实验四 构建 cache 模拟器

实验时间：5月16日至6月9日

实验人员：杨乙 21307130076

指导老师：王雪平

实验目的

实验目的：

- 完成 cache 模拟器
- 理解 cache 块大小对 cache 性能的影响
- 理解 cache 关联性对 cache 性能的影响
- 理解 cache 总大小对 cache 性能的影响
- 理解 cache 替换策略对 cache 性能的影响
- 理解 cache 写回策略对 cache 性能的影响

实验过程

通过读取配置文件获取 cache 模拟器参数并构建模拟器，对跟踪文件中的指令逐条读取和处理，记录命中率、运行时间等结果并输出到输出文件或标准输出。下面按照一定的步骤来构建 cache 模拟器。

准备工作与初始化

先从组相联映射进行分析。组相联的 cache 可以看作一个二维数组，行数目为组数，列数目为关联度。而直接映射可以看作每组只有一个 cache 行，即行数目为 cache 行数，列数目为 1 的二维数组；全相联映射可以看作只有一组，即行数目为 1，列数目为 cache 行数的二维数组。**综上，无论关联性如何，都可以把 cache 抽象为一个二维数组，**这个数组的每个元素代表了一个 cache 行，每个 cache 行需要包含如下内容：

- tag：标记，主存中的组群编号
- V：有效位，区分 cache 行是否已经装入
- cnt：计数器，进行 LRU 替换时进行计数

因为本实验不需要模拟主存到 cache 载入数据、块内寻址等细节，因此不需要申请额外的空间存放数据。`CacheLine` 结构体和初始化函数 `Initialization` 具体实现见代码及注释。

初始化函数申请空间时需要用到组数目 `groupSum` 和关联度 `associativity` 两个参数。同时，我们还需要根据主存地址 `addr` 获取某一条指令的主存地址对应的组号 `groupNum` 和块标记 `memTag`，具体实现见代码及注释。

模拟器核心原理

将 cache 模拟器的核心原理封装在函数 `CacheWorkLRU` 和 `CacheWorkRand` 中。前者采用 LRU 替换策略，后者采用随机替换策略。因为随机替换策略的原理较为简单（用 `rand()` 函数即可实现），我们在此仅对 `CacheWorkLRU` 函数的实现过程进行分析。

在函数中首先通过遍历查看是否命中，命中则更新各个引用值。同时，根据 LRU 替换策略的计数器变化规则，命中时被访问行的计数器清 0，组内比其低的计数器加 1，其余不变。若未命中，需要在组内按顺序查看是否有空行（保持顺序查找，则空行一定出现在所有非空行之后）。有空行则直接装入主存块，无空行则进行淘汰。

根据 LRU 替换策略的计数器变化规则，未命中且该组有空行时，新装入行的计数器设为 0，其余全加 1；未命中且该组无空行时，计数值最大的那一行的主存块被淘汰，新装入行的计数器设为 0，其余全加 1。注意到，无论空行是否存在，除装入行以外其余所有计数器都要加 1。根据上述规则，以及空行一定出现在所有非空行之后的判断，我们可以仅用一次遍历就完成所有工作。在遍历过程中，当标记位为 1 时将计数位加 1，并且记录计数值为 `associativity - 1` 那一行（若无空行时的替换行）的索引。在替换后再将新装入行的计数器置 0 即可。具体实现见代码及注释。

