
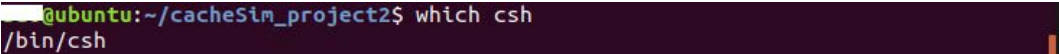
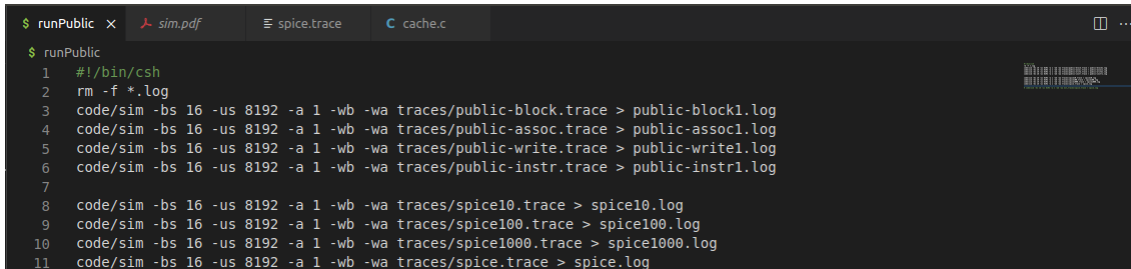


cache simulator

环境配置

- 使用linux即可
- 测试脚本需要修改
 - 测试脚本中使用的是csh `#!/usr/bin/csh`
 - 这与我电脑上csh的路径不同
 - 故修改测试脚本首行为 `#!/bin/csh`
 - 此外该脚本还有一处问题，即spice.trace文件所在路径错误 `code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/spice.trace > spice.log`，有两个解决方案，一是把spice.trace复制到测试脚本中的位置；二是修改测试脚本，我在实验时采取了前者。

- 最终测试脚本如下图



```
$ runPublic x  sim.pdf  spice.trace  cache.c
$ runPublic
1  #!/bin/csh
2  rm -f *.log
3  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/public-block.trace > public-block1.log
4  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/public-assoc.trace > public-assoc1.log
5  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/public-write.trace > public-writel.log
6  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/public-instr.trace > public-instr1.log
7
8  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/spice10.trace > spice10.log
9  code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/spice100.trace > spice100.log
10 code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/spice1000.trace > spice1000.log
11 code/sim -bs 16 -us 8192 -a 1 -wb -wa traces/spice.trace > spice.log
```

- 后续画图我使用的是python，所以还需要安装matplotlib包 `pip install matplotlib`
 - 后续报错 `ImportError: No module named _tkinter`
 - `sudo apt-get install python-tk` 成功解决问题

代码实现

cache初始化需要确定对应值，其中size，associativity等是仿真时传入的参数，而组数和对应的掩码则需要自行根据传入的参数进行计算。相关代码如下：

```
/* **** */
void init_cache()
{
    if(cache_split)
    {
        init_cache(&c1, cache_isize);
        init_cache(&c2, cache_dsize); // 如果需要使用分离的cache，则c1用作指令cache，c2
        用作数据cache
    }
    else
        init_cache(&c1, cache_usize);

    // 把统计量都初始化为0
    cache_stat_inst.accesses = 0;
    cache_stat_inst.misses = 0;
```

```

cache_stat_inst.replacements    = 0;
cache_stat_inst.demand_fetches  = 0;
cache_stat_inst.copies_back     = 0;

cache_stat_data.accesses        = 0;
cache_stat_data.misses          = 0;
cache_stat_data.replacements    = 0;
cache_stat_data.demand_fetches  = 0;
cache_stat_data.copies_back     = 0;
}

void init_cache(cache *c, int size)
{

    c->size            = size;
    c->associativity    = cache_assoc;
    c->n_sets           = size/(cache_assoc * cache_block_size);    //set数 = cache的
大小/每个set的大小
    c->LRU_head        = (Pcache_line *)malloc(sizeof(Pcache_line)*(c->n_sets));
    c->LRU_tail        = (Pcache_line *)malloc(sizeof(Pcache_line)*(c->n_sets));
    c->set_contents     = (int *)malloc(sizeof(int)*(c->n_sets));
    c->contents         = 0;

    c->index_mask_offset = LOG2(cache_block_size);
    c->index_mask = ((1<<LOG2(c->n_sets))-1) << c->index_mask_offset ;
    for(int i=0; i< c->n_sets; i++)
    {
        c->LRU_head[i] = NULL;
        c->LRU_tail[i] = NULL;
        c->set_contents[i] = 0;
    }

}

/*****/

```

cache操作的关键代码如下：

首先根据是否使用分离的cache，使得icache和dcache指针指向正确的cache结构体。在这里，我令c1为指令cache，c2为数据cache。若不使用分离的cache，则都为c1。然后根据access_type来选择对应的操作。我已经封装为perform_data_load、perform_data_store、perform_inst_load三个函数。这三个函数都有及其详尽的注释。在这里就不重复展开。

```

/*****/
void perform_access(addr, access_type)
    unsigned addr, access_type;
{
    if(cache_split)
    {
        icache = &c1;
        dcache = &c2;
    }
    else
    {
        icache = &c1;
    }
}

```

```

    dcache = &c1;
}

switch (access_type)
{
case TRACE_DATA_LOAD:      // data load
    perform_data_load(dcache, addr);
    break;
case TRACE_DATA_STORE:     // data store
    perform_data_store(dcache, addr);
    break;
case TRACE_INST_LOAD:      // instruction load
    perform_inst_load(icache, addr);
    break;
default:
    break;
}
}
/*****/

void perform_data_load(cache* dcache, unsigned addr)
{
    int index;
    unsigned int bitsSet, bitsOffset, tagMask, tag;
    int block_size_in_words = cache_block_size/WORD_SIZE;

    // 计算tag
    bitsSet = LOG2(dcache->n_sets);
    bitsOffset = LOG2(cache_block_size);
    tagMask = 0xFFFFFFFF << (bitsOffset + bitsSet);
    // 按位与运算获取地址高位->tag
    tag = addr & tagMask;

    // 取出index对应的bit后, 要右移对应的偏移量以得到正确的结果
    index = (addr & dcache->index_mask) >> dcache->index_mask_offset;

    cache_stat_data.accesses++;

    if(dcache->LRU_head[index] == NULL)
    {
        // 冷启动阶段, cache中一条记录都没有
        cache_stat_data.misses++;
        cache_stat_data.demand_fetches += block_size_in_words;

        Pcache_line new_item = malloc(sizeof(cache_line));

        new_item->tag = tag;
        new_item->dirty = 0;
        dcache->set_contents[index] = 1;
        insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], new_item);
    }
    else
    {
        if(dcache->set_contents[index] == dcache->associativity)

```

```

{
    // 当前组已满，则首先检查是对应地址的数据是否已经在cache中
    // 若在，cache hit，修改LRU即可
    // 若不在，cache miss，需要替换掉最久之前访问的记录

    Pcache_line cache_line = dcache->LRU_head[index];
    int tag_found = FALSE;

    // 遍历已有的记录，查找对应地址的数据是否已经在cache中
    while(cache_line)
    {
        if(cache_line->tag == tag)
        {
            tag_found = TRUE;
            break;
        }
        cache_line = cache_line->LRU_next;
    }

    if(tag_found) // cache hit，调整LRU，将访问的记录加到链表头（表示最近访问）
    {
        delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);
        insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);
    }
    else
    {
        // cache miss，替换掉最久之前访问的记录
        cache_stat_data.demand_fetches += block_size_in_words;
        cache_stat_data.misses++;
        cache_stat_data.replacements++;

        if(dcache->LRU_tail[index]->dirty) // 如果dirty bit=1，说明该数据在cache中被
        修改过，需要写回内存
            cache_stat_data.copies_back += block_size_in_words;

        Pcache_line new_item = malloc(sizeof(cache_line));
        new_item->tag = tag;
        new_item->dirty = 0;

        // 移除最久之前访问的记录
        delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], dcache-
        >LRU_tail[index]);
        // 将新的记录添加到LRU的链表头
        insert(&(&dcache->LRU_head[index]), &(&dcache->LRU_tail[index]),
        new_item);
    }
}
else
{
    // 如果尚有valid bit=0的行，那么在cache miss的情况下，优先写入该行
    int tag_found = FALSE;
    Pcache_line cache_line = dcache->LRU_head[index];

    // 遍历已有的记录，查找对应地址的数据是否已经在cache中

