专业：电子信息工程
姓名：王涵
学号：3200104515
日期：2022.12.23
地点：家

浙江大学实验报告

课程名称：计算机组成与设计 指导老师：沈继忠,赵武锋,唐奕,屈民军 成绩：

实验名称：Cache Organization and Performance Evaluation 实验类型：设计型

一、 Cache 实现思路

本次实验需要完成 init_cache(), perform_access(addr, access_type), flush()三个函数，分别对应 cache 的初始化，cache 的执行访问以及最后的清除。以下是三个部分详细的设计思路

1.1 init_cache

在 init_cache()中，我们需要将 cache_stat_inst 以及 cache_stat_data 这两个记录 cache 性能的结构体变量清零，进行初始化；以外还要根据是否 cache_split 来对 data cache, inst cache 进行初始化清零，以及计算 index_mask 等基本参数等操作

我的思路是如果 cache 需要 split，则把 cache c1 作为 data cache，cache c2 作为 inst cache，如果不需要 split，则只使用 cache c1

C1/c2 结构体赋值的逻辑如下，

Parameters	赋值逻辑
Size	若 cache split 则赋值 cache_dsize/cache_isize，若不需要 cache split，则赋值 cache_dsize
Associativity	Cache 关联度，直接赋值 cache_assoc
N_sets	根据 cache_size/(assoc*block_size)进行计算
Index_mask_offset	Offset 的位数，使用 LOG2(block_size)计算
Index_mask	index 掩码，index 对应的位置置 1，其余位置 0
LRU_head	Cache line 的头，置 NULL
LRU_tail	Cache line 的尾，置 NULL
Set_contents	每个 set 中的有效位，开始置 0

Cache_stat_data/inst 赋值逻辑如下，

Parameters	作用	赋值

Accesses	Data/instruction cache 访问的次数	0
Misses	Cache miss 的次数	0
Replacements	覆盖的次数	0
Demand_fetches	从 memory 中 fetch words 的数量	0
Copies_back	写回到 memory 的 words 的数量	0

1.2 Perform_access()

这部分是核心代码，基于模块化的原则，我根据 access_type 又设置了三个函数，分别为 data_load_reference, data_store_reference, inst_load_reference 对应 access_type 为 0,1,2 的情况。

首先，进入函数，需要根据 addr 计算出 tag 和 index，计算 tag 的思路和计算 index 一致，先计算出 mask_offset，然后将 1 左移 mask_offset 位-1，再做取反即可得到掩码值，计算式子如下，

```
unsigned tag_data,tag_inst,index_data,index_inst;
index_data = (addr&c1.index_mask)>>c1.index_mask_offset;
index_inst = (addr&c2.index_mask)>>c2.index_mask_offset;
```

```
unsigned tag_mask_offset = LOG2(cache_block_size)+LOG2(c1.n_sets);
unsigned tag_mask = ~((1<<tag_mask_offset)-1);
tag_data = (addr&tag_mask)>>tag_mask_offset;

tag_mask_offset = LOG2(cache_block_size)+LOG2(c2.n_sets);
tag_mask = ~((1<<tag_mask_offset)-1);
tag_inst = (addr&tag_mask)>>tag_mask_offset;
```

之后根据 access_type 调用不同的函数，值得注意的是当 cache_split 时，inst_load_reference 传入的是 c2，但当!cache_split 为 1 时，传入的应该是 c1。

另外，读指令和读数据虽然读的数据不同，但是操作基本相同，实现的逻辑也是基本一致，但读和写的逻辑相差很多，因此以下从读操作和写操作出发，阐述实现逻辑

1.2.1 读操作(data_load_reference, inst_load_reference)

进入函数，首先 accesses 访问次数需要加一，然后根据 index 比较 tag 是否存在一致。我设置了 flag 标志，再使用类型 cache line 的指针 LRU_tmp 和 LRU_head[index]中的每一项的 tag 进行比较，若 tag 一致则代表 cache hit，若不一致则继续寻找，最后根据 flag 是否为 1 判断 cache 是否命中，以下为代码截图。

```
cache_stat_inst.accesses++;
Pcache_line LRU_tmp;
LRU_tmp = c->LRU_head[index];
int flag=0;
while (1){
        if (LRU_tmp==NULL){
                // printf("inst LRU_tmp.\n");
                break;
        }

        if (LRU_tmp->tag==tag){
                flag=1;
                // printf("cache hit.\n");
                break;
        }
        if (LRU_tmp->LRU_next==NULL){
        //      printf("LRU_tmp->LRU_next==NULL.\n");
                break;
        }
        LRU_tmp = LRU_tmp->LRU_next;
        if (c->associativity==1)
                break;
}
```

