浙江大学

本科课程报告

课程名称：计算机组成与设计

设计名称：Cache Origanzation and Performance Evaluation

姓　　名：

学　　号：

学　　院：信电学院

专　　业：电子科学与技术

班　　级：

指导老师：刘鹏

2020 年 12 月 23 日

目录

1 Cache design

1.1 设计要求

本次设计要求能够动态改变 cache 的下列参数：

- Total cache size
- Block size
- Unified vs. split I- and D- caches
- Associatity
- Write back vs. write through
- Write allocate vs. no write allocate

同时要求 cache 模拟器能够追踪下列变量：

- Number of instructions references
- Number of data references
- Number of instructions misses
- Number of data misses
- Number of words fetched from memory
- Number of words copyed back to memory

1.2 概念理解

在 cache 设计前，我们需要清楚几个概念。首先是 unified 和 split I- and D- caches，前者是 data 和 instruction 共用一个 cache，后者是 data 和 instruction 使用分离的 cache，即共有 2 个 cache。然后是 write 有关的政策，write back 和 write through 政策适用于 cache hit 的情况，write allocate 和 write allocate 适用于 write miss 的情况，具体表现为：

- Cache hit：
  - write through：同时更新 cache 和 memory 相应内容
  - write back：只更新 cache 相应内容，dirty 位置 1，当 block 被替换时，才根据 dirty 位决定是否更新 memory 对应内容

- Cache miss：
  - write allocate：从 memory 取出对应 block 到 cache，然后更新 cache 相应内容
  - no write allocate：直接更新 memory 对应内容

此外，还有与 block replace 有关的 LRU policy，即替换最近最少使用的 block。这可以使用链表来实现，链表的头结点为最新使用的结点，尾结点为最近最少使用的结点，这样就可以直接替换链表的尾部。

1.3 代码设计

首先是初始化 cache 的代码，初始化 cache 要求将 cache 的 state 清零，同时将 cache 结构体内的变量设置为命令行参数相应的值。由于初始化 cache 结构体的代码基本一致，为了增强代码的重用性，我们设计了函数 ***init_cache_part*** 以初始化 cache 结构体。当数据与指令的 cache 不分离时，我们使用 c1 作为 cache，当两者分离时，我们需要使用 ***c1*** 和 ***c2***。在 cache 结构体中，***LRU_head*** 和 ***LRU_tail*** 存放的是 cache ***set*** 最新 access 的 block，当 cache 没有 access 过的时候，该值应为 ***NULL***，同时记录 ***set*** 里面有效 block 数量的 ***set_contents*** 应为 0。根据设计要求，不管数据和指令的 cache 是否分离，都需要记录有关数据和指令的 cache 状态，因此记录 cache 状态的结构体 ***cache_stat_inst*** 和 ***cache_stat_data*** 都需要初始化为 0。代码设计如下：

```
void init_cache_part(cache *c, int size)
{
    c->size = size / WORD_SIZE;
    c->associativity = cache_assoc;
    c->n_sets = c->size / (c->associativity*words_per_block);
    c->index_mask_offset = LOG2(cache_block_size);
    c->index_mask = ((1 << LOG2(c->n_sets))-1) << c->index_mask_offset;
    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;

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

```

```
void init_cache()
{
    /* initialize the cache, and cache statistics data structures */
    cache_stat_inst.accesses = 0;
    cache_stat_inst.copies_back = 0;
    cache_stat_inst.demand_fetches = 0;
    cache_stat_inst.misses = 0;
    cache_stat_inst.replacements = 0;

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

    if (!cache_split) {
        init_cache_part(&c1, cache_usize);
    } else {
        init_cache_part(&c1, cache_isize);
        init_cache_part(&c2, cache_dsize);
    }
}
```