```

```

while(cache_line)
{
    if(cache_line->tag == tag)
    {
        tag_found = TRUE;
        break;
    }
    cache_line = cache_line->LRU_next;
}

if(tag_found) // cache hit, 调整LRU, 将访问的记录加到链表头 (表示最近访问)
{
    delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);
    insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);
}
else // cache miss
{

    cache_stat_data.demand_fetches += block_size_in_words;
    cache_stat_data.misses++;

    Pcache_line new_item = malloc(sizeof(cache_line));
    new_item->tag = tag;
    new_item->dirty = 0;

    // 将新写入的记录加到LRU链表头
    insert(&(dcache->LRU_head[index]), &(dcache->LRU_tail[index]),
new_item);

    dcache->set_contents[index]++; // entry数++
}
}
}

void perform_data_store(cache* dcache, unsigned addr)
{
    int index;
    unsigned int bitsSet, bitsOffset, tagMask, tag;
    int block_size_in_words = cache_block_size/WORD_SIZE;

    // 计算tag
    bitsSet = LOG2(dcache->n_sets);
    bitsOffset = LOG2(cache_block_size);
    // 按位与运算获取地址高位->tag
    tagMask = 0xFFFFFFFF << (bitsOffset + bitsSet);
    tag = addr & tagMask;

    // 取出index对应的bit后, 要右移对应的偏移量以得到正确的结果
    index = (addr & dcache->index_mask) >> dcache->index_mask_offset;

    cache_stat_data.accesses++;

    if(dcache->LRU_head[index] == NULL)
    {

```

```

// 冷启动阶段，cache中一条记录都没有
// 使用write-no-allocate策略的时候，如果记录不在cache中，只更新memory，不需要把数
据搬到cache中
// 否则，需要先把数据fetch到cache中，然后再按照write hit的情况来处理
if(cache_writealloc==0)
{
    cache_stat_data.copies_back++;
    cache_stat_data.misses++;
}
else
{
    cache_stat_data.misses++;
    cache_stat_data.demand_fetches += block_size_in_words;

    Pcache_line new_item = malloc(sizeof(cache_line));

    new_item->tag = tag;
    new_item->dirty = 1;

    // 使用write through策略时，更新cache数据时，同时也要更新memory
    if(cache_writeback==0)
    {
        cache_stat_data.copies_back += 1;
        new_item->dirty = 0; // 写回后，dirty bit需要清零
    }

    dcache->set_contents[index] = 1;
    insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], new_item);
}
}
else // cache中有记录
{
    // cache非空，则先检查记录是否在cache中
    // 若在，则写入，并根据写回策略决定是否写入memory还是更改dirty bit
    if(dcache->set_contents[index] == dcache->associativity)
    {
        // 该组已满

        int tag_found = FALSE;
        Pcache_line cache_line = dcache->LRU_head[index];

        // 遍历已有的记录，查找对应地址的数据是否已经在cache中
        while(cache_line)
        {
            if(cache_line->tag == tag)
            {
                tag_found = TRUE;
                break;
            }
            cache_line = cache_line->LRU_next;
        }

        if(tag_found) // cache hit, 调整LRU, 将访问的记录加到链表头（表示最近访问）
        {
            delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);

```

```

insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);

    // dirty bit置1
    dcache->LRU_head[index]->dirty = 1;

// 使用write through策略时, 更新cache数据时, 同时也要更新memory
if(cache_writeback==0)
{
    cache_stat_data.copies_back += 1;
    dcache->LRU_head[index]->dirty = 0; // 写回后, dirty bit需要清零
}
}
else
{
    // cache miss
    // 如果使用write no allocate策略的时候, 如果记录不在内存中, 只更新memory, 不需要把
数据搬到cache中
    // 否则, 需要从memory中fetch数据, 而又因为该组已满, 所以需要执行LRU替换
    if(cache_writealloc==0)
    {
        cache_stat_data.copies_back += 1;
        cache_stat_data.misses++;
    }
    else
    {
        cache_stat_data.demand_fetches += block_size_in_words;
        cache_stat_data.misses++;
        cache_stat_data.replacements++;

        Pcache_line new_item = malloc(sizeof(cache_line));
        new_item->tag = tag;
        new_item->dirty = 1;

        if(dcache->LRU_tail[index]->dirty) // 如果dirty bit=1, 说明该数据在cache中
被修改过, 需要写回
            cache_stat_data.copies_back += block_size_in_words;

        // 使用write through策略时, 更新cache数据时, 同时也要更新memory
        if(cache_writeback==0)
        {
            cache_stat_data.copies_back += 1;
            new_item->dirty = 0; // 写回后, dirty bit需要清零
        }

        // 移除最久之前访问的记录
        delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], dcache-
>LRU_tail[index]);
        // 将新的记录添加到LRU的链表头
        insert(&(dcache->LRU_head[index]), &(dcache->LRU_tail[index]),
new_item);
    }
}
}
else // cache中尚有valid bit=0的记录
{

```

```

// 尚有valid bit=0的行，那么在cache miss的情况下，优先写入该行
int tag_found = FALSE;
Pcache_line cache_line = dcache->LRU_head[index];

// 遍历已有的记录，查找对应地址的数据是否已经在cache中
while(cache_line)
{
    if(cache_line->tag == tag)
    {
        tag_found = TRUE;
        break;
    }
    cache_line = cache_line->LRU_next;
}

if(tag_found) // cache hit, 调整LRU, 将访问的记录加到链表头（表示最近访问）
{
    delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);
    insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], cache_line);

    // dirty bit置1
    dcache->LRU_head[index]->dirty = 1;

    // 使用write through策略时，更新cache数据时，同时也要更新memory
    if(cache_writeback==0)
    {
        cache_stat_data.copies_back += 1;
        dcache->LRU_head[index]->dirty = 0; // 写回后，dirty bit需要清零
    }
}
else
{
    // cache miss
    // 如果使用write no allocate策略的时候，如果记录不在内存中，只更新memory，不需要把
    数据搬到cache中
    // 否则，需要从memory中fetch数据，又因为该组已满，所以需要执行LRU替换
    if(cache_writealloc==0)
    {
        cache_stat_data.copies_back += 1;
        cache_stat_data.misses++;
    }
    else
    {
        cache_stat_data.demand_fetches += block_size_in_words;
        cache_stat_data.misses++;

        Pcache_line new_item = malloc(sizeof(cache_line));
        new_item->tag = tag;
        new_item->dirty = 1;

        // 使用write through策略时，向cache写入数据时，同时也要向memory写一份
        if(cache_writeback==0)
        {
            cache_stat_data.copies_back += 1;

```



```

        new_item->dirty = 0; // 写回后, dirty bit需要清零
    }

    // 将新的记录添加到LRU的链表头
    insert(&(dcache->LRU_head[index]), &(dcache->LRU_tail[index]),
new_item);

    dcache->set_contents[index]++; // entry++
    }
    }
}
}
}

```

// inst load和 data load如果不使用分离的cache时除统计量不同外, 其他则完全一致, 而使用分离的cache时只是所使用的cache不同而已, 整个过程是一样的

```

void perform_inst_load(cache* icache,unsigned addr)
{
    int index;
    unsigned int bitsSet, bitsOffset,tagMask,tag;
    int block_size_in_words = cache_block_size/WORD_SIZE;

    // 计算tag
    bitsSet = LOG2(icache->n_sets);
    bitsOffset = LOG2(cache_block_size);
    tagMask = 0xFFFFFFFF << (bitsOffset + bitsSet);
    // 按位与运算获取地址高位->tag
    tag = addr & tagMask;

    // 取出index对应的bit后, 要右移对应的偏移量以得到正确的结果
    index = (addr & icache->index_mask) >> icache->index_mask_offset;

    cache_stat_inst.accesses++;

    if(icache->LRU_head[index] == NULL)
    {
        // 冷启动阶段, cache中一条记录都没有
        cache_stat_inst.misses++;
        cache_stat_inst.demand_fetches += block_size_in_words;

        Pcache_line new_item = malloc(sizeof(cache_line));

        new_item->tag = tag;
        new_item->dirty = 0;
        icache->set_contents[index] = 1;
        insert(&icache->LRU_head[index], &icache->LRU_tail[index], new_item);
    }
    else
    {
        if(icache->set_contents[index] == icache->associativity)
        {
            // 当前组已满, 则首先检查是对应地址的数据是否已经在cache中
            // 若在, cache hit, 修改LRU即可
            // 若不在, cache miss, 需要替换掉最久之前访问的记录