如果 cache miss 了，flag 为 0，这时需要判断 index 位置中 cache 是否已满，未满，则从内存中拷入，content 增加；若已满，则需要根据 LRU 原则覆盖掉最近没有使用过的 cache，这对应到 cache line 中为 LRU_tail[index]，覆盖之前还需要根据 writeback 抑或 writethrough 的区别进行细分，如果 writeback 且该位置数据被修改过是 dirty，则需要将数据拷回到内存，若不是则可以直接删除

更新策略如下表，

Hits/miss	Accesses	Misses	Replace	DF	CB
hit	✓	/	/	/	/
Miss(full)	✓	✓	✓	✓	✓ (writeback && dirty)
Miss(not full)	✓	✓	/	✓	/

1.2.2 写操作(data_store_reference)

写操作中需要考虑写失效和命中之后不同的策略。首先，寻找 tag 的方式和读操作一致，但是 cache hit 中要分 write back 和 write through，cache miss 中要分 write allocated 和 no-write allocated

如果 cache hit，需要更新 LRU 链表的顺序，通过先 delete 后 insert 的方法将数据移到头部，然后根据写策略区分如下操作

- ✧ write back: 把 dirty 置 1
- ✧ write through: copies_back++

若 cache miss，

- ✧ write allocate: 从内存中拷一段数据进来，demand_fetches 需要增加
  - ✧ cache 已满，则根据 LRU 替换，若被 evict 的位置为 dirty 且 writeback 则 copies_back++，再根据 write back/through 的区别对拷入的数据进行操作
  - ✧ 若 cache 未满，则直接插入一段数据，content++
- ✧ write no allocation: copies_back++

更新策略如下表，wb=write back, wt=write through, wa=write allocation, wna=write no allocation，✔需要更新

策略	Hit/miss	Access	Miss	Replace	DF	CB
Wb&Wa	hit	✓	/	/	/	/
	Miss(full)		✓	✓	✓	✓(dirty)
	Miss(not full)		✓	/	✓	/
Wt&Wa	hit		/	/	/	✓
	Miss(full)		✓	✓	✓	✓(dirty)
	Miss(not full)		✓	/	✓	✓
Wb&Wna	hit		/	/	/	/
	Miss(full)		✓	/	/	✓
	Miss(not full)		✓	/	/	✓