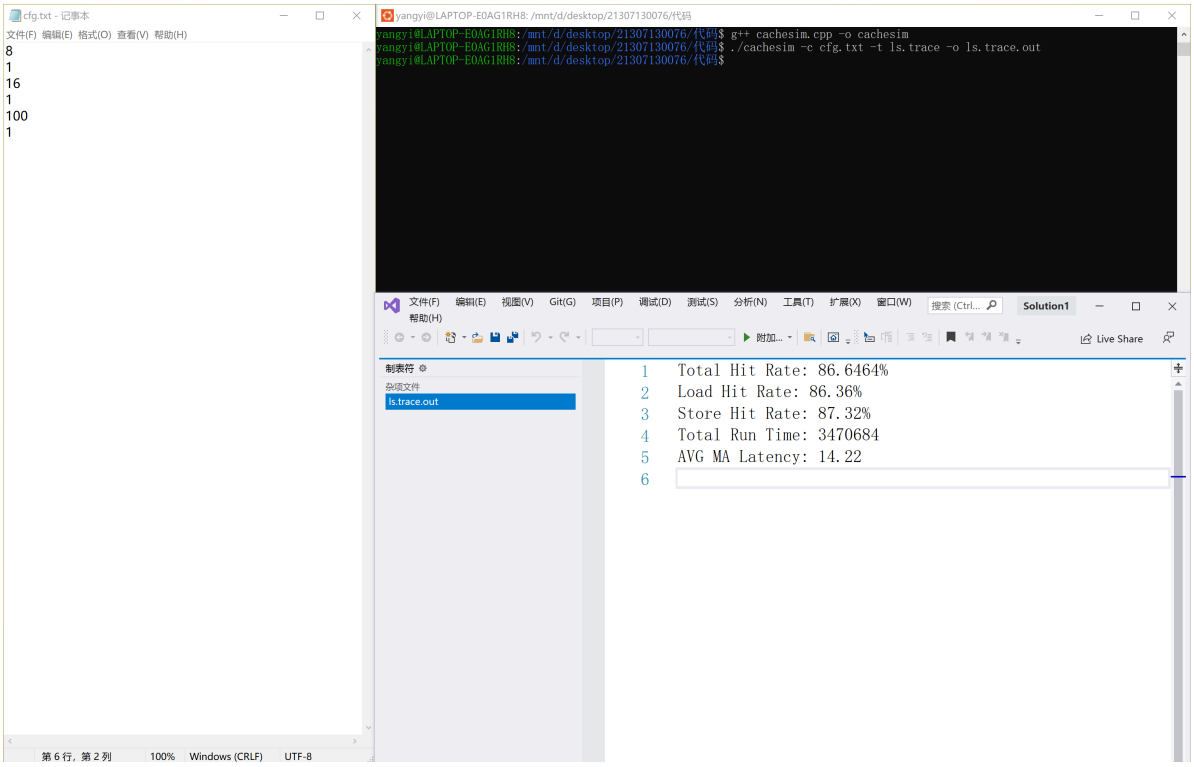
还需要注意的一点是何时需要更新 cache。除了 `writeMethod` 值为 0 时为非写入分配，不需要更新 cache，其余情况都需要。因此更新 cache 的条件为 `readorWrite == 'R' || (readorWrite == 'W' && writeMethod == 0)`

另外，在主函数中还需要进行传递参数、读文件、更新总指令条数 `sum`、更新读指令条数 `loadSum`、更新写指令条数 `storeSum` 等操作。具体实现原理较简单，可见代码及注释，在此不作分析。

至此，我们完成了 cache 模拟器的构建。

典型输入 / 输出图片

在命令行输入 `g++ cachesim.cpp -o cachesim` 编译代码，再输入 `./cachesim -c cfg.txt -t ls.trace -o ls.trace.out` 运行程序。典型的输入和输出如下：



