Cache Analysis

实验要求

用 C/C++ 编写一个 Cache 模拟器，输入访存 trace，输出统计信息和 Hit/Miss 的记录。

实现方法

运行方式

采用 C++11 实现了实验要求，用 CMake 进行构建管理。编译方法：

```
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ make
```

然后调用 `./cache` 获取一个 trace 对应的结果：

```
$ ./cache ../trace/astar.trace
```

它会多线程计算各种情况（Block 大小、相连度、替换算法、写策略）下 Cache 的行为，然后写入到和 trace 统一目录的若干文件中，文件名格式如下：

```
trace_${BLOCKSIZE}_${ALGO}_${HIT}_${MISS}_${ASSOC}.{info,trace}
BLOCKSIZE：块大小，有 8 32 64 三种
ALGO：替换算法，0代表LRU，1代表随机，2代表二叉树（PLRU）
HIT：写命中策略，0代表Writethrough，1代表Writeback
MISS：写缺失策略，0代表Write-Allocate，1代表Write-Nonallocate
ASSOC：相连度，有全相连，直接映射，4-way和8-way
info：描述了Cache的一些参数和信息
trace：描述了Cache的Hit/Miss历史
```

提交的 `trace` 目录里附带了若干个以 `_8_0_1_0_8.{info,trace}` 结尾的文件，即为实验所需要的块大小为 8B，8-way组相联，LRU策略，写分配+写回情况下各个重点 trace 的访问历史。Info 文件内记录了参数、元数据的大小和缺失率等信息。

代码组织

代码主要分为两部分，一个是 bit vector 的实现，另一个就是 cache 的实现。

Bit Vector

由于实验要求用尽量少的存储空间和位运算，所以编写了一个简单的 BitVec，通过位运算来取出中间的某些位，或者写入中间的某些位。为了保证正确性，也是编写了一个简单的 fuzz 测试，把 BitVec 输出的结果与 `std::vector<bool>` 对比。

实现的时候尽量模仿硬件描述语言，提供一个 [from:to] 的范围读和范围写的 API。虽然实际在内存中的空间可能超过所需要的位数，但通过 assert 断言保证了实际使用的位数不会超出预选设定的元数据的范围。采用 uint64_t 数组作为实际存储仅为了运行速度的优化，改为基于 uint8_t 的数组不难。

Cache

Cache 代码都在 `cache.{h,cpp}` 文件中，有非常多的代码注释，对于数据结构和组织和代码的功能都有具体的解释。

数据结构

首先是 CacheLine ，它保存了三个信息：dirty valid 和 tag，存储在一个 BitVec 中，最低位是 dirty，次低位是 valid，其余都是 tag。通过 BitVec 的相关函数进行操作。

然后是 LRUState，它保存了 LRU 的信息，是一个 n * log2(n) 位的 BitVec，其中 n 是相连度。从低到高每 log2(n) 位为一个元素，代表组中的一个位置，每次取最高的 log2(n) 作为被替换的位置，当访问一个 valid 的块的时候，则把这块挪到最前面。初始情况下，当还有剩余的块没有占用的时候，应当从这些块里面选择一个来填入，这里的实现方法是预先按照倒序插入到 LRUState 中，这样每次从最高位取的时候自然就是取得编号最低的未占用的块。

最后一个单独的数据结构是 PLRUState，保存了二叉树替换算法（我还是更习惯叫它 Pseudo LRU）。最低位保存的是是否所有的块都是 valid，如果没有valid则需要找到一个未占用的；之后则是一个二叉树，对于下标为 k 的结点，它的左子树下标是 `2*k` ，右子树下标是 `2*k+1` ，这样一共 PLRUState 的大小就等于相连度。

Trace 解析

编写了一个单独的函数 `readTrace` 从文件里解析，把文本格式转换为 Trace 结构题。不用多说。

读写处理

读入 Trace 以后，按照每条 trace 是读是写进行分别处理。

如果是读操作：

1. 先按照地址找到对应的组
2. 在组里面寻找有没有 tag 匹配并且 valid 的块
3. 如果有，就是一个 Hit，然后更新数据结构

4. 如果没有，就是一个 Miss，按照算法寻找一个 victim
5. 更新 tag 为当前这次访问

如果是写操作：

1. 先按照地址找到对应的组
2. 在组里面寻找有没有 tag 匹配并且 valid 的块
3. 如果有，就是一个 Hit，因为不涉及数据，无论是 Writeback 还是 Writethrough ，和读操作一样进行更新。
4. 如果没有，就是一个 Miss，按照 Write Miss Policy 进行处理：如果 No allocate，就没有后续的操作了；如果 Write Allocate，那么先进行一次读操作，再处理 dirty。

在这个过程中，只有两个地方涉及到了替换算法：1. Hit 的情况下状态的更新 2. victim 的选取。只要在这两个地方进行判断即可。

Cache 分析

在得到各个布局、策略和算法下 Cache 的缺失率后，可以得到一系列的数据。

不同的 Cache 布局

这一部分只考虑不同的 Cache 布局的情况，替换策略选择 LRU，写策略选择写分配+写回。通过程序输出的数据（ `*.trace_*_0_1_0_*.info` ），得到如下表格：