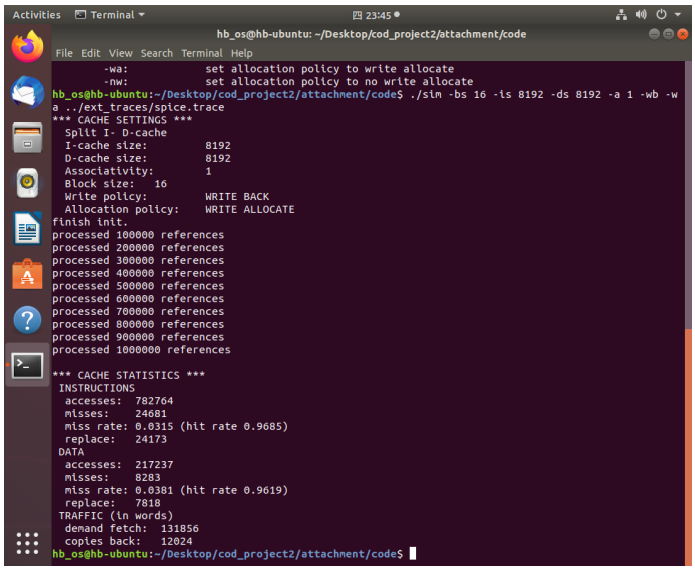
1.3 Flush

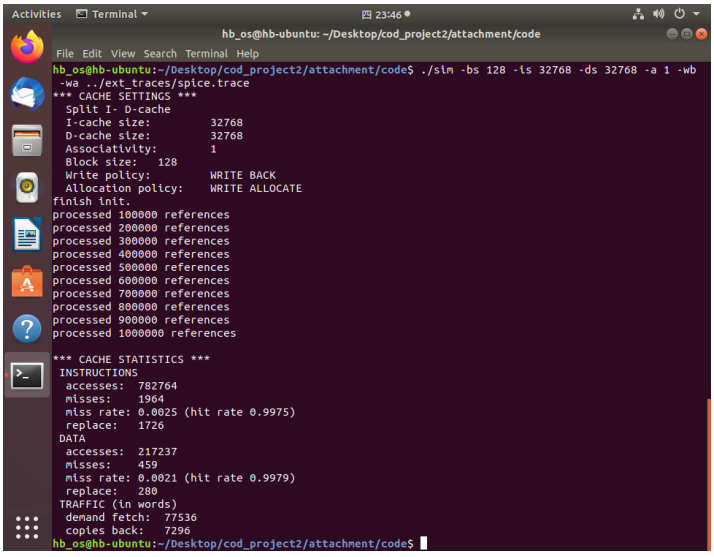
最后需要把所有 dirty 位为 1 的数据写回到内存，因为 write through 策略并不会修改 dirty，因此 dirty 位为 1 肯定对应 write back，而且 inst 中不可能存在 dirty 位为 1 的数据，因为没有对 inst 写的操作；所有我们只需要寻找 c1 结构体中是否有 dirty 位为 1 的，进行 flush 操作即可

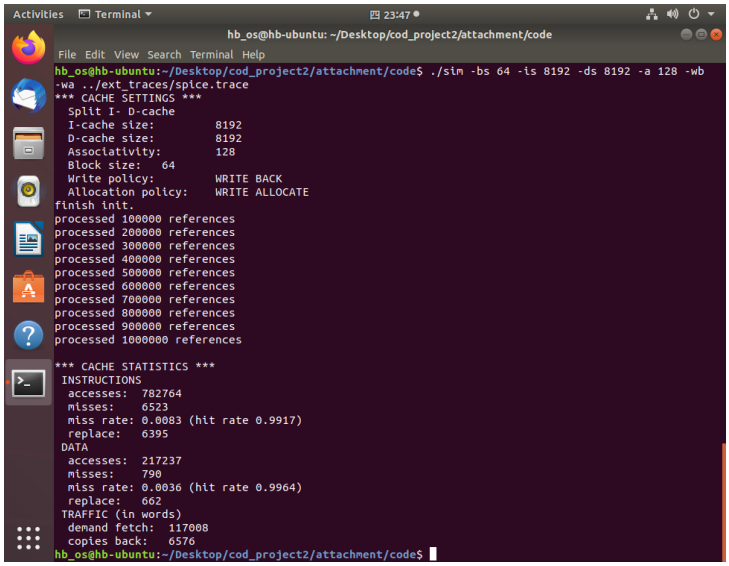
1.4 Validate

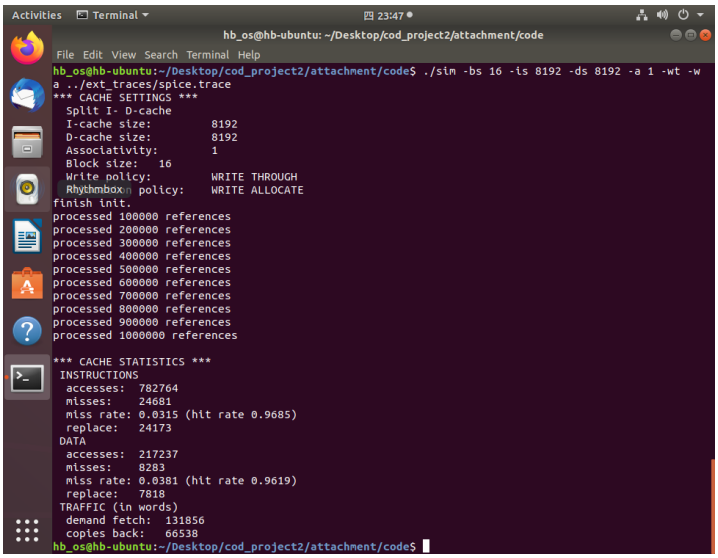
此部分为程序正确性的验证，本人已经对照 sim.pdf 中 splice.trace 在不同参数下对照了各个参数结构，验证均正确，以下展示部分的截图，具体参数可见命令行

