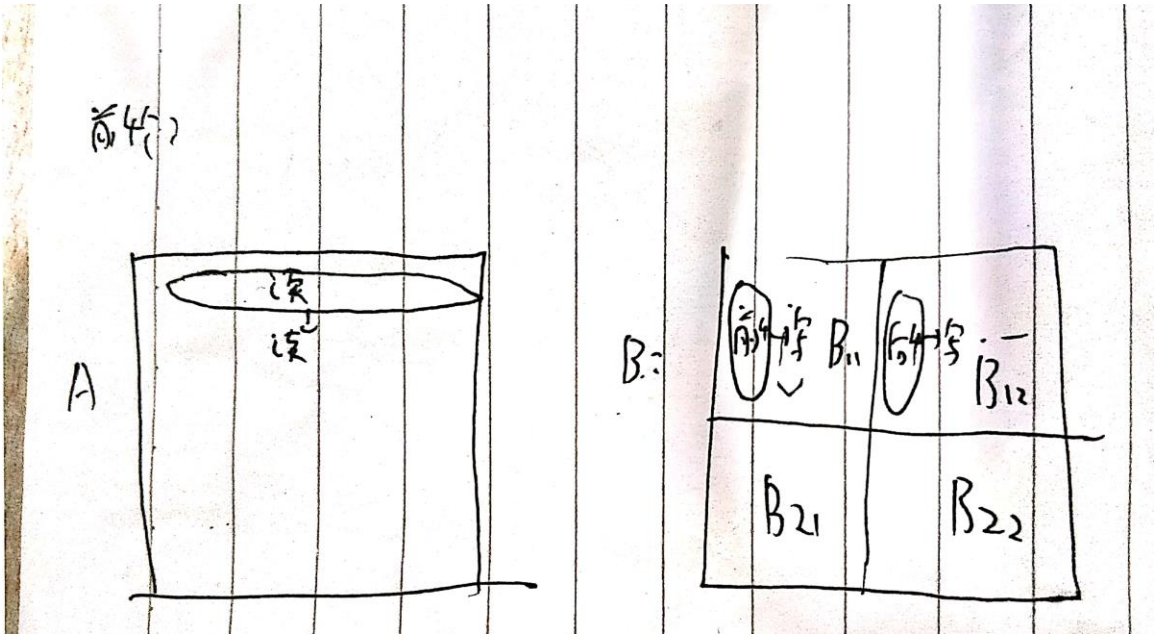

图 2. 64*64 矩阵读入前四行的示意图

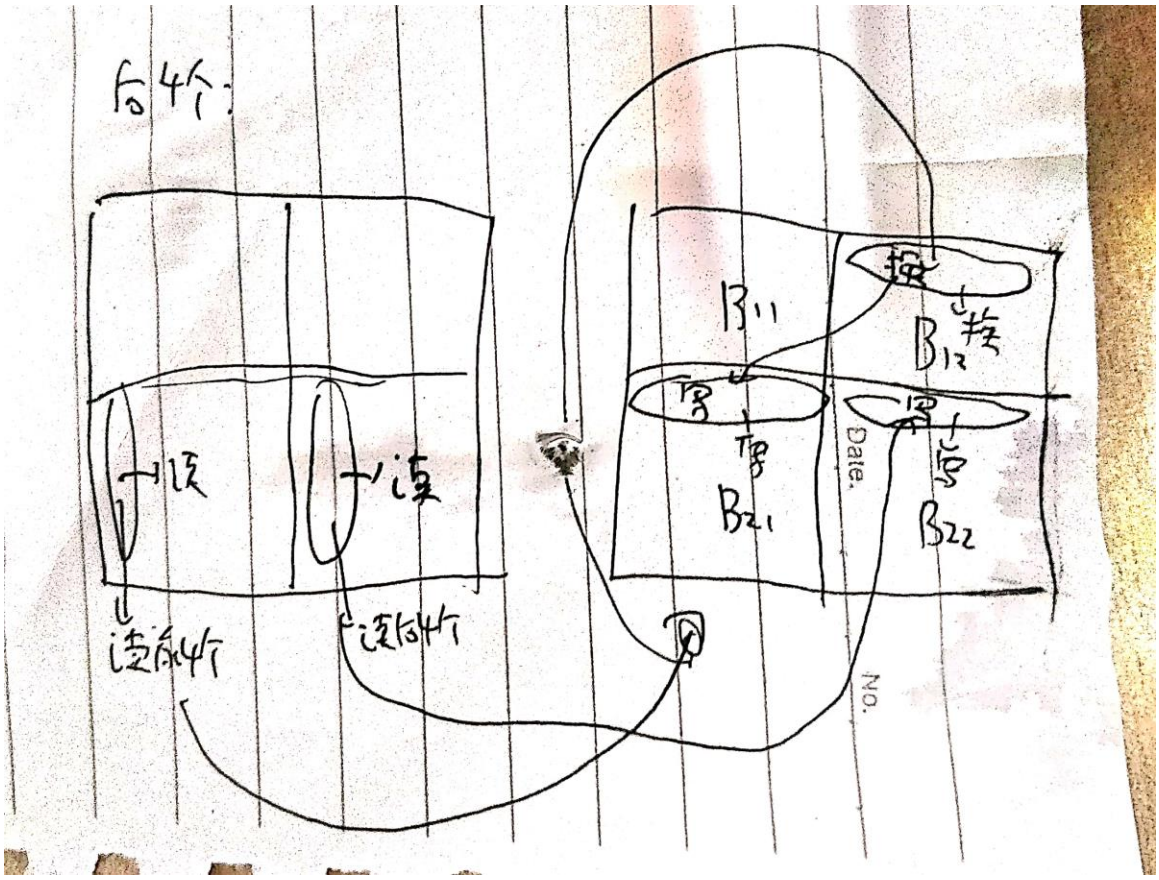

图 3. 64*64 矩阵读入后四行的示意图

经过分析，假设这个 8*8 块不在对角线上，我读取数据只 miss8 次，写入每一行都 miss1 次，一共 miss16 次。如果在对角线上，读取数据还是只 miss8 次，后四次写入各 miss 一次，而即使前四次每次写入都 miss，也就 miss32 次。最多 miss44 次。这样 miss 最多就是 16*56+44*8=1248 次，小于要求的 1300 次。测试结果也是 1200 多次，和我估计的比较接近。