Cache 分析实验报告

《计算机系统结构》（张广艳），实验一
2021-03-31
计 82　陈英发　2018080073

实验内容

本实验尝试用软件模拟几种常见的 Cache 结构，以分析不同设计对于性能的影响。实现了：

- 3 种替换策略：二叉树，LRU，和 Protected LRU（PLRU）。
- 4 种写策略：是以下两个集合的所有组合
  - 写回，写直达
  - 写分配，写不分配
- 12 中 cache 组织：是以下两种集合的所有组合
  - 块大小分别为：8，32，64
  - 直接映射，全相连，4 - 相连，8 - 相连

对于 PLRU 的实现，从论文中得知对于块大小为 64B，相连度为 16 时，最优的参数是保护 14 个最多使用的 line，用 3 比特作为 counter bit。但是因为我们只用 8 - 相连，因此我们按比例选则保护 2 个最多使用的 line（因为 $64:14 \approx 8:2$），然而 counter bit 还是 3 位，其余部分都一样（替换时清零，访问时计数加一）

结果

- 在元数据开销中，x + y 表示：
  - x 字节的空间用于存储 Cache 中各项数据（tag，dirty bit，valid bit 等）
  - y 字节的空间用于存储为了处理替换策略的数据。
- 缺失率的单位是 %。

不同 Cache 组织

缺失率

Cache 组织 (block size, ways count)	trace 1	trace 2	trace 3	trace 4	平均
8, 1	4.9	2.1	23.4	3.7	8.5
8, 4	4.6	1.2	23.3	2.0	7.8
8, 8	4.6	1.2	23.3	1.8	7.7
8, 全相连	4.6	1.2	23.3	1.8	7.7
32, 1	2.2	1.3	9.8	2.3	3.9
32, 4	1.8	0.3	9.6	1.1	3.2
32, 8	1.8	0.3	9.6	0.8	3.1
32, 全相连	1.8	0.3	9.6	0.7	3.1
64, 1	1.5	1.6	5.3	1.9	2.6
64, 4	1.1	0.2	5.0	0.8	1.8
64, 8	1.1	0.2	5.0	0.6	1.7
64, 全相连	1.1	0.2	5.0	0.4	1.7

元数据开销

Cache 组织 (block size, ways count)	trace 1	trace 2	trace 3	trace 4	平均
8, 直接映射	114688+0	114688+0	114688+0	114688+0	114688+0
8, 4	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048
8, 8	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048
8, 全相连	131072+2048	131072+2048	131072+2048	131072+2048	131072+2048
32, 直接映射	28672+0	28672+0	28672+0	28672+0	28672+0
32, 4	28672+512	28672+512	28672+512	28672+512	28672+512
32, 8	28672+512	28672+512	28672+512	28672+512	28672+512
32, 全相连	32768+512	32768+512	32768+512	32768+512	32768+512
64, 直接映射	**14336+0**	**14336+0**	**14336+0**	**14336+0**	**14336+0**
64, 4	14336+256	14336+256	14336+256	14336+256	14336+256
64, 8	14336+256	14336+256	14336+256	14336+256	14336+256
64, 全相连	16384+256	16384+256	16384+256	16384+256	16384+256

不同替换策略

缺失率

替换策略	trace 1	trace 2	trace 3	trace 4	平均
二叉树	4.6	1.2	23.3	1.8	7.7
LRU	4.6	1.2	23.3	1.8	7.7
PLRU	4.6	1.2	23.3	1.8	7.7

元数据开销

Cache 组织 (block size, ways count)	trace 1	trace 2	trace 3	trace 4	平均
二叉树	**114688+2048**	**114688+2048**	**114688+2048**	**114688+2048**	**114688+2048**
LRU	114688+6144	114688+6144	114688+6144	114688+6144	114688+6144
PLRU	114688+12288	114688+12288	114688+12288	114688+12288	114688+12288

不同写策略

缺失率

写策略	trace 1	trace 2	trace 3	trace 4	平均
写回+写分配	**4.6**	**1.2**	**23.3**	**1.8**	**7.7**
写回+写不分配	11.1	8.7	34.5	4.7	14.8
写直达+写分配	**4.6**	**1.2**	**23.3**	**1.8**	**7.7**
写直达+写不分配	11.1	8.7	34.5	4.7	14.8

元数据开销

写策略	trace 1	trace 2	trace 3	trace 4	平均
写回+写分配	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048
写回+写不分配	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048
写直达+写分配	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048
写直达+写不分配	114688+2048	114688+2048	114688+2048	114688+2048	114688+2048