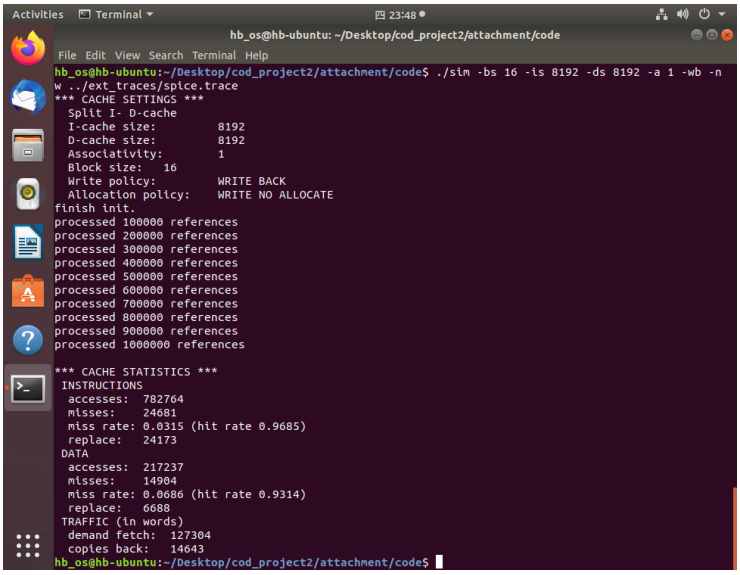
参数

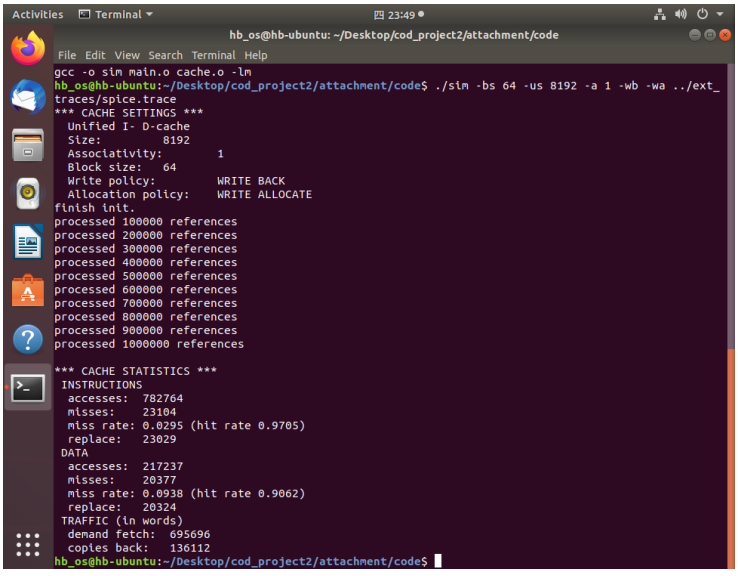

Activities
Terminal
23:45
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
-wa: set allocation policy to write allocate
-nw: set allocation policy to no write allocate
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 16 -is 8192 -ds 8192 -a 1 -wb -wa ../ext_traces/spice.trace
*** CACHE SETTINGS ***
Split I- D-cache
I-cache size: 8192
D-cache size: 8192
Associativity: 1
Block size: 16
Write policy: WRITE BACK
Allocation policy: WRITE ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 24681
miss rate: 0.0315 (hit rate 0.9685)
replace: 24173
DATA
accesses: 217237
misses: 8283
miss rate: 0.0381 (hit rate 0.9619)
replace: 7818
TRAFFIC (in words)
demand fetch: 131856
copies back: 12024
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$