```

```

Pcache_line cache_line = icache->LRU_head[index];
int tag_found = FALSE;

// 遍历已有的记录，查找对应地址的数据是否已经在cache中
while(cache_line)
{
    if(cache_line->tag == tag)
    {
        tag_found = TRUE;
        break;
    }
    cache_line = cache_line->LRU_next;
}

if(tag_found) // cache hit, 调整LRU, 将访问的记录加到链表头（表示最近访问）
{
    delete(&icache->LRU_head[index], &icache->LRU_tail[index], cache_line);
    insert(&icache->LRU_head[index], &icache->LRU_tail[index], cache_line);
}
else
{
    // cache miss, 替换掉最久之前访问的记录
    cache_stat_inst.demand_fetches += block_size_in_words;
    cache_stat_inst.misses++;
    cache_stat_inst.replacements++;

    if(icache->LRU_tail[index]->dirty) // 如果dirty bit=1, 说明该数据在cache中被
    修改过, 需要写回
        cache_stat_inst.copies_back += block_size_in_words;

    Pcache_line new_item = malloc(sizeof(cache_line));
    new_item->tag = tag;
    new_item->dirty = 0;

    // 移除最久之前访问的记录
    delete(&icache->LRU_head[index], &icache->LRU_tail[index], icache-
    >LRU_tail[index]);
    // 将新的记录添加到LRU的链表头
    insert(&icache->LRU_head[index], &icache->LRU_tail[index],
    new_item);
}
}
else
{
    // 如果尚有valid bit=0的行, 那么在cache miss的情况下, 优先写入该行
    int tag_found = FALSE;
    Pcache_line cache_line = icache->LRU_head[index];

    // 遍历已有的记录，查找对应地址的数据是否已经在cache中
    while(cache_line)
    {
        if(cache_line->tag == tag)
        {
            tag_found = TRUE;
            break;

```

```

    }
    cache_line = cache_line->LRU_next;
}

if(tag_found) // cache hit, 调整LRU, 将访问的记录加到链表头 (表示最近访问)
{
    delete(&icache->LRU_head[index], &icache->LRU_tail[index], cache_line);
    insert(&icache->LRU_head[index], &icache->LRU_tail[index], cache_line);
}
else // cache miss
{
    cache_stat_inst.demand_fetches += block_size_in_words;
    cache_stat_inst.misses++;

    Pcache_line new_item = malloc(sizeof(cache_line));
    new_item->tag = tag;
    new_item->dirty = 0;

    // 将新写入的记录加到LRU链表头
    insert(&(icache->LRU_head[index]), &(icache->LRU_tail[index]),
new_item);

    icache->set_contents[index]++; // entry数++
}
}
}
}

```

flush函数主要用于最后的程序结束时将cache中的内容写入memory, 代码如下:

```

void flush()
{
    int block_size_in_words = cache_block_size/WORD_SIZE;
    Pcache_line cache_line;

    for(int i=0; i < c1.n_sets; i++)
    {
        cache_line = c1.LRU_head[i];

        for(int j = 0; j < c1.set_contents[i]; j++)
        {
            if(cache_line->dirty) // 如果dirty bit为1, 则需要写回memory
                cache_stat_data.copies_back += block_size_in_words;

            cache_line = cache_line->LRU_next;
        }
    }
    // 如果使用分离的cache, 则需要对c2执行同样的操作
    if(cache_split)
    {
        for(int i=0; i < c2.n_sets; i++)
        {
            cache_line = c2.LRU_head[i];

```

```

        for(int j = 0; j < c2.set_contents[i]; j++)
        {
            if(cache_line->dirty)
                cache_stat_inst.copies_back += block_size_in_words;

            cache_line = cache_line->LRU_next;
        }
    }
}
}

```

测试结果

```

public-associ1.log - cacheSim_project2 - Visual Studio Code
Run Terminal Help
public-associ1.log x public-block1.log public-instr1.log public-write1.log spice.log ... public-associ1.out public-block1.out public-instr1.out public-write1.out spice1.out ...
public-associ1.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 9
12 misses: 9
13 miss rate: 1.0000 (hit rate 0.0000)
14 replace: 8
15 DATA
16 accesses: 9
17 misses: 9
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 8
20 TRAFFIC (in words)
21 demand fetch: 36
22 copies back: 0
23
outputs > public-associ1.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 9
12 misses: 9
13 miss rate: 1.0000 (hit rate 0.0000)
14 replace: 8
15 DATA
16 accesses: 9
17 misses: 9
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 8
20 TRAFFIC (in words)
21 demand fetch: 36
22 copies back: 0
23

```

```

public-block1.log - cacheSim_project2 - Visual Studio Code
Run Terminal Help
public-associ1.log public-block1.log x public-instr1.log public-write1.log spice.log ... public-associ1.out public-block1.out public-instr1.out public-write1.out spice1.out ...
public-block1.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 0
12 misses: 0
13 miss rate: 0 (0)
14 replace: 0
15 DATA
16 accesses: 22
17 misses: 2
18 miss rate: 0.0909 (hit rate 0.9091)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 8
22 copies back: 0
23
outputs > public-block1.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 0
12 misses: 0
13 miss rate: 0 (0)
14 replace: 0
15 DATA
16 accesses: 22
17 misses: 2
18 miss rate: 0.0909 (hit rate 0.9091)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 8
22 copies back: 0
23