分析

元数据开销

设 b 为块大小（字节），w 为相连度，c 为 cache 总容量（字节），可推出：

$$\text{二叉树元数据} = set_{\text{个数}} \times \text{相连组数} \times \frac{\text{字节}}{\text{比特}} = \frac{\text{块个数}}{w} \times w \times \frac{1}{8} = \frac{c}{8b} = \frac{\text{块个数}}{8} \tag{1}$$

即二叉树的元数据开销是块个数除以 8。而用于存储各 cache line 的元数据开销：（假设是写回策略，所以有 dirty bit）

$$\begin{aligned}
\text{cache line}_{\text{元数据}} &= \text{块个数} \times tag_{\text{位数}} + 2 && (2)\\
&= \frac{c}{b} \times (66 - index_{\text{位数}} - offset_{\text{位数}}) && (3)\\
&= \frac{c}{b} \times (66 - log(set_{\text{个数}}) - log(\text{块大小})) && (4)\\
&= \frac{c}{b} \times (66 - log(\frac{c}{bw}) - log(b)) && (5)\\
&= \frac{c(66 - log(c) + log(w))}{b} && (6)
\end{aligned}$$

从不同 cache 组织的结果中可以看到，不管是直接映射（相连度为 1），还是 4 - 相连，8 - 相连，cache line 的元数据开销都是一样的。这不符合预期，但是是因为按照实验要求，每个 cache line 要按字节对齐，而 w 为 1、4、8 时，log(w) 分别为 0、2、3 $66 - log(c) + log(w)$ 分别为 49、51、52，所以按字节对齐后，每个 cache line 都是用 7 个字节，但是如果看全相连（w 等于块个数），可以发现是满足上述公式的。因此我们可以得出结论，就是 b 越大且 w 越小，空间开销就越小。

至于用于替换策略（二叉树算法）的元数据的空间开销是符合预期的（除了直接映射时，因为那时候不用存储替换策略的数据（因为各 set 只有一个 line 可以被替换出来，只需要查看 valid bit 即可）。

另外，不同写策略显然是不会影响空间开销的，实验结果也验证了这一点。

缺失率

从实验结果可以看到对于实验的 trace，不同 cache 组织中块大小为 64 普遍的平均缺失率比其他块大小的缺失率低，而且几乎对于单独每个 trace 而言亦然。另外全相连和直接映射都普遍比其组相联低，其中 4 - 相连和 8 - 相连没有明显差别。课上讲过块大小越大的时候，首先缺失率下降，后来有上升，但是这里 64 还是最低，所以块大小为 128B 有可能更优，也有可能更差。另外我们知道相连度越高，缺失率就会越低，而全相连是理论上的最优相连度，但是可以发现，其实相连度为 4 及以上就没有太大的改进，但是在耗时上缺有天渊之别。如果不用哈希表作为辅助的话，跑全相连的各个 trace （在我的个人笔记本电脑：16GB RAM，i5-8300H CPU （2.3 GHz，8 core）） 平均花了几到几十分钟，其中块大小大的时候接近一个小时。而相连度为 8 或者更小时，都是秒级的运行时间，可见全相连会严重影响时间开销。这是因为在全相连里，每次都要遍历所有 line 的 tag，判断是否一致，所以时间复杂度是 O(n^2)，其中 n 为访存操作次数。如果用了哈希表则变为 O(n)，结果跑出来也是秒级。

对于不同的替换策略，可以看到缺失率基本上没有区别。通过简单统计程序可以发现在 4 个 trace 中，相同 index 而不同 tag 的访存操作非常少，所以非常少替换操作，因此替换策略也不会对缺失率有很大的影响。实际上，我们可以让模拟程序输出精确的缺失次数：

替换策略	trace 1	trace 2	trace 3	trace 4
二叉树	4347	3371	30094	4992
LRU	4347	3371	30079	5004
PLRU	4347	3371	30086	5027

因此，差异太小所以仅从这四个 trace 文件的模拟结果不能对这三种替换策略在缺失率上的性能做出评价。

从写策略的缺失率的结果看出，写回和写直达没有明显差异，而写分配明显优于写不分配。实际上我们知道写回和写直达对缺失率不会有任何影响，因为在 cache 中查找某块的过程跟写回或者写直达无关。至于写分配缺失率更低是符合直观认知的，因为写不分配忽略了写操作和读操作以及写操作之间的局部性。但是写分配每次分配时，需要进行额外一次替换操作，这意味着比写不分配多一次读操作，但是在采用写回策略时，写分配可能进行一次写操作（当被替换的是一个脏块）而写不分配必定会有一次写操作。实际上，我们可以从模拟过程中统计对存储器的读写操作次数：

