Lab4 缓存模拟器

181220075 周其乐

Cache 的实现

common.h

首先在 common.h 中将一个 cache 行定义为一个结构体。

```
typedef struct{
    bool valid_bit;            //有效位
    bool dirty_bit;            //脏位
    uint32_t tag_bit;          //标记位
    uint32_t block_label;      //块号
    uint8_t BLOCK[BLOCK_SIZE];//块大小为 64B
} cache_row;
```

为了方便后面修改 cache 构造进行性能测试，并为了能够初始化创建二维 whole_cache。将 TOTAL_SIZE_WIDTH 与 ASSOCIATIVITY_WIDTH 定义为宏。
因为 cache 有若干组，每组有若干行。所以将整个 cache 组织为一个二维结构，分为“cache 组数 * 每组行数”。两个维数的大小通过以下宏进行计算。
[exp2(TOTAL_SIZE_WIDTH **-** ASSOCIATIVITY_WIDTH **-** BLOCK_WIDTH**)]**
[exp2(ASSOCIATIVITY_WIDTH**)]**
另外，定义 cache 和主存中的有关变量，便于程序使用，提高代码可读性。

init_cache

先根据 total_size_width，associativity_width，BLOCK_WIDTH，对 cache 和主存中的有关变量进行初始化。然后对 whole_cache 中的每个 cache_row 进行初始化。

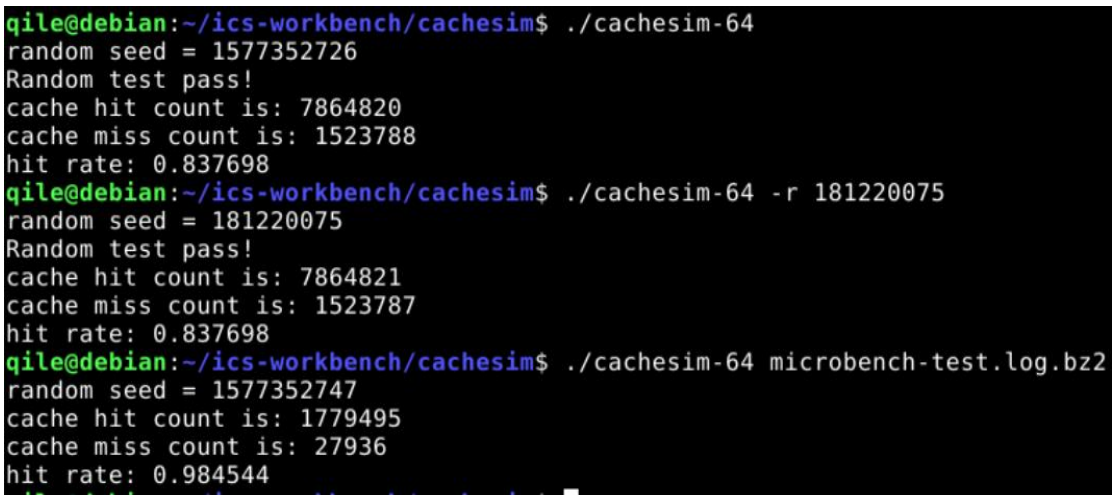
cache_read

根据 addr 获得 cache 组号与标记位后，匹配 tag 并检查有效位。若命中，则直接读 cache（注意找地址要&~0x3）。若未命中，则先判断组内有无空行，有空行则根据地址更新 cache_row。若组中已满，则先以 clock（）作为随机数种子，来随机选择一个行，若该行未被修改过（脏位为 false），则直接进行替换。若该行修改过，则先写回，再替换。最后将各分支算得的 *ret_addr 返回。

cache_write

与 read 的过程相似，先获得 cache 组号与标记位，匹配 tag 并检查有效位。若命中，则调用自己增加的 write_data 函数按掩码要求进行写入。若未命中，且组中有空行，则将该行有效位赋值 true，并更新标志位 tag_bit 和块号 block_label。接着从内存中读入数据后，进行 write_data。若组中已满，则先设置种子进行随机替换，再 write_data。

初步实现结果



尝试建模

cache 的复杂性

实验中我把复杂性建模为以下几个方面之和：

（1） 在每次 HIT 判断时，2 * 访问的 cache 行数。选择系数为 2 是因为在 HIT 判断时，要分别对 cache_row 的 tag_bit 和 valid_bit 进行检查。