下面是 ***perform_access*** 的设计。主要的设计思路是，先用 ***icache*** 和 ***dcache*** 指针指向对应的 cache，然后根据 ***access_type*** 来确定接下来的操作。每种类型都需要先取出 ***index*** 的值和输入地址的 ***tag*** 字段，用 ***index*** 找到 cache 对应的 ***set***，通过链表查找来确定是 cache miss 还是 cache hit，如果是 cache hit，需要将该结点删除后插入以保持其 LRU 状态为最新；如果是 cache miss，则要根据相应的 write policy 确定接下来的操作。其中 data load 和 instr load 只受 write through 和 write back 影响，而 data store 则需要考虑 write allocate 的问题，我们使用条件语句实现。由于代码太长，此处仅列出最复杂的 data store 部分：

```
case TRACE_DATA_STORE:
{
    int index = (addr&(dcache->index_mask))>>(dcache->index_mask_offset);
    unsigned addr_tag = addr>>(LOG2(dcache->n_sets)+dcache->index_mask_offset
        );
    Pcache_line p = dcache->LRU_head[index]; //需要替换的结点
    Pcache_line p_prev = p; //需要替换的结点的前一个

    while (p) {
```

```
        if (p->tag == addr_tag) {
            cache_stat_data.accesses += 1;
            if (!cache_writeback) {
                //write through时, 需要回写该word
                cache_stat_data.copies_back += 1;
            }
            //删除和插入来保持该结点为最新结点
            delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], p);
            insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], p);
            p->dirty = 1;
            break;
        } else {
            p_prev = p;
            p = p->LRU_next;
        }
    }

    if (p == NULL) {
        cache_stat_data.misses += 1;

        if (cache_writealloc) {
            //write allocate需要从内存取到该block
            Pcache_line item = (cache_line *)malloc(sizeof(cache_line));
            item->dirty = 1;
            item->LRU_next = NULL;
            item->LRU_prev = NULL;
            item->tag = addr_tag;

            if (dcache->set_contents[index] < dcache->associativity) {
                insert(&dcache->LRU_head[index], &dcache->LRU_tail[index],
                    item);
                dcache->set_contents[index] += 1;
                dcache->contents += 1;
                cache_stat_data.demand_fetches += words_per_block;
            } else {
            if (cache_writeback && p_prev->dirty == 1) {
                //wirte back和dirty位为1时, block替换时需要回写整个block
                cache_stat_data.copies_back += words_per_block;
            }
            cache_stat_data.demand_fetches += words_per_block;
            cache_stat_data.replacements += 1;
            delete(&dcache->LRU_head[index], &dcache->LRU_tail[index], p_prev
                );
```

```
            insert(&dcache->LRU_head[index], &dcache->LRU_tail[index], item);
            }
            if (!cache_writeback) {
            //write through时，回写该word
            cache_stat_data.copies_back += 1;
            }
        } else {
            //no write allocate直接将该word回写内存
            cache_stat_data.copies_back += 1;
        }
        cache_stat_data.accesses += 1;
    }
    break;
}
```

最后是 ***flush*** 的设计，该函数主要用于最后的程序结束时。主要的操作是，将 cache 中有效的 block 通过尾删法删除，如果是 write back，还需要根据 dirty 位的值来决定是否需要回写 memory。代码如下：

```
void flush()
{
    /* flush the cache */
    for (int i=0; i<c1.n_sets; i++) {
        for (int j=0; j<c1.set_contents[i]; j++) {
            if (cache_writeback && c1.LRU_tail[i]->dirty == 1)
            cache_stat_data.copies_back += words_per_block;
            delete(&c1.LRU_head[i], &c1.LRU_tail[i], c1.LRU_tail[i]);
        }
        c1.set_contents[i] = 0;
    }
    c1.contents = 0;
    for (int i=0; i<c2.n_sets; i++) {
        for (int j=0; j<c2.set_contents[i]; j++) {
            if (cache_writeback && c2.LRU_tail[i]->dirty == 1)
            cache_stat_data.copies_back += words_per_block;
            delete(&c2.LRU_head[i], &c2.LRU_tail[i], c2.LRU_tail[i]);
        }
        c2.set_contents[i] = 0;
    }
    c2.contents = 0;
}
```

1.4 代码测试

为了测试 cache 设计的正确性，我分别测试了 ***traces*** 文件下的 ***public-assoc.trace***、***public-block.trace***、***public-instr.trace***、***public-write.trace***、***spice10.trace***、***spice100.trace***、***spice1000.trace***、***spice10000.trace*** 文件，并将结果存放在 ***myOut_1*** 文件夹中。测试可以通过运行 ***runPublic.sh*** 文件实现。正确的 cache 结果在作业提供的 ***outputs*** 文件夹中。比较两个结果的差异可以通过运行脚本文件 ***compare.sh*** 实现。运行结果反馈如下：

```
hhubibi@hhubibi-virtual-machine:~/Documents/p2$ bash runPublic.sh
mkdir: cannot create directory ‘myOut_1’: File exists
-----run small test-----
-----run big test-----
all output done
hhubibi@hhubibi-virtual-machine:~/Documents/p2$ bash compare.sh
compare my cache design output with standard output
--test begin--
--test end--
test passed
```

