报告

李晨昊 2017011466

2020 年 3 月 26 日

目录

1 运行方法

直接执行`make`即可生成可执行文件`main`。直接运行`main`，对`trace`文件夹中的所有文件执行题目中要求的各种 Cache 参数组合的测试，包括输出要求的 log 文件 (trace 文件名是写死在代码中的，不会考虑`trace`文件夹中的实际内容)。因为 Cache 组内查找和部分替换算法的运行时间都正比于关联路数，所以全相联的情形下总的运算量很大，需要等待较长时间才会得到输出。

2 Cache 布局性能分析

我在空间开销上已经几乎做到了软件实现的最优：对于缓存行的数组和替换算法的信息的数组，每个逻辑上的元素都是数目相同的u8单元，其数目是通过计算每个元素所需要的位数除以u8的大小，向上取整得到的。如果还要进一步优化，可以允许元素不使用完整的u8单元，不过这样代码的复杂程度也会进一步上升。

2.1 缺失率

测试数据如下：

Cache 布局 / 文件	直接映射			4-way 组相联			8-way 组相联			全相联		
	8	32	64	8	32	64	8	32	64	8	32	64
astar	23.40	9.84	5.27	23.28	9.63	5.01	23.28	9.63	5.00	23.26	9.59	4.97
bodytrack_1m	2.06	1.33	1.59	1.22	0.31	0.15	1.22	0.31	0.15	1.22	0.31	0.15
bzip2	4.24	1.34	0.85	4.11	1.20	0.68	4.10	1.19	0.68	4.09	1.19	0.67
canneal.uniq	3.67	2.31	1.89	2.07	1.14	0.85	1.79	0.82	0.62	1.75	0.66	0.39
gcc	6.58	2.16	1.22	6.54	2.12	1.15	6.54	2.12	1.15	6.54	2.11	1.15
mcf	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
perlbench	1.07	1.07	1.07	0.76	0.76	0.76	0.76	0.76	0.76	0.75	0.75	0.76
streamcluster	4.94	2.20	1.46	4.58	1.82	1.08	4.58	1.82	1.08	4.58	1.82	1.08
swim	4.34	4.34	4.34	2.65	2.65	2.65	2.38	2.38	2.38	0.85	1.28	2.05
twolf	1.18	0.39	0.27	1.14	0.34	0.20	1.14	0.34	0.20	1.14	0.34	0.20

观察数据可以看出，随着相联度的上升，缺失率有一定下降，但是不是很明显；随着块大小的上升，缺失率一般会明显下降。当然，缺失率不会一直随着块大小的上升而下降，我尝试了一下更大的块大小，缺失率会先下降再上升，只是要求测试的区间全部包含在了缺失率下降的区间中。

不同的文件对缺失率也有明显的影响，对于某些文件，Cache 布局对缺失率的影响很小。

2.2 空间开销

Cache 的布局会影响缓存行大小，缓存行个数；替换算法所需的信息单元大小，信息单元个数。不过因为实现的限制，每个逻辑上的数组元素都是由一定数量的u8单元保存的，所以数组元素的位大小变化时，不一定能在实际申请的内存中体现出来。

具体来说，这些值的计算方式分别为：

$$\begin{aligned}
\text{缓存行大小 (只考虑元信息)} &= 64 - (\log_2 \text{缓存容量} - \log_2 \text{关联路数}) + 1 + [\text{写回}] \\
\text{缓存行个数} &= \text{缓存容量}/\text{块大小} \\
\text{信息单元大小 (LRU)} &= \text{关联路数} \times \log_2 \text{关联路数} \\
\text{信息单元个数} &= \text{缓存容量}/\text{块大小}/\text{关联路数}
\end{aligned}$$

值得注意的是，缓存行大小不受块大小的影响。这是因为块大小会同时影响地址中的索引和块内偏移两个部分，二者的影响抵消了。

容易看出，单个块越大，关联度越低，缓存行总空间和信息单元总空间越小。

3 替换算法性能分析

除了要求的三个算法之外，我还实现了 LFU 替换算法。我对原始的算法进行了一点改动：当一行的计数值达到关联度数时，把同一组内的所有计数值都除以 2。这样可以避免原始算法的一个问题：历史上访问次数很多而最近没有被访问的行可能迟迟不被换出，同时控制了计数值的最大值 (关联路数 − 1)，可以减少算法的空间消耗。

3.1 算法实现

维护信息 (命中一行时)：

1. LRU：对于一组内的每个索引值，如果它比命中的行的索引值小则加一，相等则置零，否则不变。这就是模拟把这一行从栈中抽出而移到栈顶，在它上方的行都向下移动一个位置，在它下方的行都不动。

```
// read_bits和write_bits分别实现从指针p偏移start个bit处读出/写入长度为
   len的比特序列，这个实现要求len < 16
// _bextr_u32是BMI指令集提供的intrinsic，任何常见的CPU都应该支持它。
   _bextr_u32(x, start, len)读出32位整数的比特区间[start, start + len)
   的值
u32 read_bits(const u8 *p, u32 start, u32 len) {
  p += (start >> 4) << 1, start &= 15;
  return _bextr_u32(*(u32 *) p, start, len);
}
void write_bits(u8 *p, u32 start, u32 len, u32 val) {
  p += (start >> 4) << 1, start &= 15;
  *p1 = *p1 ^ _bextr_u32(*p1, start, len) << start | val << start;
```

```
}
...
u32 cur = read_bits(p, i * lg_nway, lg_nway); // 读出命中的行的索引值
for (u32 j = 0; j < nway; ++j) {
  // 读出某行的索引值，计算新的索引值并存入
  u32 val = read_bits(p, j * lg_nway, lg_nway);
  u32 new_ = val < cur ? val + 1 : val == cur ? 0 : val;
  write_bits(p, j * lg_nway, lg_nway, new_);
}
```