Activities
Terminal
23:46
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 128 -is 32768 -ds 32768 -a 1 -wb -wa ../ext_traces/spice.trace
*** CACHE SETTINGS ***
Split I- D-cache
I-cache size: 32768
D-cache size: 32768
Associativity: 1
Block size: 128
Write policy: WRITE BACK
Allocation policy: WRITE ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 1964
miss rate: 0.0025 (hit rate 0.9975)
replace: 1726
DATA
accesses: 217237
misses: 459
miss rate: 0.0021 (hit rate 0.9979)
replace: 280
TRAFFIC (in words)
demand fetch: 77536
copies back: 7296
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$


Activities
Terminal
23:47
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 64 -is 8192 -ds 8192 -a 128 -wb -wa ../ext_traces/spice.trace
*** CACHE SETTINGS ***
Split I- D-cache
I-cache size: 8192
D-cache size: 8192
Associativity: 128
Block size: 64
Write policy: WRITE BACK
Allocation policy: WRITE ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 6523
miss rate: 0.0083 (hit rate 0.9917)
replace: 6395
DATA
accesses: 217237
misses: 790
miss rate: 0.0036 (hit rate 0.9964)
replace: 662
TRAFFIC (in words)
demand fetch: 117008
copies back: 6576
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$

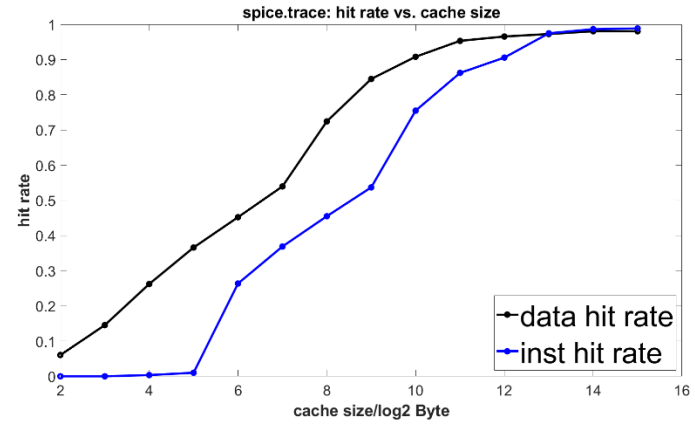

Activities
Terminal
23:47
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 16 -is 8192 -ds 8192 -a 1 -wt -w
a ../ext_traces/spice.trace
*** CACHE SETTINGS ***
Split I- D-cache
I-cache size: 8192
D-cache size: 8192
Associativity: 1
Block size: 16
Write policy: WRITE THROUGH
Rhythmbox policy: WRITE ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 24681
miss rate: 0.0315 (hit rate 0.9685)
replace: 24173
DATA
accesses: 217237
misses: 8283
miss rate: 0.0381 (hit rate 0.9619)
replace: 7818
TRAFFIC (in words)
demand fetch: 131856
copies back: 66538
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$

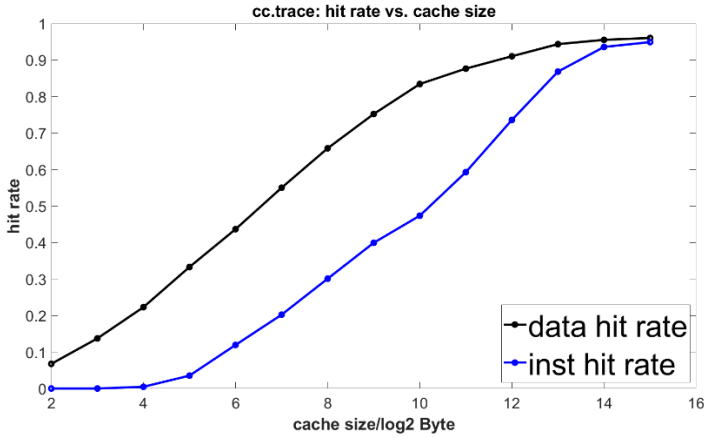

Activities
Terminal
23:48
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 16 -is 8192 -ds 8192 -a 1 -wb -n
w ../ext_traces/spice.trace
*** CACHE SETTINGS ***
Split I- D-cache
I-cache size: 8192
D-cache size: 8192
Associativity: 1
Block size: 16
Write policy: WRITE BACK
Allocation policy: WRITE NO ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 24681
miss rate: 0.0315 (hit rate 0.9685)
replace: 24173
DATA
accesses: 217237
misses: 14904
miss rate: 0.0686 (hit rate 0.9314)
replace: 6688
TRAFFIC (in words)
demand fetch: 127304
copies back: 14643
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$

