Cache

陈嘉杰 2021310745

本实验代码与本科生课程比较接近，因此我在我自己本科时编写的代码基础上进行修改，得到了目前的版本。

编译和运行

代码采用 CMake 构建系统，编译和运行只需要执行 `run.sh`，脚本会完成 cmake 配置，编译，并运行各个 trace 对应的输出。

设计思路

Cache 部分代码主要在 `cache.cpp` 和 `cache.h` 中，定义了一系列的 cache 参数：总大小、块大小、相连度等等，并且针对不同的算法进行了实现。读取 trace 以后，按照下面的逻辑进行处理：

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。

在本实验中，没有对 Write Hit Policy 和 Write Miss Policy 进行要求，因此下面的所有实验都采用了 Write Back + Write Allocate 的方式。

LRU 替换算法

LRU 替换算法比较简单，用一个数组维护了 LRU 的历史列表，可以查询需要被替换的 victim 项，也可以更新某一项到最近访问。

MRU 路预测算法

MRU 路预测算法也比较简单，对于每一个 set，记录一下最近一次访问的 way，如果匹配，就认为是 First Hit；如果不匹配，匹配到了其他的 Way，就认为是 Non-First Hit。

Multi Column 路预测算法

Multi Column 路预测算法比较复杂。首先，对每个 set，记录了一个 bit vector，大小为 way * way，分成 way 的个部分，每个部分中有 way 个 bit。当接受到内存访问请求的时候，从 Tag 的低位获取 Major Location，然后直接去访问 Major Location 对应的 Tag，如果匹配了，就是 First Hit；如果不匹配，就从 bit vector 同一部分剩下的 1 （Selected Locations）中去找，找到了，就是 Non-First Hit。

实现比较麻烦的是如何把 MRU 放到正确的位置。因为，First Hit 的情况下，Major Location 对应的部分中的 Major Location 行就应该对应 MRU，但是有时候并不满足，这时候就要把这两行交换一下，同时要把 LRU 的状态也更新了。更复杂的情况是，如果出现了 Miss，那么被替换掉的条目可能在其他的部分中，这时候就要扫描 bit vector，把该行在其他部分中的 bit 去掉，再进行 swap 操作。

Victim Cache

Victim Cache 就是在正常匹配 Miss 以后，再进行一次查找，采用全相连的方式。需要注意的是，Victim Cache 中的 Tag 应该是原来的 Tag + Index。比较复杂的是数据在 Cache 和 Victim Cache 之间 Swap 的过程，如果 Hit 了 Victim Cache 中的某一项，就要和 Cache 中某一条进行交换；如果在 Cache 和 Victim Cache 中都 Miss 了，那么，从内存中读取数据到 Cache 的同时，如果被替换掉的项目是 Valid；就要挪到 Victim Cache。

统计数据

Task 1

直接映射方法下，Cache 命中率见下：

命中率	直接映射
game	21.80%
office	30.83%
PDF	19.33%
photo	24.45%

可以看到命中率都是非常低的。

Task 2

组关联方式下，Cache 命中率见下：

命中率	2-way	4-way	8-way	16-way
game	23.51%	25.16%	25.57%	28.24%
office	33.75%	35.62%	35.71%	40.35%
PDF	19.41%	18.88%	18.70%	20.19%
photo	19.18%	17.46%	12.68%	11.89%

可以看到，一些 trace 中相连度和命中率是正相关的，一些是负相关，一些是浮动。

Task 3

采用 MRU 路预测方法，一次命中率、非一次命中率和总的命中率见下：

一次/非一次/总命中率	2-way	4-way	8-way	16-way
game	36.04%/63.96%/23.51%	7.79%/92.21%/25.16%	1.34%/98.66%/25.57%	0.59%/99.41%/28.24%
office	46.14%/53.86%/33.75%	20.32%/79.68%/35.62%	5.70%/94.30%/35.71%	3.48%/96.52%/40.35%
PDF	34.38%/65.62%/19.41%	7.05%/92.95%/18.88%	1.10%/98.90%/18.70%	0.56%/99.44%/20.19%
photo	26.69%/73.31%/19.18%	3.69%/96.31%/17.46%	3.25%/96.75%/12.68%	3.36%/96.64%/11.89%

