SimpleScalar cache分析

1．cache.h 文件的内容

- cache 的替换策略，包括三种 LRU, random 和 FIFO
- 整个 cache 结构的组织：
  - 表示整个cache的是*struct cache_t*
  - 接下来的次一级的是*struct cache_set_t*
  - 再接下来次一级的是 *struct cache_blk_t*

也即：首先由 cache 块组成 cache 组，然后由 cache 组形成整个 cache 结构。先看 cache_t 中的定义： 首先是给出了 cache 中的一些常用参数，这些参数虽然可以通过其他方法获取，不过这儿体现了所谓的面向对象的概念，因为在 cache_t 的定义中，就使用了函数指针；而在 cache_set 中，不难发现，每个组其中就是有若干个块组成的；

- 剩下的是其他函数了，具体碰到再看。

2．介绍 cache.c 文件的实现

- cache_create 函数
  - 首先是对传入参数的判断，组数不能小于 0，而(nsets & (nsets-1)) != 0 的作用且是判断 nsets 必须是 2 的幂次；
  - 给 cache 分配空间，注意，这儿的分配是(nsets-1)个 cache_set,这主要是因为在 cache_t 的定义中已经包含了一个 struct cache_set_t sets[1]，这也就是 SS 的高明之处；
  - 开始给 cache_t 中的一些变量赋值，其中的 CACHE_HIGHLY_ASSOC 宏判断构造的 cache 是不是相联度很大的 cache，因为如果相连度大了，那么为了效率考虑，需要采用不同的搜索方法；
  - 给 cache 中的数据分配空间，其中的(cp->balloc ? (bsize*sizeof(byte_t)) : 0)根据传入的参数，决定是不是要给 cache 分配存放数据的空间。

    现在看其中初始化 cache 部分的一段代码：

    cache_dl1 = cache_create(name, nsets, bsize, / balloc */FALSE,/* usize */0, assoc, cache_char2policy(c),dl1_access_fn, /* hit latency */1);*

    对照参数，可以知道，其中的 balloc 给出的结果是 FALSE，也就是说，SS 并不会直接给 cache 分配存放数据的空间。而在我们一般的理解中，在存 储器中会存放数据，在 cache 中也会存放一部分数据，从而提高数据存取效率，但是 SS 并没有给 cache 分配空间，cache 拥有的只是一些 tag 而 已，从而也就造成 SS 的 cache 行为模型不精确，现在被称之为玩具。
  - 接下来是按照每个组针对组内的块进行处理，即：for (bindex=0,i=0; i<nsets; i++)

    1) 针对其中的三个语句：

    cp->sets[i].way_head = NULL;

    cp->sets[i].way_tail = NULL;

    cp->sets[i].blks = CACHE_BINDEX(cp, cp->data, bindex);

    而在cache_t的定义中，我们只能看到有关组的这个定义 *struct cache_set_t sets[1];*这主要是因为在之前给cache分配空间的时候，采用了这个方式：

*cp = …… + (nsets-1)*sizeof(struct cache_set_t));*

即把(nsets-1)*sizeof(struct cache_set_t)个空间放在了整个cache空间的结尾，从而在内存空间中，前一部分是一个cache结构，其中有一个cache_set，接下来就是 *(nsets-1)个cache_set*了，从而可以使用cp->sets[i]，这也就是SS的高明之处。

2) 宏 CACHE_BINDEX 的作用是去确定每个组内的块的位置.这儿觉得 SS 弄得有点神秘了，完全可以老实点按层次实现 cache，有空看看 RSIM 和 M5 怎么实现的。不过我就是很老实实现了 cache，理解起来也方便，运行结果也蛮好……

3) 然后就是针对组内的块进行处理： *for (j=0; j<assoc; j++)* 这儿看 *blk->user_data = ……calloc(usize, sizeof(byte_t)) : NULL);*这句话，看其中的参数 usize，也是 0！

✧ cache.c 文件中最重要的函数应该就是 cache_access 函数了，现在看这个函数的实现

- 首先利用宏得到要访问addr的一些tag和se, 然后会进行判断，由于访问的字节是 1 个、2 个、4 个、8 个，所以要*(nbytes & (nbytes-1)) != 0*。而系统在存放数据的时候，是对其存放的地址是对齐的，所以 *(addr & (nbytes-1)) != 0*
- 在 cache 访问中，有时候为了节约 cache 访问时间，可以假设本次访问的 cache 块与上一次访问的 cache 块是同一个块，所以有 *if (CACHE_TAGSET(cp, addr) == cp->last_tagset),* 从而避免更多的查找时间，如果不是的话，会带来一定的开销，当然，一般说来，这个条件可以成立。计算机体系结构的发展，其实没有什么规律的，完全是运气运气出来的！如果条件不成立，就接下去寻找 cache 块。
- 先看如果在 cache 中没有找到的情况，即 cp->misses++那么，根据选择的替换策略，要选择一个被替换出去的 cache 块，这儿 *case LRU:case FIFO:*可以看出，把 LRU 和 FIFO 的替换看成是一样的，其实不是，而且 LRU 处理起来还要 10 多条语句。

  好，到此为止，找到了需要被替换出去的块。

  1) 在cache块中的数据，有三种可能：没有数据、有干净的数据、脏数据，在SS中，是把干净的数据和脏数据都认为是有效的VALID的数据，所以接下来的判断 *if (repl->status & CACHE_BLK_VALID)*就是判断cache中还有没有数据，因为新cache里面肯定是没有数据的！当有数据的时候，当然，这儿不管是干净的还是脏的了，需要被替换，这儿是模拟了存储总线的忙闲状态。

     如果其中的数据是脏数据，则明显还要写回。

  2) 当 cache 中原来没有数据时，则进行一般的处理操作。
- 如果命中 cache 块了，那么 cp->hits++

接下来就读读写写了。