2. TREE：在一棵数组表示的完美二叉树上，从根走到命中的行对应的叶子节点，每个中间节点的值都置为表示指向另一侧的值。

```
// read_bit和write_bit是常规的bitset的实现，没有什么必要列出来。从功能上来说它们可以用read_bits和write_bits来实现，不过单独实现它们性能更好一些
void access_tree(u8 *p, u32 way) {
  u32 idx = 0;
  // ~j >> 31和有符号整数的j >= 0语义相同，只是我基本一直都在用无符号整数，所以就不想改成有符号整数了
  for (u32 j = lg_nway - 1; ~j >> 31; --j) {
    bool right = way >> j & 1; // right表示是否往右走
    write_bit(p, idx, !right); // 若往右走，"告诉别人往左走"
    idx = (idx << 1) + 1 + right; // 计算往右走的下标
  }
}
...
access_tree(p, i); // 在树上标记刚才访问的行
```

3. RAND：无操作。

4. LFU：将命中的行的计数值加一，若加一后的值等于关联度数，则将一行内所有计数值都折半。代码比较简单，这里就不分析了。

替换动作：

1. LRU：选择索引值等于 关联路数 -1 的行替换掉。同时将所有的索引值都加 1(相当于

整体向下移动一个单位)，将替换掉的这一行的索引值重新置为 0(表示它是刚刚访问的)。

```
// fill将一行标记为valid
// 根据计算，一行的位数(只考虑元数据)一定<= 64，所以可以直接操作u64
void fill(u32 line_base, u32 way, u64 tag) {
  u64 *pline = (u64 *) &lines[line_base + way * linesz];
  // 这样只会修改标记为和tag，其余位保持不变(因为这个u64并不都属于这行)
  *pline = 1 | tag << 1 | *pline ^ _bextr_u64(*pline, 0, linesz * 8);
}
...
for (u32 i = 0; i < nway; ++i) {
  u32 val = read_bits(p, i * lg_nway, lg_nway);
  u32 new_ = val + 1;
  // free_way是前面遍历中找到的未填的行，若不存在则是-1
  if (val == nway - 1 || i == free_way) {
    new_ = 0, fill(line_base, i, tag);
  }
  write_bits(p, i * lg_nway, lg_nway, new_);
}
```

2. TREE：在一颗数组表示的完美二叉树上，从树根按照每个节点的值的指向，依次走到一个叶子节点，把它替换掉。同时使用与维护信息时一样的算法，从根节点走到这个叶子并标记沿途的节点。

```
if (free_way == -1u) {
  free_way = 0;
  for (u32 i = lg_nway - 1; ~i >> 31; --i) {
    // 按照节点的指示走到一个叶子节点
    free_way = (free_way << 1) + 1 + read_bit(p, free_way);
  }
  free_way -= nway - 1; // 需要注意叶子节点的下标并不直接对应行的索引，
   差一个偏移量
}
access_tree(p, free_way);
fill(line_base, free_way, tag);
```

3. RAND：随机选择一行替换掉。

4. LFU：选择计数值最小的行替换掉，同时把这一行的计数值重新置为 1(表示它最近访问过一次)。代码比较简单，这里就不分析了。

3.2 缺失率

测试数据如下：

替换算法 / 文件	LRU	TREE	RAND	LFU
astar	23.28	23.29	23.22	23.23
bodytrack_1m	1.22	1.22	1.22	1.25
bzip2	4.10	4.09	4.12	4.12
canneal.uniq	1.79	1.78	1.79	2.53
gcc	6.54	6.54	6.58	6.54
mcf	0.25	0.25	0.25	0.25
perlbench	0.76	0.77	0.83	0.86
streamcluster	4.58	4.58	4.60	4.58
swim	2.38	2.48	3.01	2.76
twolf	1.14	1.14	1.14	1.14

从数据可以看出，在这个给定的 Cache 布局下，LRU 算法的表现总体来看最好，TREE 算法那比它差一些，但是也很接近，这是符合预期的。而我实现 LFU 的缺失率表现比较一般。不过我还测试了块大小更大的情形，LFU 的相对表现会变好很多，甚至在很多测例中达到最好。

3.3 空间开销

- LRU：每组需要 关联路数 $\times \log_2$ 关联路数 位的信息。
- TREE：每组需要 关联路数 -1 位的信息。
- RAND：代码层面上不保存任何信息，看起来是零空间开销的。如果实际在硬件上实现，也可以只用常数空间维护一个随机数种子。
- LFU：每组需要 关联路数 $\times \log_2$ 关联路数 位的信息，与 LRU 算法一致。

4 写策略性能分析

4.1 缺失率

如果 Cache 替换算法没有考虑写回引入的 dirty 位，那么写策略选择写回还是写直达对 Cache 的缺失率没有任何影响。因此这里只考虑写分配与写不分配。

测试数据如下：

写策略 / 文件	写分配	写不分配
astar	23.28	34.50
bodytrack_1m	1.22	8.67
bzip2	4.10	8.67
canneal.uniq	1.79	4.66
gcc	6.54	9.61
mcf	0.25	0.25
perlbench	0.76	0.76
streamcluster	4.58	11.15
swim	2.38	2.38
twolf	1.14	1.45

可以看出，除了少数 trace，写分配都的缺失率明显优于写不分配的确实率，这是符合预期的。

4.2 空间开销

写策略选择写分配还是写不分配对空间开销没有影响。写策略选择写回时，每个缓存行的元数据需要多一个 dirty 位来记录这个缓存行是否被写过，用于后续换出时决定是否需要写内存。