```

```

public-instr1.log - cacheSim_project2 - Visual Studio Code
Run Terminal Help
public-associ1.log public-block1.log public-instr1.log x public-write1.log spice.log ... public-associ1.out public-block1.out public-instr1.out public-write1.out spice1.out ...
public-instr1.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 5
12 misses: 5
13 miss rate: 1.0000 (hit rate 0.0000)
14 replace: 5
15 DATA
16 accesses: 2
17 misses: 2
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 1
20 TRAFFIC (in words)
21 demand fetch: 28
22 copies back: 0
23
outputs > public-instr1.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 5
12 misses: 5
13 miss rate: 1.0000 (hit rate 0.0000)
14 replace: 5
15 DATA
16 accesses: 2
17 misses: 2
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 1
20 TRAFFIC (in words)
21 demand fetch: 28
22 copies back: 0
23

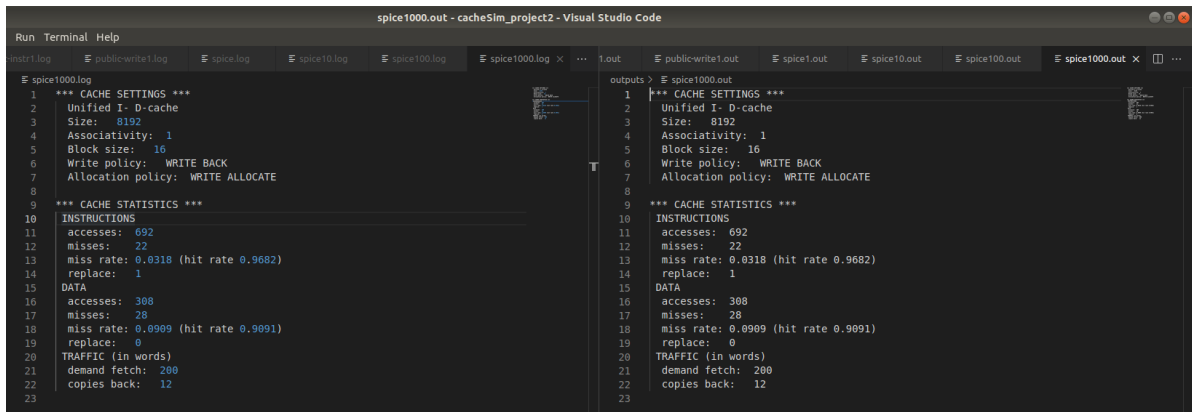
```

```
public-write1.out - cacheSim_project2 - Visual Studio Code
Run Terminal Help
public-associ1.log public-block1.log public-instr1.log public-write1.log x spice.log | ... public-associ1.out public-block1.out public-instr1.out public-write1.out x spice1.out
F public-write1.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 0
12 misses: 0
13 miss rate: 0 (0)
14 replace: 0
15 DATA
16 accesses: 5
17 misses: 2
18 miss rate: 0.4000 (hit rate 0.6000)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 8
22 copies back: 8
23
outputs > F public-write1.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 0
12 misses: 0
13 miss rate: 0 (0)
14 replace: 0
15 DATA
16 accesses: 5
17 misses: 2
18 miss rate: 0.4000 (hit rate 0.6000)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 8
22 copies back: 8
23
```

```
spice1.out - cacheSim_project2 - Visual Studio Code
Run Terminal Help
public-associ1.log public-block1.log public-instr1.log public-write1.log x spice.log x ... public-associ1.out public-block1.out public-instr1.out public-write1.out x spice1.out x
F spice.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8 processed 100000 references
9 processed 200000 references
10 processed 300000 references
11 processed 400000 references
12 processed 500000 references
13 processed 600000 references
14 processed 700000 references
15 processed 800000 references
16 processed 900000 references
17 processed 1000000 references
18
19 *** CACHE STATISTICS ***
20 INSTRUCTIONS
21 accesses: 782764
22 misses: 36136
23 miss rate: 0.0462 (hit rate 0.9538)
24 replace: 35787
25 DATA
26 accesses: 217237
27 misses: 21261
28 miss rate: 0.0979 (hit rate 0.9021)
29 replace: 21098
30 TRAFFIC (in words)
31 demand fetch: 229588
32 copies back: 37844
33
outputs > F spice1.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8 processed 100000 references
9 processed 200000 references
10 processed 300000 references
11 processed 400000 references
12 processed 500000 references
13 processed 600000 references
14 processed 700000 references
15 processed 800000 references
16 processed 900000 references
17 processed 1000000 references
18
19 *** CACHE STATISTICS ***
20 INSTRUCTIONS
21 accesses: 782764
22 misses: 36136
23 miss rate: 0.0462 (hit rate 0.9538)
24 replace: 35787
25 DATA
26 accesses: 217237
27 misses: 21261
28 miss rate: 0.0979 (hit rate 0.9021)
29 replace: 21098
30 TRAFFIC (in words)
31 demand fetch: 229588
32 copies back: 37844
33
```

```
spice10.out - cacheSim_project2 - Visual Studio Code
Run Terminal Help
instr1.log public-write1.log x spice.log x spice10.log x spice100.log x spice1000.log ... instr1.out public-write1.out x spice1.out x spice10.out x spice100.out x spice1000.out
F spice10.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 8
12 misses: 3
13 miss rate: 0.3750 (hit rate 0.6250)
14 replace: 0
15 DATA
16 accesses: 2
17 misses: 2
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 20
22 copies back: 0
23
outputs > F spice10.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 8
12 misses: 3
13 miss rate: 0.3750 (hit rate 0.6250)
14 replace: 0
15 DATA
16 accesses: 2
17 misses: 2
18 miss rate: 1.0000 (hit rate 0.0000)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 20
22 copies back: 0
23
```

```
spice100.out - cacheSim_project2 - Visual Studio Code
Run Terminal Help
instr1.log public-write1.log x spice.log x spice10.log x spice100.log x spice1000.log ... instr1.out public-write1.out x spice1.out x spice10.out x spice100.out x spice1000.out
F spice100.log
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 71
12 misses: 20
13 miss rate: 0.2817 (hit rate 0.7183)
14 replace: 1
15 DATA
16 accesses: 29
17 misses: 11
18 miss rate: 0.3793 (hit rate 0.6207)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 124
22 copies back: 12
23
outputs > F spice100.out
1 *** CACHE SETTINGS ***
2 Unified I- D-cache
3 Size: 8192
4 Associativity: 1
5 Block size: 16
6 Write policy: WRITE BACK
7 Allocation policy: WRITE ALLOCATE
8
9 *** CACHE STATISTICS ***
10 INSTRUCTIONS
11 accesses: 71
12 misses: 20
13 miss rate: 0.2817 (hit rate 0.7183)
14 replace: 1
15 DATA
16 accesses: 29
17 misses: 11
18 miss rate: 0.3793 (hit rate 0.6207)
19 replace: 0
20 TRAFFIC (in words)
21 demand fetch: 124
22 copies back: 12
23
```



对比输出结果和提供的.out文件，结果均一致。

性能评估

测试脚本及绘图脚本

在最后的性能评估与思考问题部分，我们需要更改对应的cache参数，分析其对cache性能的影响。

其中Working Set Characterization用于测试的shell脚本如下：

```
#!/bin/bash
rm /home/noname/cachesim_project2/ext_traces/*.txt # 首先，需要移除上一次保存测试结果的文件。因为我是使用追加的方式来保存结果，如果在新的测试开始前不删除原先的结果，则会不断追加。
cnt=0
k=0
for f in /home/noname/cachesim_project2/ext_traces/* # 遍历ext_traces文件夹下的.trace文件
do
    (( cnt++ ))
    for ((i=1; i <= 268435456 ; i=2*i));
    do
        k=$((i*4))
        echo -e 'Executing ' $f 'with cache size ' $k
        ./code/sim -is $k -ds $k -bs 4 -a $i -wb -wa $f >> "$f.txt" # 保存结果
    done
    grep -E "miss rate|cache size" "$f.txt" > "$cnt.txt" # 通过grep对结果进行进一步筛选，以提取用于作图的数据
done
```

统计结果文件如下图：

```

ext_traces > ≡ cc.trace.txt
1  *** CACHE SETTINGS ***
2  Split I- D-cache
3  I-cache size: 4
4  D-cache size: 4
5  Associativity: 1
6  Block size: 4
7  Write policy: WRITE BACK
8  Allocation policy: WRITE ALLOCATE
9  processed 100000 references
10 processed 200000 references
11 processed 300000 references
12 processed 400000 references
13 processed 500000 references
14 processed 600000 references
15 processed 700000 references
16 processed 800000 references
17 processed 900000 references
18 processed 1000000 references
19 |
20 *** CACHE STATISTICS ***
21 INSTRUCTIONS
22 accesses: 757341
23 misses: 757341
24 miss rate: 1.0000 (hit rate 0.0000)
25 replace: 757340
26 DATA
27 accesses: 242661
28 misses: 226269
29 miss rate: 0.9324 (hit rate 0.0676)
30 replace: 226268
31 TRAFFIC (in words)
32 demand fetch: 983610
33 copies back: 82235
34 *** CACHE SETTINGS ***

```

但这显然有很多多余的数据，所以我们通过grep命令对结果进一步筛选，筛选后，结果如下：

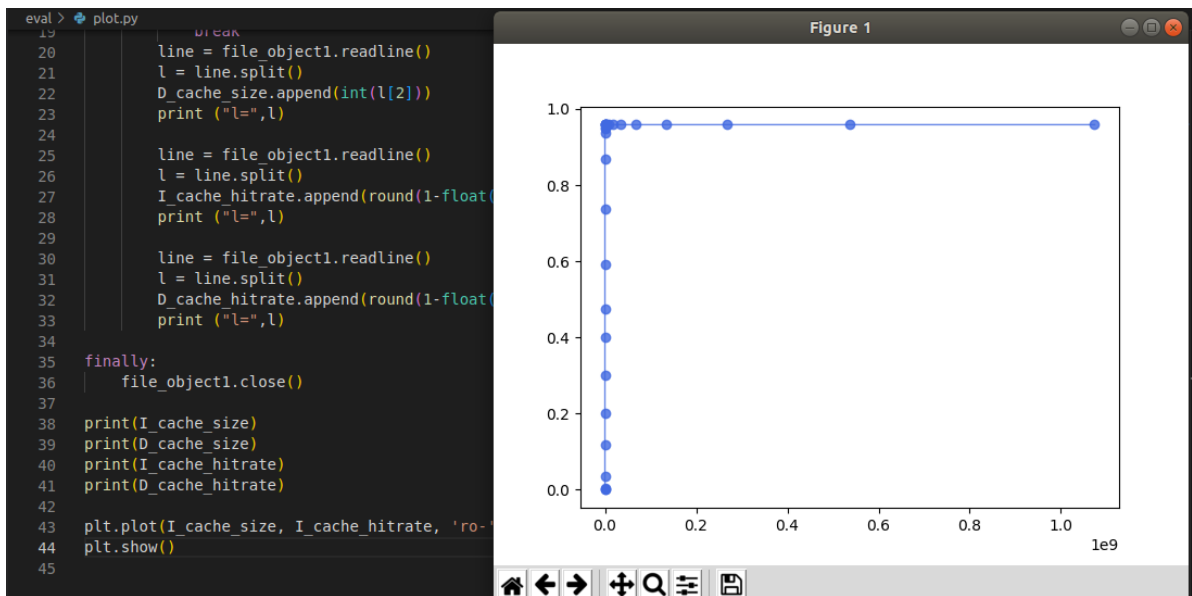
```

eval > ≡ 1.txt
1  I-cache size: 4
2  D-cache size: 4
3  miss rate: 1.0000 (hit rate 0.0000)
4  miss rate: 0.9324 (hit rate 0.0676)
5  I-cache size: 8
6  D-cache size: 8
7  miss rate: 1.0000 (hit rate 0.0000)
8  miss rate: 0.8627 (hit rate 0.1373)
9  I-cache size: 16
10 D-cache size: 16
11 miss rate: 0.9953 (hit rate 0.0047)
12 miss rate: 0.7771 (hit rate 0.2229)
13 I-cache size: 32
14 D-cache size: 32
15 miss rate: 0.9647 (hit rate 0.0353)
16 miss rate: 0.6670 (hit rate 0.3330)
17 I-cache size: 64
18 D-cache size: 64
19 miss rate: 0.8806 (hit rate 0.1194)
20 miss rate: 0.5634 (hit rate 0.4366)
21 I-cache size: 128
22 D-cache size: 128
23 miss rate: 0.7977 (hit rate 0.2023)
24 miss rate: 0.4498 (hit rate 0.5502)
25 I-cache size: 256
26 D-cache size: 256
27 miss rate: 0.6990 (hit rate 0.3010)
28 miss rate: 0.3417 (hit rate 0.6583)
29 I-cache size: 512
30 D-cache size: 512
31 miss rate: 0.6007 (hit rate 0.3993)
32 miss rate: 0.2477 (hit rate 0.7523)