2 Performace evaluate

2.1 Working Set Characterization

在本小节中，我们将讨论 cache performance 与 cache size 的关系。我们使用的 cache 为全关联缓存，block size 为 4 字节，采取 write back 和 write allocate policy。cache size 的初始大小为 4 字节，然后每次变为原先的 2 倍。可以通过运行脚本文件 ***workingset.sh*** 来得到输出结果，三个结果文件 ***cc_stats_1_wsc.txt***、***spice_stats_1_wsc.txt***、***tex_stats_1_wsc.txt*** 存放在 ***myOut_2*** 文件夹中。图 2.1 由 ***plot.py*** 绘制得到，展示了 cache hit rate 与 cache size 的关系，其中 x 轴为 cache size 以 2 为底的对数，y 轴为 cache hit rate。

下面进行问题的回答：

针对问题一，我们可以从图 2.1 看到，随着 cache size 的增加，hit rate 也随之增加。因为 cache 越大，它包含所需数据的可能性就越大；但是，渐近线到达 1，无论大小增加多少，hit rate 已经它达到了最大值。

针对问题二，表 2.1 展示了三个文件指令集和数据集的大小。

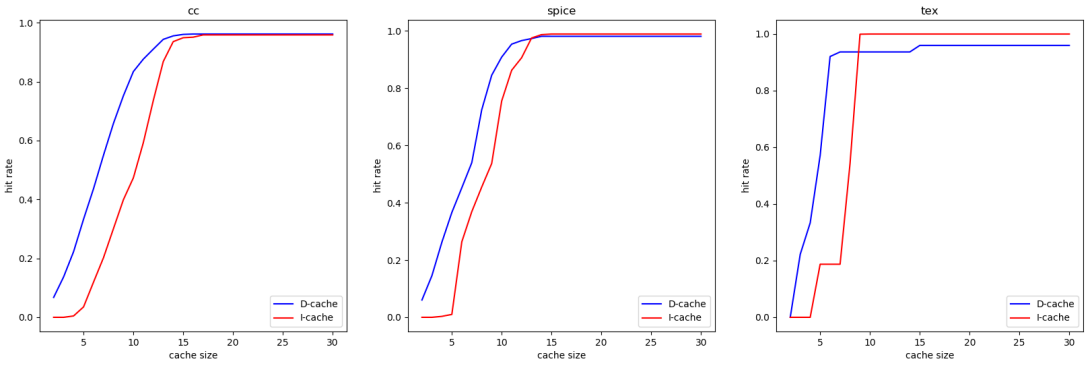

图 2.1: hit rate 与 cache size 的关系

表 2.1: 指令集和数据集的大小

file	type	working set size
cc	D	242 661
cc	I	757 341
spice	D	217 237
spice	I	782 764
tex	D	235 168
tex	I	597 309

2.2 Impact of Block Size

在本小节中，我们将讨论 cache performance 与 block size 的关系。我们采用 I 和 D 分离的 cache，大小为 8 K bytes，关联度为 2，采取 write back 和 write allocate policy。Block size 的初始大小为 4 字节，然后每次变为原先的 2 倍，直到增大为 4 K bytes。可以通过运行脚本文件 ***blocksize.sh*** 来得到输出结果，三个结果文件 ***cc_stats_2_block.txt***、***spice_stats_2_block.txt***、***tex_stats_2_block.txt*** 存放在 ***myOut_2*** 文件夹中。图 2.2 由 ***plot.py*** 绘制得到，展示了 cache hit rate 与 block size 的关系，其中 x 轴为 block size 以 2 为底的对数，y 轴为 cache hit rate。

下面进行问题的回答：

针对问题一，我们可以从图 2.2 看到，最初，block size 越大，hit rate 越高，但是速率在一定点后降低，data cache 的这种减少更为明显。其中所有三个文件的 hit rate 都从 block size = 128 左右开始急剧降低。这与空间局部性原理有关。首先，如果增加块大小，则会加载更多的存储位置以供使用；但如果块大小非常大，具有固定的缓存大小，则较大的块意味着较少的集合在同一个缓存中，因此必须进行更多替换。

针对问题二，表 2.2 展示了三个文件的较优 block size。

表 2.2: the optimal block size

file	D-block size	I-block size
`cc`	32	2 048
`spice`	32	512-1 024
`tex`	128	32-2 048

