Cache系统组织与设计

郑凯文

2021 年 4 月 2 日

1 代码说明

1.1 运行方式

代码位于`main.cpp`，提供了Makefile，可直接进行make。运行后程序默认计算并输出所有要求的缺失率数据，并将要求的访问Log输出到文件。加载的trace、输出的log文件名硬编码在全局常量中，测试的参数硬编码在`main()`函数中，可进行相应修改。

跑完所有测试在我的WSL2中大约需要140s，我对于用到的位运算操作进行了一定的优化并封装成函数。

1.2 优化手段

由于所有元数据都要求用u8数组储存，模拟过程中会频繁用到“存取一段内存中连续范围内的位（或某一位）”的操作。存取某一位很容易用位运算做到：

```
inline void setBit(u8 *bitmap, u32 pos, bool val)
{
    bitmap[pos >> 3] = val ? (bitmap[pos >> 3] | (1u << (pos & 7)))
                           : (bitmap[pos >> 3] & ~(1u << (pos & 7)));
}

inline bool getBit(const u8 *bitmap, u32 pos)
{
    return bitmap[pos >> 3] >> (pos & 7) & 1;
}
```

但存取连续区间内的位会显得稍微麻烦。当然朴素的做法是一位一位地调用上面函数存取，不过显然可以利用区间连续的特点，将位于同一u8内的位一并获取，思路是特殊处理首尾的u8，中间的u8全部并入。代码如下：

```
inline u64 getBits(const u8 *bitmap, u32 start, u32 len)
{
    u32 head_arr_pos = start >> 3;
    u32 tail_arr_pos = (start + len) >> 3;
    u32 head_bit_pos = start & 7;
    u32 tail_bit_pos = (start + len) & 7;
    if (head_arr_pos == tail_arr_pos)
    {
        return (bitmap[head_arr_pos] >> head_bit_pos) & ((1u << len) - 1);
    }
    u64 res = 0;
    res += bitmap[tail_arr_pos] & ((1u << tail_bit_pos) - 1);
    for (u32 i = tail_arr_pos - 1; i > head_arr_pos; i--)
    {
        res <<= 8;
        res += bitmap[i];
    }
    res <<= 8 - head_bit_pos;
    res += bitmap[head_arr_pos] >> head_bit_pos;
    return res;
}

inline void setBits(u8 *bitmap, u32 start, u32 len, u64 data)
{
    u32 head_arr_pos = start >> 3;
    u32 tail_arr_pos = (start + len) >> 3;
    u32 head_bit_pos = start & 7;
    u32 tail_bit_pos = (start + len) & 7;
    if (head_arr_pos == tail_arr_pos)
    {
        bitmap[head_arr_pos] = (bitmap[head_arr_pos] & ((1u << head_bit_pos) - 1))
            | (data << head_bit_pos) | ((bitmap[head_arr_pos] >> tail_bit_pos) << tail_bit_pos);
        return;
    }
    bitmap[head_arr_pos] = (bitmap[head_arr_pos] & ((1u << head_bit_pos) - 1))
                        | ((data & ((1u << (8 - head_bit_pos)) - 1)) << head_bit_pos);
    data >>= (8 - head_bit_pos);
    for (u32 i = head_arr_pos + 1; i < tail_arr_pos; i++)
    {
        bitmap[i] = data & ((1u << 8) - 1);
        data >>= 8;
```

```
    }
    bitmap[tail_arr_pos] = ((bitmap[tail_arr_pos] >> tail_bit_pos) << tail_bit_pos)
                                                | (data & ((1u << tail_bit_pos) - 1));
}
```

确实显得丑了一些，但这也是为了处理边界情况。总之，在调试了很久后，终于修复了大部分bug，现在这两个函数已经完全可以胜任所有测试了。在块大小为8B、全相联的情况下，跑一个trace的时间从两分多钟减小到二十多秒，有7~8倍的常数级优化。

2 不同组织方式

2.1 Cache块元数据储存操作方式与元数据开销

按照指导书中要求的组织方式，每个Cache line的元数据从低位到高位分别为Valid位（1位）、Tag、Dirty位（1位，只有写回时有）。Tag的位数计算方式如下：块内偏移位数=$\log_2$块大小，索引位数=$\log_2$组数，组数=Cache大小/路数/块大小，Tag位数=地址长度-块内偏移位数-索引位数。这样，总的元数据大小（Cache布局相关，以位记）=(Tag位数+2（或1）)×组数×路数。