写策略	**trace 1**		**trace 2**		**trace 3**		**trace 4**		**平均**	
	写	读	写	读	写	读	写	读	写	读
写回+写分配	**8670**	23239	**0**	6627	**83952**	117192	**195**	9024	**23204.25**	39020.5
写回+写不分配	52203	**4390**	43296	**3913**	143235	**31485**	16544	**7150**	63819.5	**11734.5**
写直达+写分配	227518	23239	175170	6627	211702	117192	205389	9024	204944.8	39020.5
写直达+写不分配	227518	**4390**	175170	**3913**	211702	**31485**	205389	**7150**	204944.8	**11734.5**

可以发现读主存次数跟是写回还是写直达无关，而写不分配普遍比写分配少读，这跟上面的讨论吻合。然而可以发现写回 + 写分配在写主存次数上明显比其他策略优，这是也符合上面的讨论，因为写分配在写操作时只有一定概率进行写主存操作，而写不分配是一定写主存。类似地，写直达对于每个写操作都一定会写主存，而写回则只有在替换时（即 miss 的情况下）才会有一定概率写到主存里。值得注意的是，写回 + 写分配的写主存次数是写直达的两种方案的十分之一。

编译方法

操作系统：Ubuntu 20.04
编程语言：C++11
实验平台：Lenovo y7000，WSL Ubuntu 20.04。

> 注：请使用 Linux 执行。

编译

在 lab1 目录下，执行：`make build`，结果程序是 `main`，位于 `lab1/src` 子目录下。

执行

> 注：程序的读入方法是固定的，不管怎样都会读入 `1.trace`、`2.trace`、`3.trace`、`4.trace`，并且不会读入任何其他文件。所以如果要改成其他输入文件，必须覆盖这四个文件。然后输出不管怎样都会是 `1.log`、`2.log`、`3.log`、`4.log`。
>
> 另外，以下执行任何 make 都会先编译一下（所以其实可以直接执行 `make all` 来检查）。

- 运行所有不同的 cache 组织和策略：`make all`
- 运行不同的 cache 组织：`make structure`
- 运行不同的替换策略：`make replace`
- 运行不同的写策略：`make write`

- 单独运行某一种参数：

```
make build
cd src && ./main <块大小> <组数> <替换策略> <写策略>
```

为了方便，代码中块大小（`block_size`）等于 0 代表全相连。

> 注：全相连非常慢，尤其是块大小比较小或者命中率比较低的时候，可能要一个小时以上。如果不想测试全相连，可以在测试文件（`run_structure.sh`）中注释掉。

输出结果放到 `lab1/output` 子目录下，实验中所要求的 log 文件就在此。另外在 `lab1/output/stats` 子目录下，有含有其他统计数据的文件，助教可以忽视。

文件结构

- **input**：存放 trace 文件
- **output**：存放模拟结果，包括每一个**重点 trace 的访问 Log**。
  - 另外每此用不同参数进行模拟，都会输出一个以参数命名的文件，其中含有一些统计数据。命名格式为：`stats_<块大小>_<块大小>_<块大小>_<块大小>_<块大小>.tsv`。助教可以忽视。
- **report**：实验报告
- **src**：源代码
- **stats**：由 `stats.py` 生成，对 `output` 中的统计文件进行数据处理后的结果，助教可以忽视。
- `run_xxx.sh`：辅助执行文件，助教可以忽视。上述执行的 make 操作实际上就是运行这些文件。

总结

我们通过软件模拟，对 Cache 中不同设计对性能的影响有了更深入的理解。我们模拟了不同的块大小，相连度，替换策略，写策略，并观察他们的 cache 性能的影响。可以发现他们满足理论计算以及课上老师讲过的经验结果。

讨论中发现空间开销比较容易精确预计，与参数的关系较为简单。缺失率则很难精确计算，跟很多问题一样，评测一个系统性能往往很难，但是通过对 cache 的理解，我们可以得到关于性能的一些直观的认知，而可以发现实验结果符合大部分直观认知。

另外，本实验使用的 4 个 trace 文件对于实际计算机应用具有有限的代表性，所以我们绝对不能仅从本实验的实验结果给出哪些参数是更优的。而且，我们本来就没有测试足够多的不同的参数，而且不同参数的