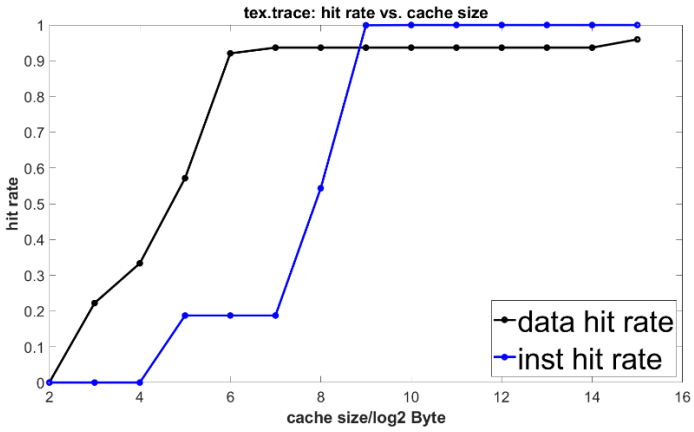

Activities
Terminal
23:49
hb_os@hb-ubuntu: ~/Desktop/cod_project2/attachment/code
File Edit View Search Terminal Help
gcc -o sim main.o cache.o -lm
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$ ./sim -bs 64 -us 8192 -a 1 -wb -wa ../ext_
traces/spice.trace
*** CACHE SETTINGS ***
Unified I- D-cache
Size: 8192
Associativity: 1
Block size: 64
Write policy: WRITE BACK
Allocation policy: WRITE ALLOCATE
finish init.
processed 100000 references
processed 200000 references
processed 300000 references
processed 400000 references
processed 500000 references
processed 600000 references
processed 700000 references
processed 800000 references
processed 900000 references
processed 1000000 references
*** CACHE STATISTICS ***
INSTRUCTIONS
accesses: 782764
misses: 23104
miss rate: 0.0295 (hit rate 0.9705)
replace: 23029
DATA
accesses: 217237
misses: 20377
miss rate: 0.0938 (hit rate 0.9062)
replace: 20324
TRAFFIC (in words)
demand fetch: 695696
copies back: 136112
hb_os@hb-ubuntu:~/Desktop/cod_project2/attachment/code$