与替换策略相关的元数据不同，由于每个Cache line都对应一块Cache布局相关的元数据，我将每个Cache line的元数据对齐到u8。我还定义了Line类型对每一块meta进行包装，并提供了读和写的成员函数：

```
struct Line
{
    bool valid, dirty;
    u64 tag;
};

Line readLine(const u8 *src)
{
    Line line;
    u64 meta = getBits(src, 0, write_back + tag_len + 1);
    line.valid = meta & 1;
    line.tag = (meta >> 1) & ((1ull << tag_len) - 1);
    if (write_back)
        line.dirty = (meta >> (1 + tag_len)) & 1;
    return line;
}

void writeLine(u8 *dst, Line line)
{
    u64 meta
```

```
    = write_back ? (u64)line.valid | (line.tag << 1) | ((u64)line.dirty << (1 + tag_len))
                         : (u64)line.valid | (line.tag << 1);
    setBits(dst, 0, write_back + tag_len + 1, meta);
}
```

可以看出其元数据布局是遵从要求的。同时，在写回法中，虽然Dirty位在模拟中不会被用到（对缺失率无影响），但我还是对其进行了维护（见代码注释）。

2.2 缺失率

表 1: 固定替换策略（二叉树替换算法），固定写策略（写分配+写回）的前提下，不同Cache布局下的缺失率（%）

组织方式	直接映射			4路组相联			8路组相联			全相联		
	8B	32B	64B	8B	32B	64B	8B	32B	64B	8B	32B	64B
trace1	4.94	2.20	1.46	4.58	1.82	1.08	4.58	1.82	1.08	4.58	1.82	1.08
trace2	2.06	1.33	1.59	1.22	0.31	0.15	1.22	0.31	0.15	1.22	0.31	0.15
trace3	23.40	9.84	5.27	23.28	9.63	5.01	23.29	9.63	5.00	23.26	9.60	4.98
trace4	3.67	2.31	1.89	2.05	1.09	0.82	1.78	0.80	0.59	1.75	0.66	0.39

可以看出，相联度的增加和块大小的增大都会降低缺失率，且块大小的影响更大一些。结合课件上缺失率随块大小的变化曲线，测试的块大小应处于下降区段，块大小过大会增加冲突缺失，使缺失率增大。同时，相联度过大没有应用价值，从全相联的运行时间可以体会到。

3 不同替换策略

3.1 替换策略及其元数据储存和开销

除指导书中的替换算法外，我主要参考了论文`CRC: Protected LRU algorithm`，并尝试不同的参数组合和思路组合作为不同策略进行实验。

3.1.1 LRU

我实现LRU是用“固定位置记录路”的方式模拟栈。栈中每个元素的长度是$\log_2$路数，代表路的编号，从低位到高位是栈顶到栈底的顺序。命中时，寻找路在栈中的位置，将更低位置的元素整体向高位移动，命中的位置放在最低位置（栈顶）。未命中时，如果需要装入新块（读或写分配），若有空闲路则将其编号压入栈顶，否则选择栈底的路编号进行替换，并对栈底进行等同于命中的上移操作。（见`search_stack()`、`push_stack()`、`flow_up()`函数）

每组需要路数× $\log_2$路数的元数据，总的元数据大小为组数×路数× $\log_2$路数（组数计算公式见上节）。

3.1.2 TREE

叶子节点个数为路数，中间节点个数为路数-1，同时需要1位Valid位，因此一组需要长度为路数的元数据，总的元数据大小为组数×路数。我将Valid位放在最低位，从第1位开始存储二叉树，这样，编号为`i`的节点的左孩子为`i<<1`，右孩子为`(i<<1)+1`，以此从根节点向下走。每次访问时，沿途将标记反向（见`reverse_path()`、`follow_tree()`函数）。装入块时，若空闲路的编号为路数-1，代表组即将被填满，于是将Valid位置为1。