实验结论

修改配置文件参数，使用命令行运行程序，观察输出内容，分析各参数对 cache 性能的影响：

1 cache 块大小和数据大小

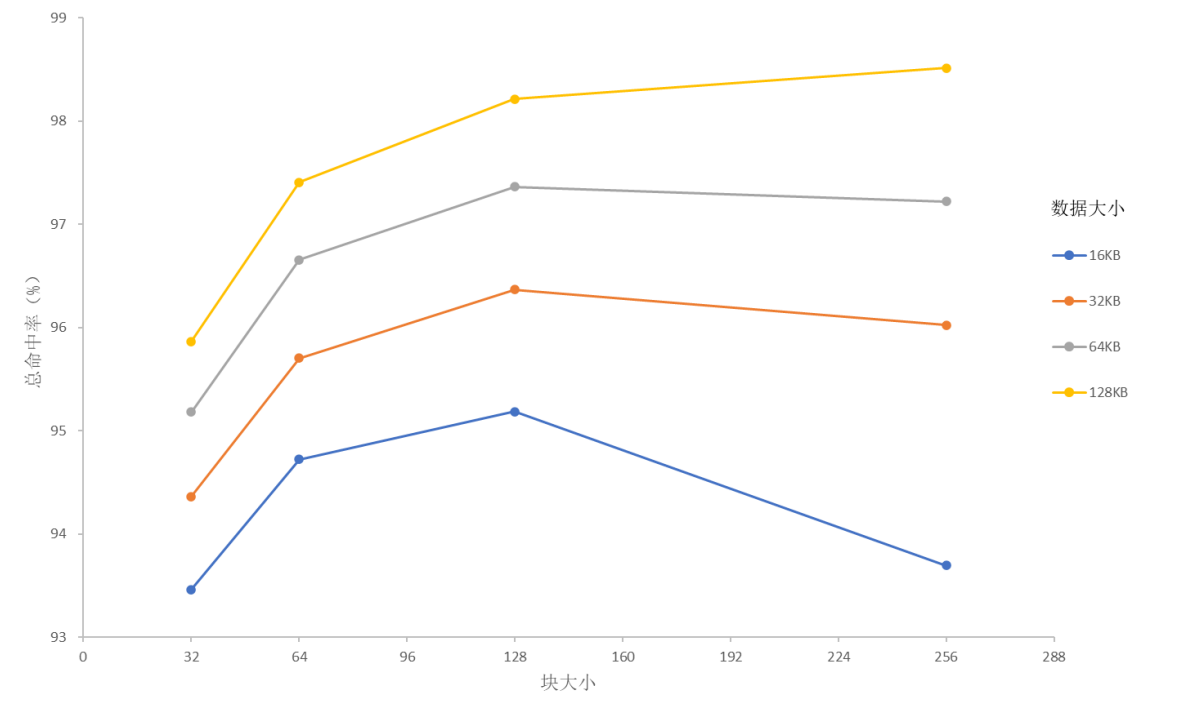
块大小和数据大小变化，其他参数固定如下：

关联性	替换政策	非命中开销	写入分配
直接映射	LRU替换	100	写入分配

记录块大小和 gcc.trace.out 文件中输出内容如下（仅给出三组）：

块大小	数据大小	总命中率	读命中率	写命中率	总周期	平均周期
32B	16KB	93.4648%	92.83%	94.94%	1823126	7.47
64B	16KB	94.7254%	93.91%	96.62%	1518503	6.22
64B	32KB	95.7026%	95.17%	96.94%	1282388	5.25

更改块大小和数据大小，得到更多组数据。绘制 **块大小和数据大小 - 总命中率** 性能对比图如下：



1. 根据输出结果可见，随着块大小的增加，总命中率呈现出先上升再下降的趋势

   先上升的原因：随着块大小的增加，程序空间局部性起主要作用，同一块中数据的利用率比较高，因此 cache 命中率开始时升高

后下降的原因：块变得过大的话，会减少 cache 总行数。而且也会使得离所访问的位置较远的块被再次使用的概率变小。因此，这种增加趋势在某一个最佳大小处使命中率达到最大值。这一点以后，命中率随着块大小的增加反而减小

事实上，**随着 cache 块的增大，存取时间也相对变长（这一点没有在参数中修改）**，因此非命中开销也会增大。综上，选取适中大小的 cache 块有利于性能的提高。