```

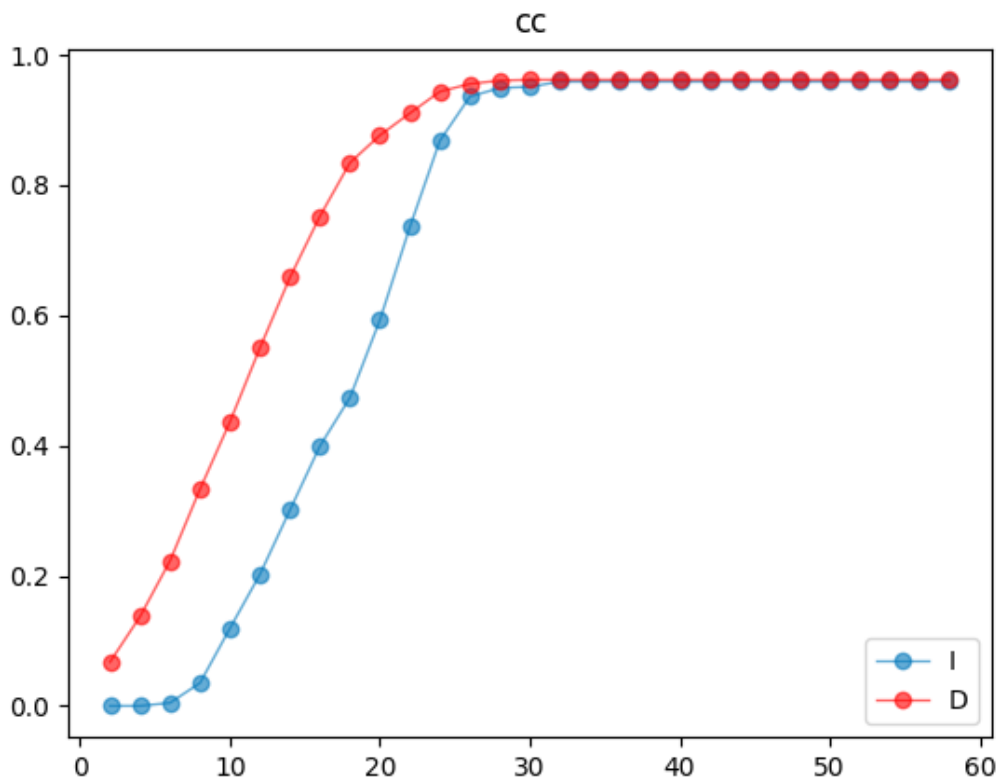
这时，我们就可以轻松的使用python中的split方法，提取我们需要的数据进行绘图了。

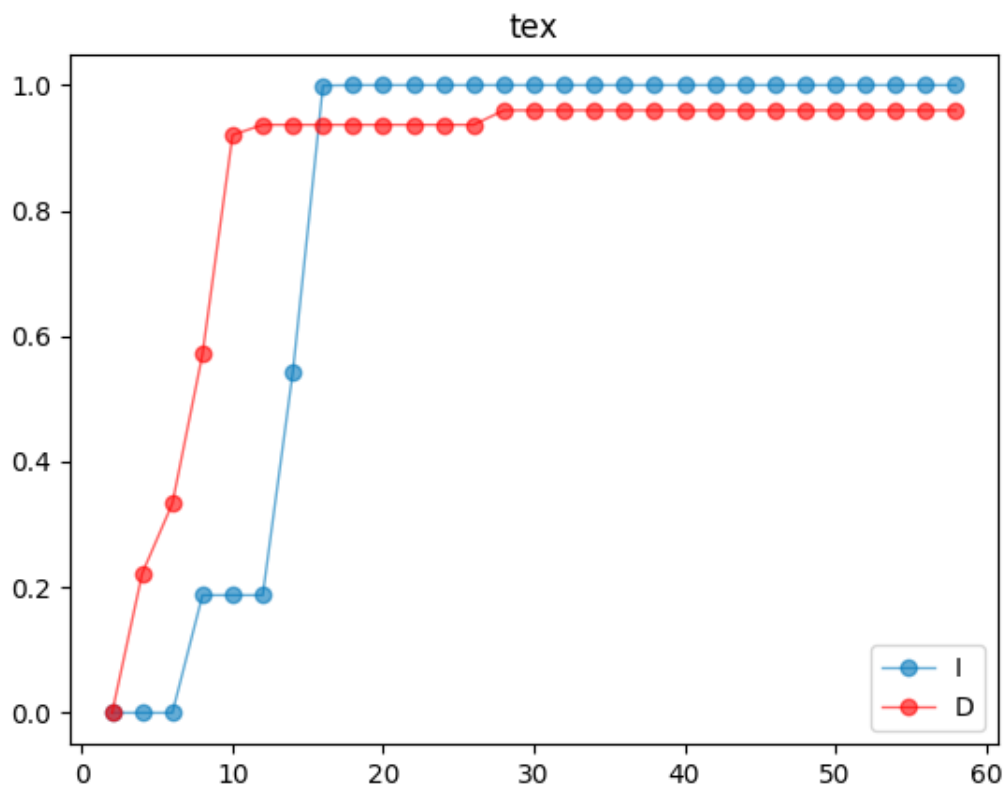
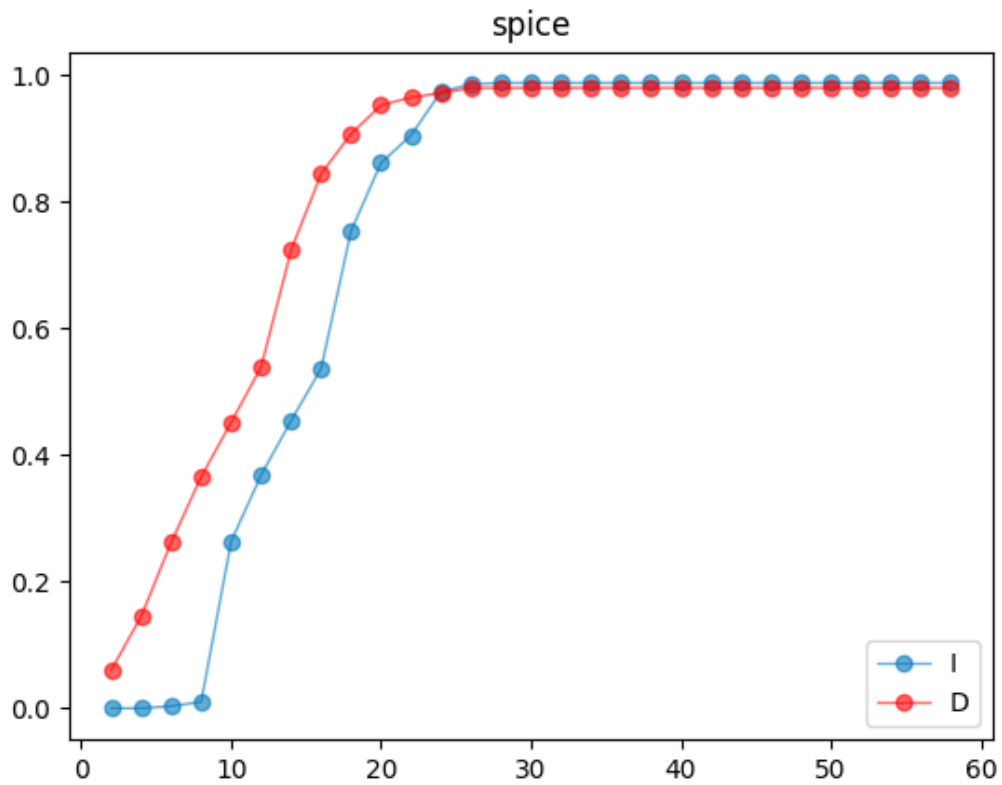
但是，一开始的时候，我直接把横坐标设置为cache size。在绘图结果中感受到了指数爆炸。