可以看到，MRU 路预测算法随着关联度加大，一次命中率降低地很快。路预测算法不影响总的命中率。

Task 4

采用 Multi-Column 路预测方法。

一次/非一次/总命中率/搜索长度	2-way	4-way	8-way	16-way
game	67.36%/32.64%/23.51%/0.24	57.94%/42.06%/25.16%/0.54	54.04%/45.96%/25.57%/0.70	50.33%/49.67%/28.24%/0.80
office	74.24%/25.76%/33.75%/0.25	66.94%/33.06%/35.62%/0.52	64.77%/35.23%/35.71%/0.69	59.26%/40.74%/40.35%/0.81
PDF	68.62%/31.38%/19.41%/0.25	60.20%/39.80%/18.88%/0.56	56.46%/43.54%/18.70%/0.74	53.13%/46.87%/20.19%/0.85
photo	78.97%/21.03%/19.18%/0.15	72.31%/27.69%/17.46%/0.42	368.33%/31.67%/12.68%/0.58	65.48%/34.52%/11.89%/0.66

可以看到，Multi Column 路预测方法随着关联度加大，一次命中率逐渐降低，但是降低比较缓慢。

Task 5

MRU 路预测方法特别简单，硬件上只需要很少的存储即可实现，缺点是随着路数增多，一次命中率急剧下降，导致这种方法的实用性不大。

Multi-Colunn 路预测方法硬件比 MRU 复杂，但是效果较好，即使路数增多了，一次命中率也可以达到很高的水准，并且就算一次没有命中，需要线性搜索的次数也是比较少的，原论文中除了线性搜索版本，还设计了并行搜索版本，但从这个数据看来，用线性搜索版本即可得到较好的效果。但是，随着路数增多，Multi-Column 需要的存储空间是二次增长，而且交换的操作比较复杂和耗时。

Task 6

Task6 采用不同的块大小，结果如下：

命中率	8B	16B	32B	64B	128B	256B
game	48.67%	42.70%	36.34%	25.16%	42.38%	53.77%
office	60.45%	52.65%	46.04%	35.62%	47.42%	55.10%
PDF	46.59%	38.64%	30.32%	18.88%	31.64%	40.51%
photo	26.84%	25.92%	23.44%	17.46%	54.06%	73.73%

从上表可以看到，命中率和块大小在一定范围内负相关，一定范围内正相关。这和课件上画的曲线是反的，可能和测例本身有关，本次实验提供的 trace 局部性比较差。

Task 7

Task 7 实现了 Victim Cache，在不同大小下，命中率如下：

命中率	256	512	1024	2048
victim1	74.13%	74.53%	75.39%	77.15%
victim2	53.09%	53.58%	54.79%	56.95%

可以看到，随着 Victim Cache 不断增大，命中率不断提高，但是提高缓慢。

一点疑惑

做实验的时候，首先的疑惑就是为什么命中率这么低，在本科的实验中，都是从缺失率的角度来进行统计，因为命中率基本都在 80% 以上。仔细观察 trace 以后，发现一些问题。在本科时候的 trace 中，有很多连续的访问，stride = 8，是比较常见的 64 位 CPU 会呈现的 trace，这样局部性很好，得出来的数据也比较真实。在本实验的 trace 中，连续的访问 stride=64，这是比较奇怪的，导致局部性较差，得出来的命中率很低，怀疑是不是这个 trace 采集的是 L2 的数据，如果 L1 cache 的 block size 是 64，那么对 L2 的访问地址 stride=64 也就可以理解了。