缺失率	CacheLine 元数据大小（B）/LRU 元数据大小（B）	astar	bzip2	mcf	perlbench
8B 直接映射	802816/0	23.40%	2.06%	4.94%	3.67%
8B 全相联	1032192/229376	23.26%	1.22%	4.58%	1.75%
8B 4-way 组相联	835584/32768	23.28%	1.22%	4.58%	2.07%
8B 8-way 组相联	851968/49152	23.28%	1.22%	4.58%	1.79%
32B 直接映射	200704/0	9.84%	1.33%	2.20%	2.31%
32B 全相联	249856/49152	9.59%	0.31%	1.82%	0.66%
32B 4-way 组相联	208896/8192	9.63%	0.31%	1.82%	1.14%
32B 8-way 组相联	212992/12288	9.63%	0.31%	1.82%	0.82%

64B 直接映射	100352/0	5.27%	1.59%	1.46%	1.89%
64B 全相联	122880/22528	4.97%	0.15%	1.08%	0.39%
64B 4-way 组相联	104448/4096	5.01%	0.15%	1.08%	0.85%
64B 8-way 组相联	106496/6144	5.00%	0.15%	1.08%	0.62%

由于 LRU 在直接映射的情况下退化，所以采用的数据是 `*.trace_*_1_1_0_1.info` 。

从以上表格可以得到以下初步的结论：

1. 其他变量不变的情况下，元数据大小和相联度正相关。
2. 其他变量不变的情况下，元数据大小和块大小负相关。
3. 其他变量不变的情况下，一般来说缺失率和相联度负相关。
4. 其他变量不变的情况下，一般来说缺失率和块大小在一定范围内负相关，一定范围内正相关。
5. 对于同一个布局，不同 trace 的缺失率差异可能很大。

这和课本上的数据是一致的。

不同的 Cache 替换算法

这一部分只考虑不同的替换算法：LRU、Random 和 PLRU。固定块大小为 8B，8-way 组相联和写策略（写分配+写回）。通过程序输出的数据 `*.trace_8_*_1_0_8.info` 得到如下表格：

缺失率	替换算法元数据大小（B）	astar	bzip2	gcc	mcf	perlbench	swim	twolf
LRU	49152	23.28%	1.22%	4.10%	4.58%	1.79%	6.54%	1.14%
Random	0	23.22%	1.22%	4.11%	4.60%	1.79%	6.57%	1.14%
PLRU	16384	23.29%	1.22%	4.10%	4.58%	1.78%	6.54%	1.14%

从以上表格可以得到以下初步的结论：

1. 从元数据的占用来说，Random < PLRU < LRU。
2. 从缺失率来看，三种替换算法的效果都差不多，各在一些 trace 上有较好的效果。
3. 如果要进一步降低缺失率，可能需要使用更先进的替换算法。

替换时所执行的动作的差异：

1. LRU：把被替换的项挪到开头
2. Random：没有额外的开销
3. PLRU：修改从根到被替换的项路径上的 bit

不同的写策略

这一部分只考虑不同的写策略。固定块大小为 8B，8-way 组相联和LRU替换算法。通过程序输出的数据 `*.trace_8_0_*_*_8.info` 得到如下表格：

缺失率	astar	bzip2	gcc	mcf	perlbench	swim	twolf
写不分配+写直达	34.50%	8.67%	8.67%	11.15%	4.66%	9.61%	1.45%
写分配+写回	23.28%	1.22%	4.10%	4.58%	1.79%	6.57%	1.14%
写不分配+写回	34.50%	8.67%	8.67%	11.15%	4.66%	9.61%	1.45%
写分配+写直达	23.28%	1.22%	4.10%	4.58%	1.79%	6.54%	1.14%

从以上表格可以得到以下初步的结论：

1. 写回还是写直达不影响缺失率，因为它们都是命中时的写策略。
2. 写分配比写不分配有比较显著的优势。

对比写回与写直达：

1. 写回：需要更新 Cache 中数据的内容就可以让 cpu 继续。如果需要保证外设看到内存的更新，需要缓存 flush 支持。
2. 写直达：同时写到内存和 Cache 中，性能较差，因为内存延迟较高。但外设可以直接看到新的数据。

对比写分配与写部分配：

1. 写分配：保证在连续写一段内存的情况下较高的缓存命中率。
2. 写不分配：延迟较高，但延迟稳定。

对于 Cache 设计的建议：1. 使用写分配+写回的组合 2. 若要采用写直达，一般需要实现写合并。

实验小结

实现了一个 Cache 模拟器，针对不同的参数和 trace 得到了一系列的数据和 log。采用控制变量法针对不同的参数，分析了参数对缺失率的影响，与课件上的结论基本符合。

代码实现上符合实验要求，采用位运算并尽量压缩了存储空间，模仿了硬件的实现方法。