为了实现横坐标以2为底取对数，我重新定义了一个列表变量x，并且根据题目的要求对列表复制，实现了较为理想的绘图结果。因为下面的部分展示了结果，故这里不重复粘贴图片。shell测试脚本和python绘图脚本都在提交文件的eval目录下。

Working Set Characterization





说明：图表中纵轴为命中率。假设横轴的值 x ，则cache size为 2^x 大小。

1.Explain what this experiment is doing, and how it works. Also, explain the significance of the features in the hit-rate vs. cache size plots.

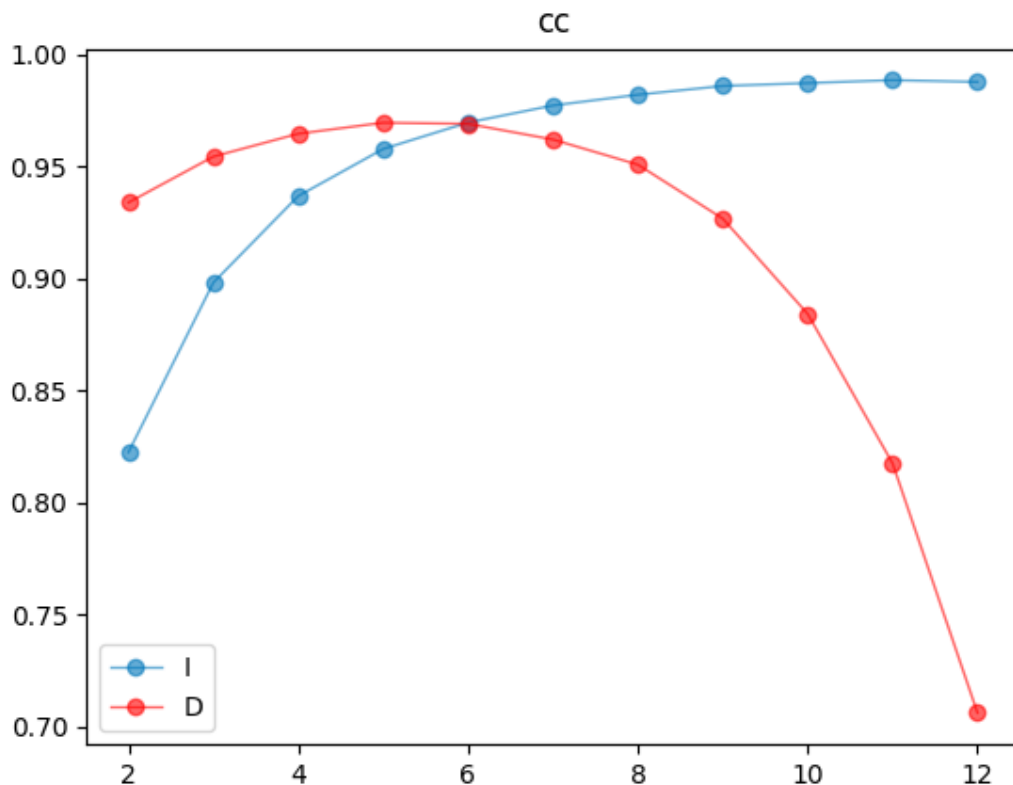
这个实验观察了cache大小对命中率的影响。显然，因为我们采取的是fully associative cache，所以当cache足够大的时候，除了冷启动时的强制缺失，其他情况下，缓存都会命中。而我们的实验结果也正说明了这点。随着cache size的增加，hit rate不断增加，接近1。

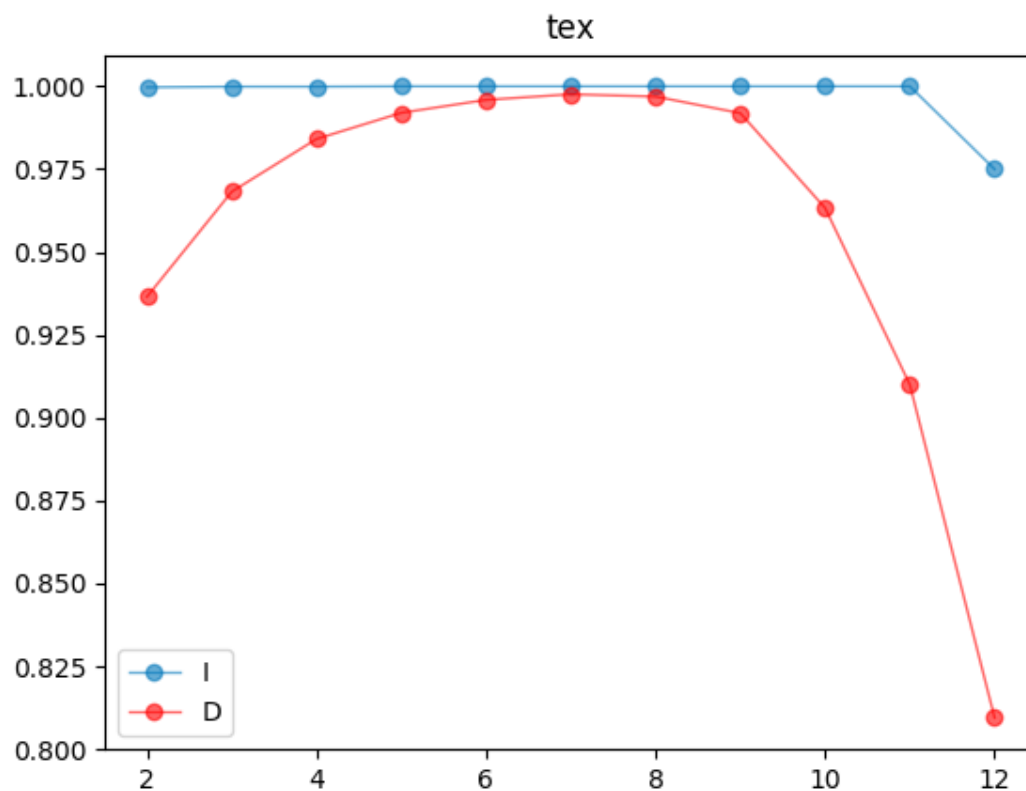
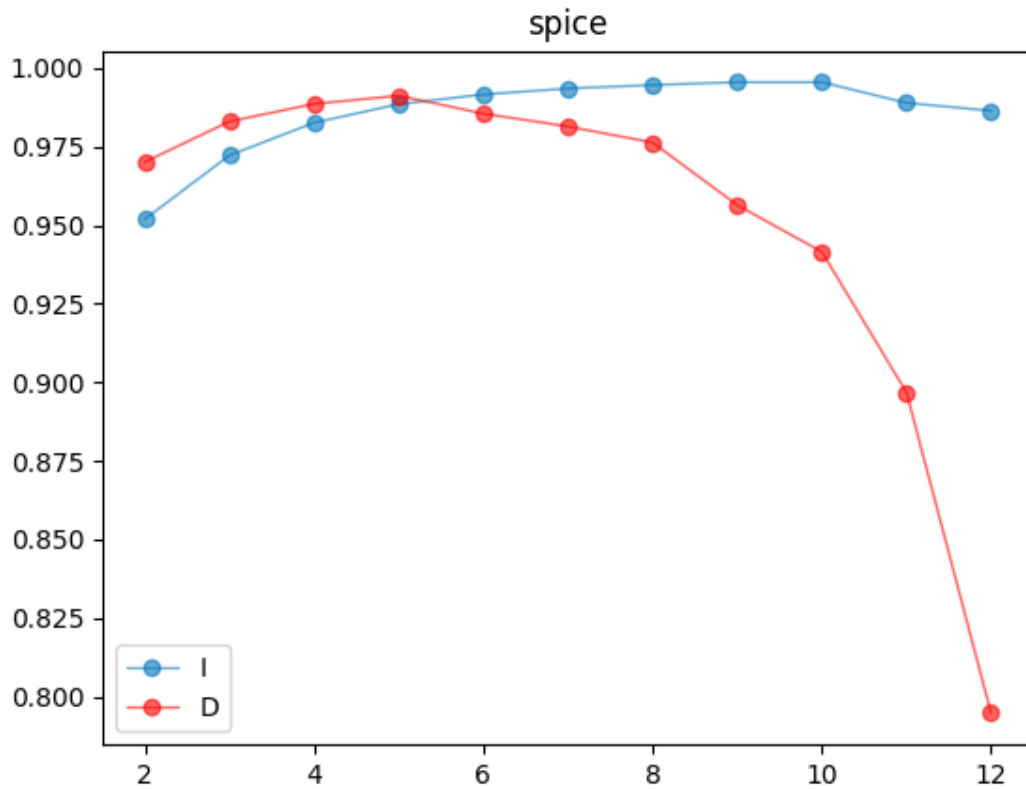
但是，当cache size足够大，大到超过指令和数据数量的数量时，则hit rate会保持一个定值不变。而且，增长是非线性的，开始时hit rate增长快，后面逐渐放缓，乃至出现之前提到的保持定值的情况。