二、 思考题

2.1 Working Set Characterization



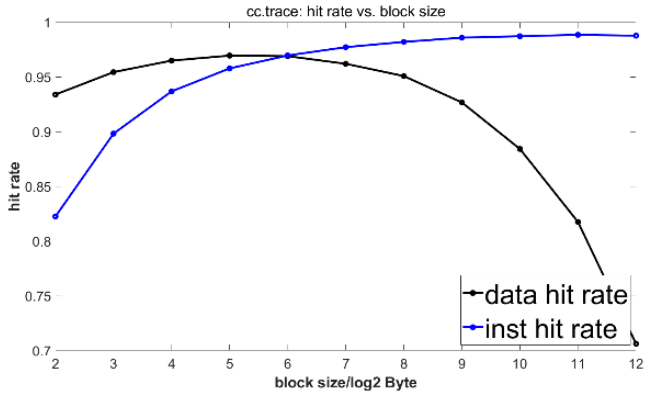


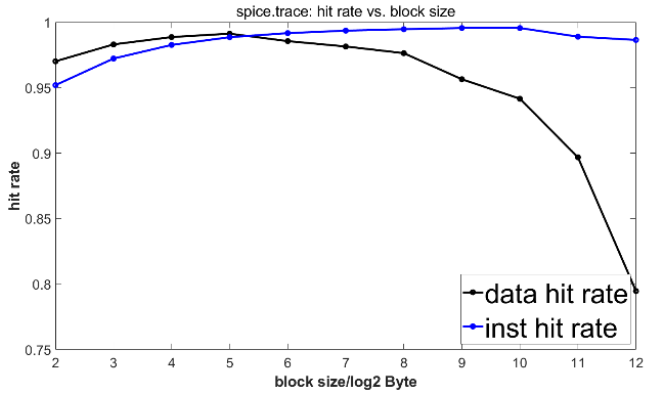


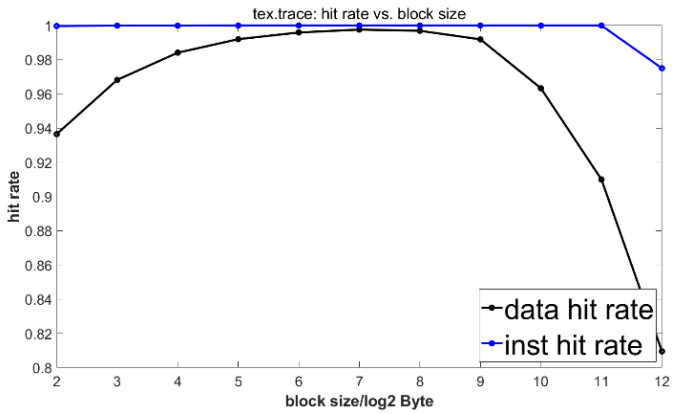
2.1.1 这实验通过修改 cache size 的大小，观察 hit rate 的变化，当 cache size 越大，hit rate 也会越大，原因在于 cache 的容量变大了，那么存储的数据变多，自然地，在 cache 中找不到数据的几率也会减小；当 cache_size 大到一定数量之后，hit rate 就会趋向饱和，不再增加

2.1.2 spice.trace 中 instruction 的 working set size 为 322KB，data 的 16KB；cc.trace 中 inst 为 128KB，data 为 64KB；tex 中 inst 为 1KB，data 为 32KB

2.2 Impact of Block Size







2.2.1 根据图像，可以发现 hit rate 随着 block size 的增大先增大，后减小。其原因可能在于，block 大小的增加使得数据空间局部性增加，因此 hit rate 会有些许的上升，后来随着 block size 的增加，而 cache size 保持不变，那么 n_sets 会减小，那么需要

替换的数据可能会增多，因此 block 再大，会导致 replacements 的增加，从而导致 cache miss

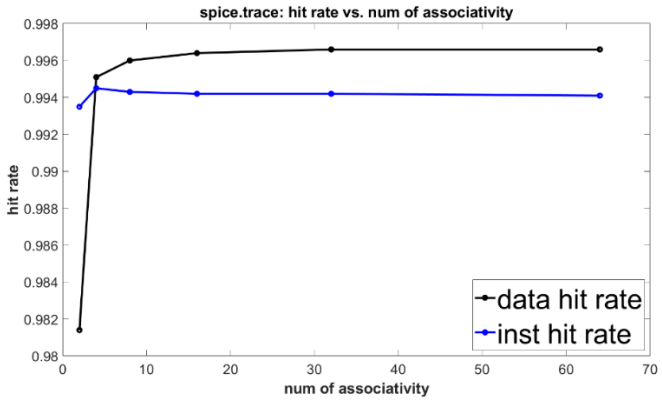
2.2.2 对于 spice.trace，对于 inst，选择 1K 的 block size，对于 data，选择 32B

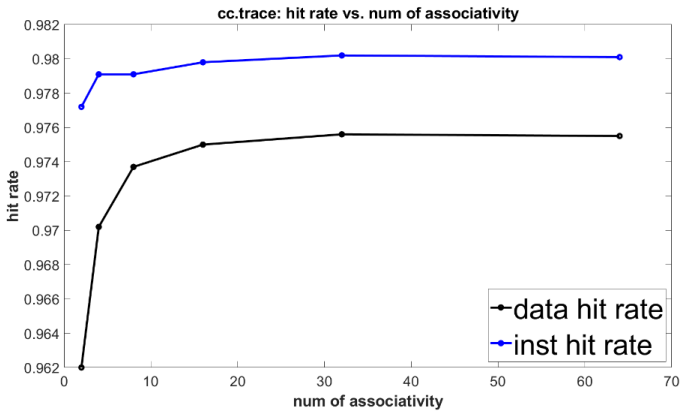
对于 cc.trace，对于 data，选择 32B，对于 inst，选择 2K