（2） 在每次对组内是否有空闲行时，1 * 访问的 cache 行数。选择系数为 1 是因为在 empty 判断时，只需要对 valid_bit 进行检查。

（3） 在更新 cache 行时，3 * 更新的 cache 行总数。选择系数为 3 是因为在更新 cache 行时，要分别对 valid_bit，tag_bit，block_label 进行赋值。根据代码中的实现，这个衡量指标也将“随机替换”涵盖在内。

cache miss 的代价

实验中我在 cache_read 和 cache_write 函数中，都使用 clock 函数，通过 miss 分支产生的时间代价来进行度量。最后将结果除以 1000 获得以毫秒为单位的度量。

尝试修改 cache 的参数(为控制变量，对单个变量测试时，其他变量都为尝试修改前原值)

尝试修改 TOTAL_SIZE_WIDTH

	hit rate	hit judge cost	empty judge cost	update line cost	cache complex	cache miss cost (ms)
10	**98.2966%**	9199720	123124	153915	**9476759**	**144.513**
12	**98.4165%**	9014952	114388	142975	**9272315**	**136.962**
14	**98.4532%**	8919106	111448	138760	**9169314**	**132.900**
16	**98.4851%**	8688830	107988	132350	**8929168**	**122.928**
18	**98.7042%**	8465034	87536	97185	**8649755**	**98.026**
20	**99.4054%**	5471914	19962	1330	**5493206**	**15.363**
22	**99.4054%**	3684222	10756	1330	**3696308**	**14.093**

在 TOTAL_SIZE_WIDTH 增大时，命中率单调递增，cache 复杂度单调递减，cache miss 的代价单调减少。分析因为因为总大小增大，储存的数据多，命中率自然会增加，从而 cache miss 的情况少，总代价减少是自然的。而 cache complex 的下降也应该与命中率增大有关。因为若命中率增加，就越容易成功命中，否则就要遍历一整组来进行 hit judge 等判断。而 TOTAL_SIZE_WIDTH 在超过 16 后，cache 复杂性和 miss 代价下降极快，分析应与指数的增大迅速有关，不是反常现象。但是在 TOTAL_SIZE_WIDTH 从 20 增加到 22 时，命中率几乎一样，cache miss 的代价也是仅有少量减小。这可以为设计较适合的 cache 做参考。

尝试修改 ASSOCIATIVITY_WIDTH

	hit rate	hit judge cost	empty judge cost	update line cost	cache complex	cache miss cost (ms)
1	**98.4516%**	5431342	55844	138900	**5626086**	**131.149**
2	**98.4532%**	8919106	111448	138760	**9169314**	**132.900**
3	**98.4535%**	16498088	222720	138730	**16859538**	**132.222**
4	**98.4532%**	30983862	445408	138760	**31568030**	**140.668**
5	**98.4516%**	57534728	891584	138900	**58565212**	**150.864**

在 ASSOCIATIVITY_WIDTH 增大时，命中率几乎不变，cache 复杂度单调递增，miss 的代价单调递增。而其中 cache 复杂度的增加速度吻合二次关系。分析，关联度增加，相联比较的电路越复杂。在 hit judge，empty judge 等对 cache 行判断中，代价与每组的行数紧密相关。所以关联度的增加，导致每次要遍历 cache 组时，检查的行数直接增加，从而导致 cache 复杂度增加。命中率基本不变代价却会增大，这可以为设计较合适的 cache 做参考。

microbench-test 的 cache 最佳设计

从对 **TOTAL_SIZE_WIDTH** 和 **ASSOCIATIVITY_WIDTH** 的修改，可以得出结论：

对于 **TOTAL_SIZE_WIDTH**，参数越大，命中率单调递增，cache 复杂度单调递减，cache miss 的代价单调减少。但是参数增大到 20 后，命中率与 cache miss 的代价减少速度变小，趋于平稳。因此我认为应该将 **TOTAL_SIZE_WIDTH** 设置为 20。

对于 **ASSOCIATIVITY_WIDTH**，参数越大，命中率几乎不变，cache 复杂度单调递增，miss 的代价单调递增。反而参数为 1 时，拥有几乎相等的命中率，且 cache 的复杂度和 miss 的代价为最低水平。所以我认为关联度应该设置为 1。这样的组相联映射就变为了直接映射。

因此根据实验结论，我认为，对于 microbench-test，最佳的 cache 设计为：

大小为 1MB，采取直接映射。

设计运行结果：

命中率 99.405%，cache 复杂度 4007989，cache miss 代价 6.076ms，相较于其他设计确实表现良好。

```
qile@debian:~/ics-workbench/cachesim$
random seed = 1577365001
hit rate: 0.994050
hit judge cost: 3993256
empty judge cost: 13318
update line cost: 1415
cache complex: 4007989
cache miss cost: 6.076000 ms
```

感谢老师的审阅