2.What is the total instruction working set size and data working set size for each of the three sample traces?

sample trace	type	working set size
cc	D	242661
cc	I	757341
spice	D	217237
spice	I	782764
tex	D	235168
tex	I	597309

Impact of Block Size





说明：图表中纵轴为命中率。假设横轴的值x，则block size为 2^x 大小。

1.Explain why the hit rate vs. block size plot has the shape that it does. In particular, explain the relevance of spatial locality to the shape of this curve.

对于数据cache来说，block size增加时，hit rate首先会增加，但之后会降低。

对于数据cache来说，block size增加时，hit rate的变化趋势和测试数据集有关。在cc中随着block size增加，hit rate不断增加，最后有略微的降低。但是在spice上，hit rate先增加，但之后会降低。在tex上，block size从4增加到32时，hit rate有略微的增加，而后hit rate恒等于1，当block size大于2048时，hit rate降低。

开始时，随着block size的增加，同一个块中能储存的数据增加，出现cache miss时，一次会加载更多的数据到cache中，因为空间局部性，所以hit rate增加。但是，当block size大到一定程度时，cache中的block num就变小，可能出现ping pong效应，导致hit rate反而下降。

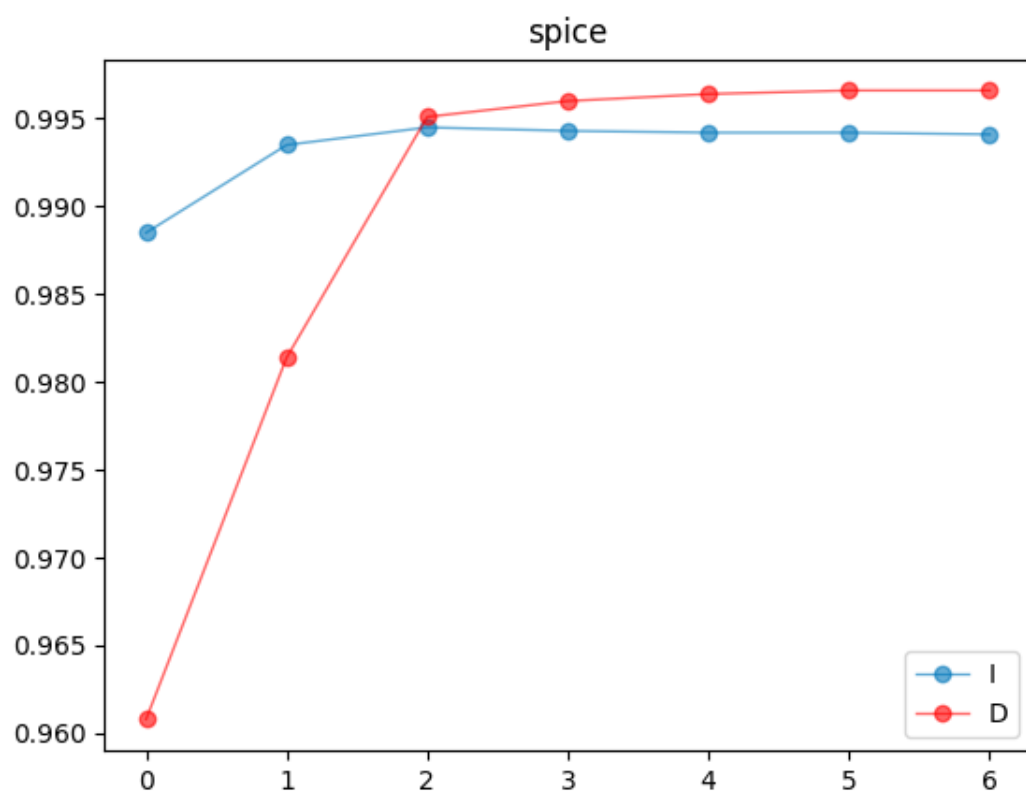
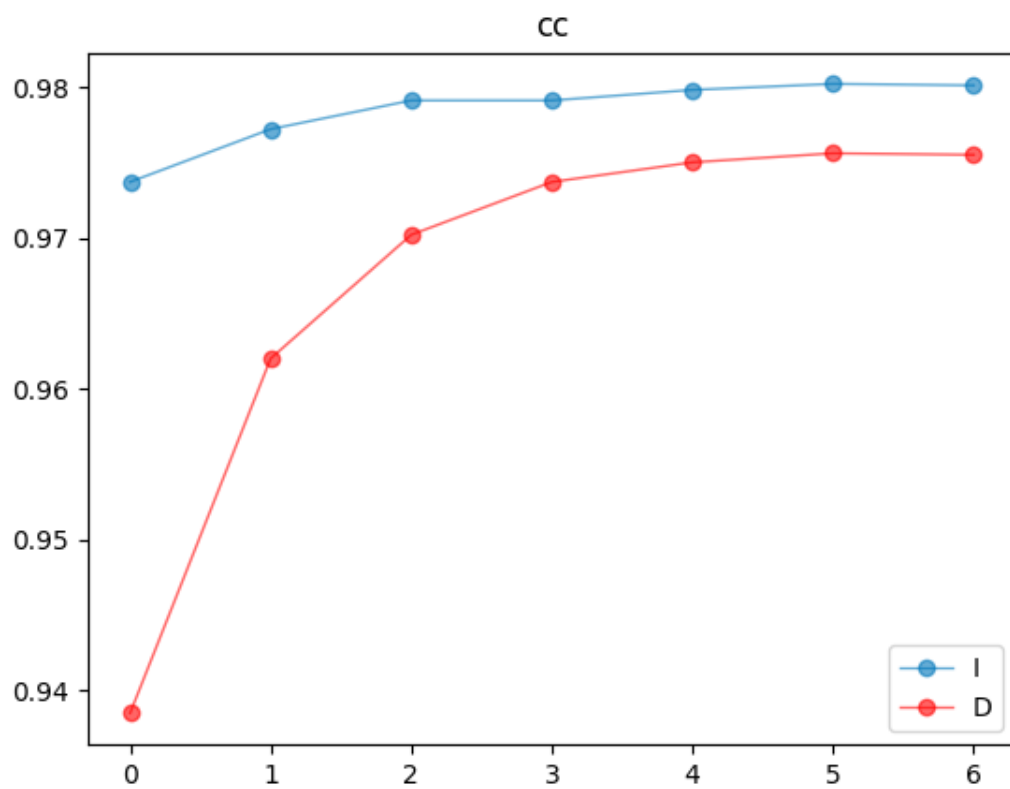
2.What is the optimal block size (consider instruction and data references separately) for each trace?