3.1.3 LFU

LFU的思想很简单，记录一组内每个Cache line被访问的次数，每次替换时选择访问次数最少的。但实现时需要考虑如何选择计数器的位数，我Google了很久也没找到相关的论述。于是我采用了不同的策略。LFU表示采用足够长的计数器位数以避免溢出，我选用了20；LFU_SMALL表示采用较短的计数器位数，并使其自由溢出，我选用了3；LFU_SMALL_PROTECT的计数器位数也是3，但借鉴了论文中Protected LRU计数器的维护方法：若某次访问命中后对应计数器即将溢出，则将一组内所有计数器的值右移一位（减半）。（见`lfu_hit()`、`find_lfu()`函数）

每组需要路数×计数器位数的元数据，总的元数据大小为组数×路数×计数器位数。

3.1.4 PROTECTED_LRU

PROTECTED_LRU将LRU和LFU的思想结合起来，在大部分保持LRU策略的基础上，同时对每路维护一个计数器记录访问次数（我将LRU栈元素放在低位，计数器放在高位）。在进行替换时，提供“保护”：设定要保护的路数n，使得访问次数最多的n路不被替换。计数器溢出时的处理见前述。（见`search_protected_stack()`函数，同时`push_stack()`、`flow_up()`函数兼顾了对其的处理）

每组需要的元数据大小为路数×(计数器位数+ $\log_2$路数)，总的元数据大小为组数×路数×(计数器位数+ $\log_2$路数)。我选取计数器位数为10，PROTECTED_LRU_1保护1路，PROTECTED_LRU_2保护一半数量的路。

3.2 缺失率

表 2: 固定Cache布局（块大小8B，8路组相联），固定写策略（写分配+写回）的前提下，不同替换策略的缺失率（%）

替换算法	LRU	TREE	LFU	LFU_S	LFU_S_P	P_LRU_1	P_LRU_2
trace1	4.58	4.58	4.58	4.61	4.58	4.58	4.58
trace2	1.22	1.22	1.22	1.22	1.22	1.22	1.22
trace3	23.28	23.29	23.22	23.29	23.20	23.22	23.22
trace4	1.79	1.78	1.79	1.79	1.79	1.79	1.81

可以看到LRU和TREE不分伯仲，前者在trace3好一点，后者在trace4好一点。这是有些令人意外的，因为TREE的策略很粗暴，看似记录的状态少一些，时间复杂度也是LRU的对数级别，但却能达到类似的缺失率。我尝试了随机替换的方法，也比有策略的替换差不到哪去，看起来这个数据集上替换策略的影响比较小。LFU在trace3上表现很好，但计数器有20位过于浪费资源；LFU_SMALL减小到3位，但由于溢出问题，缺失率整体都变高了；LFU_SMALL_PROTECT采用溢出减半策略后，效果极好，trace3上的表现甚至达到最好，用部分时间复杂度（溢出时遍历整组）换取了空间复杂度。溢出减半的策略可以频繁更新计数器，防止某计数过大但长期未访问的项一直占用Cache。PROTECTED_LRU_1和PROTECTED_LRU_2在trace3上有效果，在trace4上稍差，且PROTECTED_LRU_1比起PROTECTED_LRU_2更接近LRU的缺失率，这是因为前者保护的路数更少，极限情况下，保护0路便退化为LRU。从算法特点来说，LRU不适用于周期性数据循环访问，LFU不适用于访问次数多的数据长期不访问，Protected LRU是二者的折衷。

4 不同写策略

4.1 元数据开销

写回法需要在Cache块的元数据中增加一位Dirty位，写直达法则不需要。写分配和写不分配对元数据开销没有影响。

4.2 缺失率

表 3: 固定Cache布局（块大小8B，8路组相联），固定替换策略（二叉树替换策略）的前提下，不同写策略的缺失率（%）

写策略	写分配	写不分配
trace1	4.58	11.15
trace2	1.22	8.67
trace3	23.29	34.50
trace4	1.78	4.66

由于本次实验只是模拟Cache以得到命中缺失情况，不需要考虑访存情况，且替换算法没有用到Dirty位，因此写回或写直达对缺失率没有影响。上表列出了写分配与否对缺失率的影响，写分配远远好于写不分配。写不分配适用的场合应该是一次写操作访问的内存长期不会再被访问，但不太符合真实环境。