针对问题三，数据和指令缓存较优的 block size 是不同的，这告诉我们使用分离的 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。

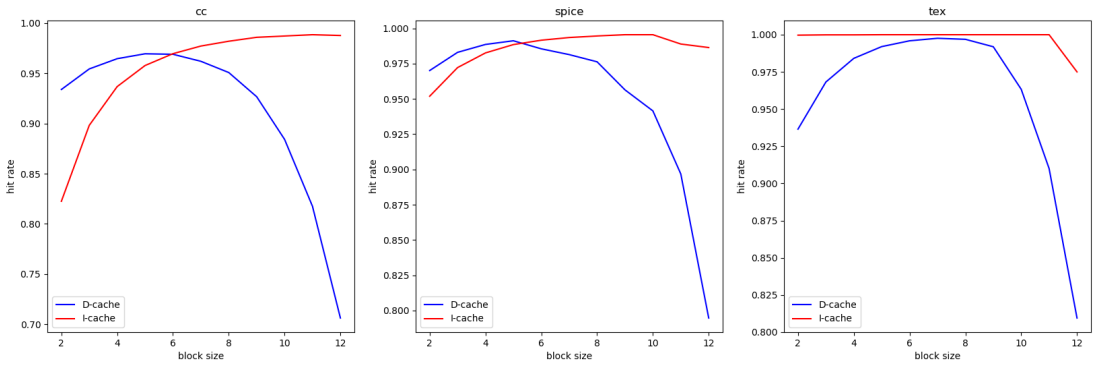

图 2.2: hit rate 与 block size 的关系

2.3 Impact of Associatity

在本小节中，我们将讨论 cache performance 与 associatity 的关系。我们采用 I 和 D 分离的 cache，大小为 8 K 字节，块大小为 128 字节，采取 write back 和 write allocate policy。关联度最初为 2，以后逐次翻倍，直到关联度变为 64。可以通过运行脚本文件 ***associatity.sh*** 来得到输出结果，三个结果文件 ***cc_stats_3_assoc.txt***、***spice_stats_3_assoc.txt***、***tex_stats_3_assoc.txt*** 存放在 ***myOut_2*** 文件夹中。图 2.2 由 ***plot.py*** 绘制得到，展示了 cache hit rate 与 associatity 的关系，其中 x 轴为 associatity 以 2 为底的对数，y 轴为 cache hit rate。

下面进行问题的回答：

针对问题一，我们可以从图 2.3 看到，associatity 越高，hit rate 越高。因为 associatity 越高，由 conflict 带来的块替换和 cache miss rate 都会减少。

针对问题二，数据和指令缓存的命中率曲线是不同的。这告诉我们在设计 cache 时，要更加注重数据缓存的关联度，因为数据间的空间局部性关系较大，提高 cache 的关联度可以明显增加 cache hit rate，而指令缓存的关联度则没那么重要。

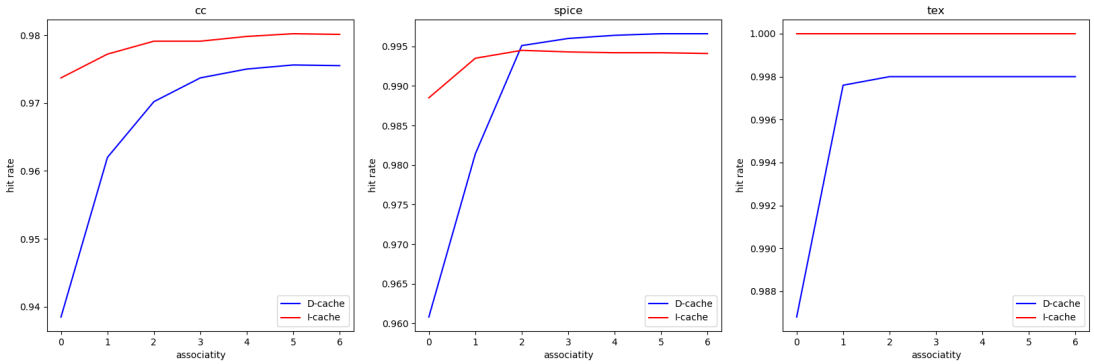

图 2.3: hit rate 与 associatity 的关系

2.4 Memory Bandwidth

在本小节中，我们将讨论 cache performance 与 write policy 的关系。我们采用 I 和 D 分离的 cache，大小为 8 K 字节或 16K 字节，块大小为 64 字节或 128 字节，关联度为 2 或 4。可以通过运行脚本文件 ***bandwidth.sh*** 来得到输出结果，三个结果文件 ***cc_stats_4_memory_bandwidth.txt***、***spice_stats_4_memory_bandwidth.txt***、***tex_stats_4_memory_bandwidth.txt*** 存放在 ***myOut_2*** 文件夹中。我们以 ***spice_stats_4_memory_bandwidth.txt*** 文件内容为例进行讨论与分析。