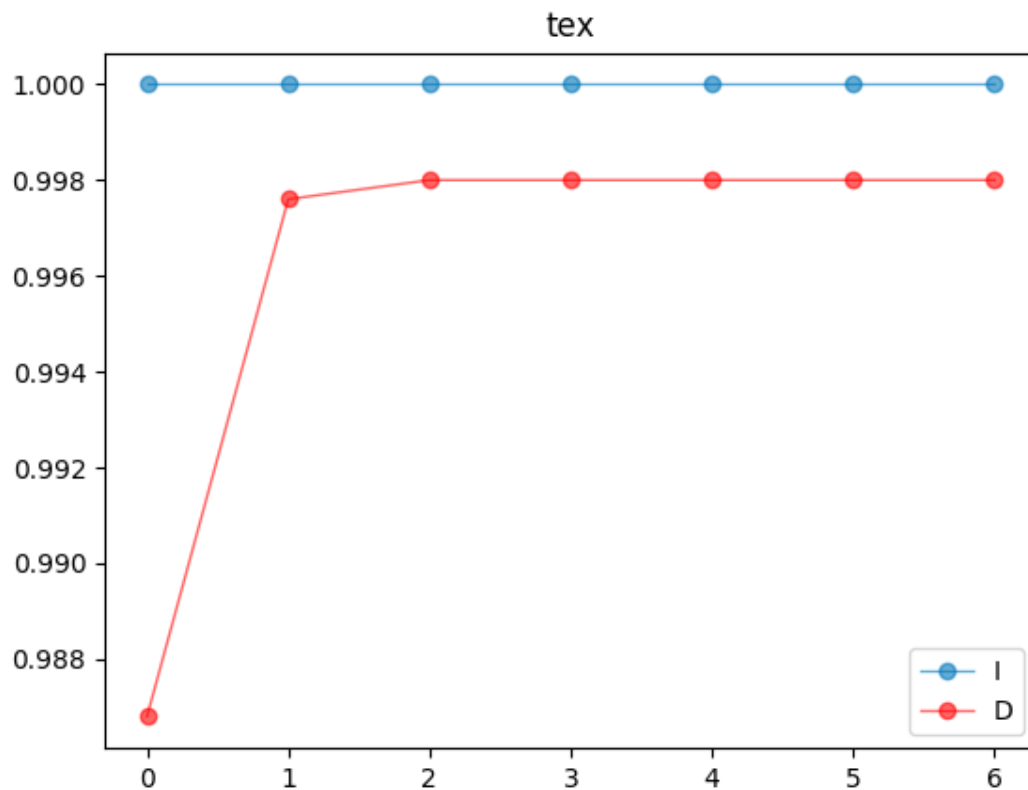
sample trace	optimal D-cache block size	optimal I-cache block size
cc	32	2048
spice	32	512 or 1024
tex	128	32 ~ 2048

3.Is the optimal block size for instruction and data references different? What does this tell you about the nature of instruction references versus data references?

根据实验数据，数据和指令缓存最优的block size是不同的。说明指令的空间局部性更好一些，I-cache可以使用更大的block size。此外，因为两者最优的block size不同，所以使用分离的cache可以使设计者灵活调整数据和指令缓存的块大小来提高缓存命中率。

Impact of Associativity





说明：图表中纵轴为命中率。假设横轴的值x，则associativity为 2^x 大小。

1.Explain why the hit rate vs. associativity plot has the shape that it does.

对于数据cache，关联性越高，hit rate越高。这是因为关联性越高，hit rate 越高。因为关联性越高，conflict misses就越少，自然的因为碰撞发生的替换也减少，所以hit rate提高。但对于指令cache，随着关联性增加，hit rate并不是不断升高，而是会有一些波动。

2.Is there a difference between the plots for instruction and data references? What does this tell you about the difference in impact of associativity on instruction versus data references?

数据cache和指令cache的命中率曲线是不同的。指令cache的hit rate总是高于数据cache。

而且，增加关联性对提高数据cache的命中率比指令cache显著。所以我们在设计cache 时，要更加注重提高数据cache的关联性。

Memory Bandwidth

采用write no allocate policy，以spice.trace.txt为例，汇总统计量结果如下

cache size	block size	associativity	write policy	allocation policy	I- misses	I- replace	d- misses	d- replece	demand fetch	copies back
8k	64	2	write through	write no allocate	6590	6462	8638	2726	151104	66538
8k	64	2	write back	write no allocate	6590	6462	8638	2726	151104	13624
8k	64	4	write through	write no allocate	6025	5897	5596	533	107296	66538

cache size	block size	associativity	write policy	allocation policy	l-misses	l-replace	d-misses	d-replece	demand fetch	copies back
8k	64	4	write back	write no allocate	6025	5897	5596	553	107296	9219
8k	128	2	write through	write no allocate	5073	5009	9940	3637	280768	66538
8k	128	2	write back	write no allocate	5073	5009	9940	3637	280768	32287
8k	128	4	write through	write no allocate	4273	4209	5858	733	162240	66538
8k	128	4	write back	write no allocate	4273	4209	5858	733	162240	13733
16k	64	2	write through	write no allocate	3006	2750	5449	324	57008	66538
16k	64	2	write back	write no allocate	3006	2750	5449	324	57008	8252
16k	64	4	write through	write no allocate	1924	1668	4984	203	38064	66538
16k	64	4	write back	write no allocate	1924	1668	4984	203	38064	7681
16k	128	2	write through	write no allocate	2490	2362	5388	310	93632	66538
16k	128	2	write back	write no allocate	2490	2362	5388	310	93632	9880
16k	128	4	write through	write no allocate	1665	1537	4893	218	64320	65538
16k	128	4	write back	write no allocate	1665	1537	4893	218	64320	9668

采用write back policy，以spice.trace.txt为例，汇总统计量结果如下

cache size	block size	associativity	write policy	allocation policy	l-misses	l-replace	d-misses	d-replece	demand fetch	copies back
8k	64	2	write back	write allocate	6590	6462	3160	3032	156000	66538
8k	64	2	write back	write no allocate	6590	6462	8638	2726	151104	13624
8k	64	4	write back	write allocate	6025	5897	875	747	110400	7296
8k	64	4	write back	write no allocate	6025	5897	5596	553	107296	9219
8k	128	2	write back	write allocate	5073	5009	4039	3975	291584	36256
8k	128	2	write back	write no allocate	5073	5009	9940	3637	280768	32287
8k	128	4	write back	write allocate	4273	4209	1075	1011	171136	14592
8k	128	4	write back	write no allocate	4273	4209	5858	733	162240	13733
16k	64	2	write back	write allocate	3006	2750	735	500	59856	6208
16k	64	2	write back	write no allocate	3006	2750	5449	324	57008	8252
16k	64	4	write back	write allocate	1924	1668	559	306	39728	5280

cache size	block size	associativity	write policy	allocation policy	I-misses	I-replace	d-misses	d-replece	demand fetch	copies back
16k	64	4	write back	write no allocate	1924	1668	4984	203	38064	7681
16k	128	2	write back	write allocate	2490	2362	604	478	99008	9088
16k	128	2	write back	write no allocate	2490	2362	5388	310	93632	9880
16k	128	4	write back	write allocate	1665	1537	407	280	66304	7200
16k	128	4	write back	write no allocate	1665	1537	4893	218	64320	9668

1.Which cache has the smaller memory traffic, the write-through cache or the write-back cache? Why?

write-back。

如果使用write through的话，每次更新cache数据的同时都需要更新memory中的数据。而使用write back的话，只有在执行LRU替换且dirty bit=1时，才会将cache中的内容写回到memory。这使得write back的copies back操作相对于write through极大的减少。

2.Is there any scenario under which your answer to the question above would flip? Explain.

当多核需要共享内存时，直写策略会优于回写策略，因为write through能够保证共享内存中的同步 concurrency，而write back则需要较为复杂的机制来实现多核共享内存的问题。

3.Which cache has the smaller memory traffic, the write-allocate or the write-no-allocate cache? Why?

write no allocate。

使用write no allocate策略的时候，如果记录不在cache中，只更新memory，不需要把数据fetch到cache中。但使用write allocate的话，则需要先把数据fetch到cache中，然后再按照write hit的情况来处理。所以使用write no allocate可以使得fetch操作相较于write allocate极大的减少。

4.Is there any scenario under which your answer to the question above would flip? Explain.

当写操作频繁时，write no allocate没有write allocate好。因为直接写memory消耗的时钟周期是远大于写cache的。而使用write allocate策略时，当我们出现write miss的时候，会把数据fetch到cache中。这样，下一次再写这个地址的数据时，我们就只需要在cache中更新，并且设置dirty bit就可以了。如果采用write no allocate策略时，举个极端的例子，我们假设对同一个地址执行了100次写操作，那么需要直接访问内存100次。而write no allocate则只需要第一次fetch的时候从内存取数据，之后99次都可以直接在cache中更新，而不需要和内存交互。