2. 随着数据大小的增加，总命中率升高

   显然 cache 总大小越大，命中率越高

2 cache 关联性

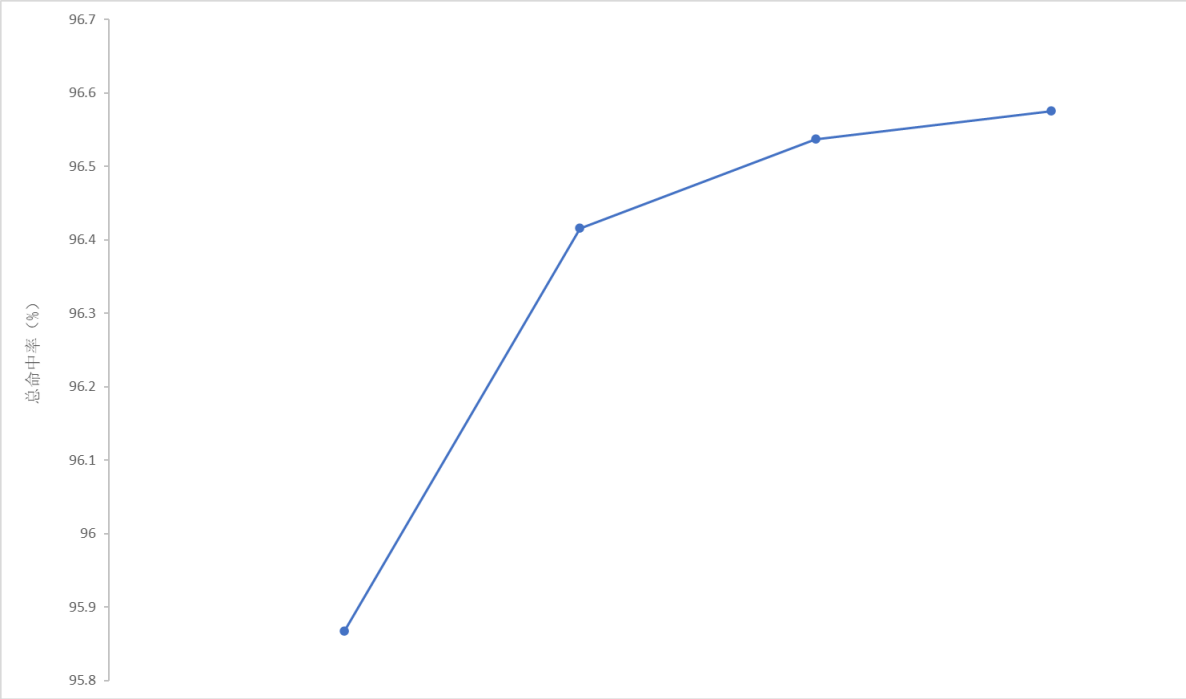
关联性变化，其他参数固定如下：

块大小	数据大小	替换政策	非命中开销	写入分配
32B	128KB	LRU替换	100	写入分配

记录关联性和 gcc.trace.out 文件中输出内容如下：

关联性	总命中率	读命中率	写命中率	总周期	平均周期
直接映射	95.8677%	95.94%	95.69%	1242491	5.09
2路组相联	96.4155%	96.36%	96.54%	1110128	4.55
4路组相联	96.5368%	96.51%	96.61%	1080824	4.43
8路组相联	96.5753%	96.55%	96.62%	1071518	4.39

绘制 **cache 关联性 - 总命中率** 性能对比图如下（从左到右依次为直接映射、2路组相联、4路组相联、8路组相联）：



由输出结果可见，关联度越高，命中率越高。但事实上，**关联度越高，判断是否命中的开销就越大，命中时间越长。同时标记所占额外空间的开销就越大。**本实验因为统一将命中周期设置为 1，未能体现这一变化。综上，选取适当大小的关联度有利于性能的提高。

4 cache 替换策略

cache 替换策略不同，其他参数固定如下：

块大小	数据大小	关联性	非命中开销	写入分配
8B	128KB	4路组相联	100	写入分配

记录 cache 替换策略和 gcc.trace.out 文件中输出内容如下：

cache 替换策略	总命中率	读命中率	写命中率	总周期	平均周期
LRU 替换	89.3554%	89.30%	89.49%	2816096	8.27
随机替换	89.2268%	89.18%	89.33%	2847182	11.54

LRU 算法可以正确地反映程序的访问局部性，因为当前最少使用的块一般来说也是将来最少被访问的。因此 LRU 算法的命中率高于随机替换算法

5 cache 写回策略

cache 写回策略不同，其他参数固定如下：

块大小	数据大小	关联性	替换政策	非命中开销
32B	128KB	直接映射	LRU替换	100

记录 cache 写回策略和 gcc.trace.out 文件中输出内容如下：

cache 写回策略	总命中率	读命中率	写命中率	总周期	平均周期
写入分配	95.8677%	95.94%	95.69%	1242491	5.09
非写入分配	92.4331%	95.25%	85.88%	2072408	8.49

非写入分配策略仅更新主存单元但不装入 cache 行，未能很好地利用空间局部性，因此命中率更低。**但这种做法可以减少读入主存块的时间。**这一点在参数设置过程中未能体现。

实验感想

1.根据各个参数性质，适当调整 cache 参数，可以提高 cache 性能

2.cache 的性能受命中率、命中周期、读入主存块时间等综合因素的影响