Cache 实验报告

姓名：谢兴宇
学号：2017011326

2020 年 4 月

1 运行方式

本工程可使用常规的单文件 C++ 代码的编译运行方式。
下面给出在开发者环境下的编译运行方式。

1.1 项目结构

```
|-trace
| | astar.trace
| | bzip2.trace
| | gcc.trace
| | mcf.trace
| | perlbench.trace
| | swim.trace
| | twolf.trace
|-log
| | astar.log
| | bzip2.log
| | gcc.log
| | mcf.log
| | perlbench.log
| | swim.log
| | twolf.log
| bitvector.h
```

```
| cache_simulator.cpp
| report.pdf
```

本工程仅包含 cache_simulator.cpp 和 bitvector.h 两个源代码文件。

1.2 测试环境

- Ubuntu 18.04
- G++ 7.4.0

1.3 编译

```
g++ cache_simulator.cpp -o cache_simulator -O2 -std=c++17
```

1.4 运行

```
./cache_simulator
```

2 实现

本次实验需要尝试不同的 Cache 布局、不同的替换策略和不同的写策略，故很好的模块设计和解耦是十分必要的。与常见的 OOP 风格的工程思路不同，我采用了更加紧凑的基于函数指针的设计，使用了三个函数指针

- `void (*cache_replacement)(int, uint64_t)`
- `bool (*cache_write)(uint64_t)`
- `void (*cache_meta_update)(int, int)`

在尝试不同的策略时将它们赋为相应的函数，让工程得以良好地模块化，减少代码冗余。

为了实现更加高效的替换策略以及更加精准地模拟硬件，我实现了BitVector结构，类似于STL库中的bitset和boost库中的dynamic_bitset，其是一个基于unsigned char数组模拟的任意大的二进制数，但可以支持以下功能：

- 以一个 64 位无符号整数初始化

- 以一个 64 位无符号整数赋值
- 与一个 64 位无符号整数直接比较大小
- 取出某一位的值
- 将某一位设为 0/1
- 从一个位区间中取出一个 64 位无符号整数
- 将一个位区间设为一个 64 位无符号整数

我的第一版实现是用`boost`库的`dynamic_bitset`完成的，实现后发现其甚至无法在一小时内模拟 astar 上的直接映射。再自己重新实现了这个`BitVector`之后，运行所有实验共计只需 10 分钟。

3 实验及分析

实验使用的数据包括除了 bodytrack_1m，canneal.uniq 和 streamcluster 三个未标明操作是读是写的 trace 之外的所有数据。

3.1 Cache 布局

下面给出了在每个 trace 下的实验结果

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	23.40%	23.28%	23.28%	23.26%
32B	9.84%	9.63%	9.63%	9.59%
64B	5.27%	4.97%	5.01%	5.00%

表 1: 不同 Cache 布局下的缺失率：astar

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	2.06%	1.22%	1.22%	1.22%
32B	1.33%	0.31%	0.31%	0.31%
64B	1.59%	0.15%	0.15%	0.15%

表 2: 不同 Cache 布局下的缺失率：bzip2

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	4.24%	4.11%	4.10%	4.09%
32B	1.34%	1.20%	1.19%	1.19%
64B	0.85%	0.67%	0.68%	0.68%

表 3: 不同 Cache 布局下的缺失率：gcc

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	4.94%	4.58%	4.58%	4.58%
32B	2.20%	1.82%	1.82%	1.82%
64B	1.46%	1.08%	1.08%	1.08%

表 4: 不同 Cache 布局下的缺失率：mcf

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	3.67%	2.07%	1.79%	1.75%
32B	2.31%	1.14%	0.82%	0.66%
64B	1.89%	0.39%	0.85%	0.62%

表 5: 不同 Cache 布局下的缺失率：perlbench

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	6.58%	6.54%	6.54%	6.54%
32B	2.16%	2.12%	2.12%	2.11%
64B	1.22%	1.15%	1.15%	1.15%

表 6: 不同 Cache 布局下的缺失率：swim

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	1.18%	1.14%	1.14%	1.14%
32B	0.39%	0.34%	0.34%	0.34%
64B	0.27%	0.20%	0.20%	0.20%

表 7: 不同 Cache 布局下的缺失率：twolf

下面我们对已获得的数据做一些统计分析：

块大小	直接映射	4 路组关联	8 路组关联	全关联
8B	6.58%	6.13%	6.09%	6.08%
32B	2.80%	2.37%	2.32%	2.29%
64B	1.79%	1.23%	1.30%	1.27%

表 8: 不同 Cache 布局下的缺失率：平均值

从上表中可以看出，直接映射的缺失率要略高于组关联和全关联，块越大的 Cache 缺失率也会越低。

不同 Cache 布局的元数据开销空间如下表所示（假定采用写直达策略，无 dirty 位，不考虑替换策略所需的元数据）：

块大小	直接映射	4 路组相联	8 路组相联	全关联
8B	96.00KiB	100.00KiB	102.00KiB	124.00KiB
32B	24.00KiB	25.00KiB	25.50KiB	30.00KiB
64B	12.00KiB	14.75KiB	12.50KiB	12.75KiB

表 9: 不同 Cache 布局的元数据开销

从表中可以看出，块越大，元数据开销越大；关联度越高，元数据开销也会增大。

3.2 Cache 替换策略

LRU、二叉树、随机三种替换策略执行动作的差异：

- LRU 策略：替换栈底的路，将替换后的路重新置于栈顶。
- 二叉树策略：从二叉树的根节点开始，遇到一个标为 0 的节点就向他的右儿子走，遇到一个标为 1 的节点就向他的左儿子走，直到找到一路，将这一路替换，更新其祖先的标记。
- 随机策略：在组内随机选一路替换。

不同替换策略的元数据开销如下表所示

替换策略	LRU	二叉树	随机
元数据开销	6KiB	1.75KiB	0

表 10: 不同替换策略的元数据开销

三种策略在各 trace 上的缺失率如下表所示（avg 表示平均值）：

Trace	LRU	二叉树	随机
astar	23.28%	23.29%	23.23%
bzip2	1.22%	1.22%	1.22%
gcc	4.10%	4.09%	4.12%
mcf	4.58%	4.58%	4.60%
perlbench	1.79%	1.78%	1.79%
swim	6.54%	6.54%	6.57%
twolf	1.14%	1.14%	1.14%
avg	6.093%	6.086%	6.094%

表 11: 不同替换策略的缺失率

从表中可以看出，三种策略的缺失率大致排名为：二叉树 « LRU < 随机

相较于 LRU，二叉树算法使用了更少的元数据，却获得了更低的缺失率，这或许可以佐证二叉树算法是比 LRU 算法更好的算法。

3.3 写策略

Trace	写不分配 写直达	写不分配 写回	写分配 写直达	写分配 写回
astar	34.50%	34.50%	23.28%	23.28%
bzip2	8.67%	8.67%	1.22%	1.22%
gcc	8.67%	8.67%	4.10%	4.10%
mcf	11.15%	11.15%	4.58%	4.58%
perlbench	4.66%	4.66%	1.79%	1.79%
swim	9.61%	9.61%	6.54%	6.54%
twolf	1.4%	1.4%	1.14%	1.14%
avg	11.24%	11.24%	6.09%	6.09%

表 12: 不同写策略的缺失率

对不同写策略的实验结果如上表所示（其中 avg 表示在各 trace 上的均值），从中可以发现：

- 写回或写直达对于缺失率没有影响
- 采用写不分配策略的缺失率要显著高于写分配策略