我们的 cache 先采用 write no allocate 政策，结果体现在图 2.3 中，下面进行问题的回答：

					Instructions		Data		Total	
CS	BS	Assoc	Write	Alloc	Misses	Repl	Misses	Repl	DF	CB
8K	64	2	WT	WNA	6590	6462	8638	2726	151104	66538
8K	64	2	WB	WNA	6590	6462	8638	2726	151104	13624
8K	64	4	WT	WNA	6025	5897	5596	553	107296	66538
8K	64	4	WB	WNA	6025	5897	5596	553	107296	9219
8K	128	2	WT	WNA	5073	5009	9940	3637	280768	66538
8K	128	2	WB	WNA	5073	5009	9940	3637	280768	32287
8K	128	4	WT	WNA	4273	4209	5858	733	162240	66538
8K	128	4	WB	WNA	4273	4209	5858	733	162240	13733
16K	64	2	WT	WNA	3006	2750	5449	324	57008	66538
16K	64	2	WB	WNA	3006	2750	5449	324	57008	8252
16K	64	4	WT	WNA	1924	1668	4984	203	38064	66538
16K	64	4	WB	WNA	1924	1668	4984	203	38064	7681
16K	128	2	WT	WNA	2490	2362	5388	310	93632	66538
16K	128	2	WB	WNA	2490	2362	5388	310	93632	9880
16K	128	4	WT	WNA	1665	1537	4893	218	64320	66538
16K	128	4	WB	WNA	1665	1537	4893	218	64320	9668

图 2.4: spice workload 结果

针对问题一，我们从图 2.3 可以看到，对于相同的块大小，缓存大小和相同的关联性，回写策略的 demand fetch 与直写策略完全相同；但是，使用回写策略可以减少 copy back。

因此回写策略具有较小的 memory traffic。

针对问题二，当多核需要共享内存时，直写策略会优于回写策略，因为直写策略能够保持内存中的东西总是最新的，而回写策略则需要较为复杂的机制来实现多核共享内存的问题。

然后我们的 cache 采用 write back 政策，结果体现在图 2.4 中，下面进行问题的回答：

					Instructions		Data		Total	
CS	BS	Assoc	Write	Alloc	Misses	Repl	Misses	Repl	DF	CB
8K	64	2	WB	WA	6590	6462	3160	3032	156000	18880
8K	64	2	WB	WNA	6590	6462	8638	2726	151104	13624
8K	64	4	WB	WA	6025	5897	875	747	110400	7296
8K	64	4	WB	WNA	6025	5897	5596	553	107296	9219
8K	128	2	WB	WA	5073	5009	4039	3975	291584	36256
8K	128	2	WB	WNA	5073	5009	9940	3637	280768	32287
8K	128	4	WB	WA	4273	4209	1075	1011	171136	14592
8K	128	4	WB	WNA	4273	4209	5858	733	162240	13733
16K	64	2	WB	WA	3006	2750	735	500	59856	6208
16K	64	2	WB	WNA	3006	2750	5449	324	57008	8252
16K	64	4	WB	WA	1924	1668	559	306	39728	5280
16K	64	4	WB	WNA	1924	1668	4984	203	38064	7681
16K	128	2	WB	WA	2490	2362	604	478	99008	9088
16K	128	2	WB	WNA	2490	2362	5388	310	93632	9880
16K	128	4	WB	WA	1665	1537	407	280	66304	7200
16K	128	4	WB	WNA	1665	1537	4893	218	64320	9668

图 2.5: spice workload 结果

针对问题一，我们从图 2.4 可以看到，对于相同的块大小，缓存大小和相同的关联性，写分配策略的 demand fetch 比写不分配策略多，但差距不大，两者的 copy back 与 cache 大小有关系，然而写分配策略具有较少的 data cache miss 和 data cache replace。因此写分配策略具有较小的 memory traffic。

针对问题二，当写频繁而 cache size 没有那么大时，write allocate 可能没有 write no allocate 效果好。