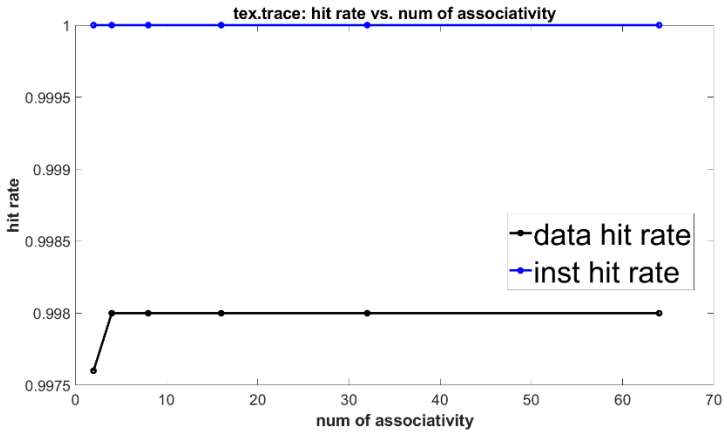
对于 tex.trace，对于 inst，选择 4B~2K，对于 data，选择 128B

2.2.3 是的。这告诉我们指令 cache 的性能和数据 cache 的性能可能会适合不同的参数设计，可以通过分开设计硬件的方式，使两者性能同时达到最优。

2.3 Impact of Associativity







2.3.1 随着 associativity 的增加，hit rate 也会增加，然后会趋于饱和；原因可能是每个 set 中可存的条目变多，在读写操作中产生覆盖某个 cache line 的几率下降，因此 hit rate 上升

2.3.2 是的，两者存在不同，但是区别不大；associativity 数值的变化对于 hit rate 影响不大，且对 data reference 和 inst reference 的区别也不大，但在 associativity 较小时，data cache hit rate 变化较大，inst cache hit rate 变化较小，可以在设计时，可以着重考虑 data cache 的 associativity

2.4 Memory Bandwidth

我选择在 tex.trace 中进行测试，数据如下

Cache_size(KB)	Block_size(B)	assoc	Wb	Wt	Wa	Nw	DF	CB
8	64	4	✓			✓	3312	29903
8	128	4	✓			✓	4064	29935
16	64	2	✓			✓	2464	29909
16	128	2	✓			✓	2656	29957
8	64	4		✓		✓	3312	104513
8	128	4		✓		✓	4064	104513
16	64	2		✓		✓	2464	104513
16	128	2		✓		✓	2656	104513
8	64	4	✓		✓		15408	7568
8	128	4	✓		✓		15648	7648
16	64	2	✓		✓		11696	7616

16	128	2	✓		✓		13696	8256

2.4.1 观察数据发现，write back 和 write through 的 demand fetches 是一致的，但 write through 中 copies back 比 write back 要高出不少，write back 的 memory traffic 较小。原因在于两者 write back 和 write through 的策略并不会改变从内存中申请的空间，因此 demand_fetches 值是一致的，但是 write though 的 copies back 要高出很多，因为每次写操作后，write through 都要直接写回内存，而 write back 只是先将 dirty 位置 1，等待被替换才写回内存，操作次数较少，因此 copies back 较小

2.4.2 可以反转，只要写操作次数够少，那么 write through 每次只修改一个字节，write back 中每次都需要替换一个 block size 的数据，次数较少时，copies back 的情况可能会出现反转

2.4.3 write allocate 的 demand_fetches 高于 write no allocate，copies back 的次数小于 write no allocate。原因在于，每次写失效后，write allocate 都需要从内存中申请一段空间拷贝到 cache，而 write no allocate 不用，因此 demand fetches 自然会高；而 write allocate 策略下 copies back 较小的原因可能是当写失效后，数据拷贝到 cache 中后，被替换的概率较小，因此 copies back 较小，但这也与指令有关，并不绝对。

2.4.4 是的。当使用 write allocate 策略，发生写失效后，数据拷贝到 cache 中，被替换的概率较大，那么会频繁的 copies back，而且写回数据的大小一次为一个 block size，此时使用 write allocate 就会造成 copies back 较大