计算机体系结构gem5实验三实验报告

PB21111681 朱炜荣

一、NMRU的实现方法和修改配置的方法

1. 按照实验文档中给出的示例做法，先创建（复制粘贴）得到两个文件`ru_rp.hh`和`nmru_rp.hh`，首先所做的修改为将复制得到的文件中，所有包含LRU的部分修改成NMRU。
2. 编辑`/src/mem/cache/replacement_policies`路径下的`ReplacementPolicies.py`，添加：

```
class NMRURP(BaseReplacementPolicy):
    type = 'NMRURP'
    cxx_class = 'gem5::replacement_policy::NMRU'
    cxx_header = 'mem/cache/replacement_policies/nmru_rp.hh'
```

3. 然后修改 SimObject 所在目录中的 SConscipt 文件。将以下语句添加到`/src/mem/cache/replacement_policies/SConscript`：

```
Source('nmru_rp.cc')
```

同时修改SimObject表格中的内容，向其中添加新的元素‘NMRURP’

4. 修改`nmru_rp.cc`中的选择替换逻辑（即getVictim函数），具体代码如下：

```
ReplaceableEntry*
NMRU::getVictim(const ReplacementCandidates& candidates) const
{
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);

    // Visit all candidates to find victim
    ReplaceableEntry* last = candidates[0];
    for (const auto& candidate : candidates) {
        // Update victim entry if necessary
        if (std::static_pointer_cast<NMRUReplData>(
                    candidate->replacementData)->lastTouchTick <
                std::static_pointer_cast<NMRUReplData>(
                    last->replacementData)->lastTouchTick) {
            last = candidate;
        }
    }
    ReplaceableEntry *victim = candidates[0];

    if(candidates.size() > 1){
        int victimIndex;
        std::random_device rd;  // 用于获取种子
        std::mt19937 gen(rd()); // 使用Mersenne Twister算法生成随机数
        std::uniform_int_distribution<> dis(0, candidates.size()-1); // 定义
分布范围
        do{
            victimIndex = dis(gen);
```

```
        }while(candidates[victimIndex] == last);
        victim = candidates[victimIndex];
    }

    for (const auto& candidate : candidates) {
        if (std::static_pointer_cast<NMRUReplData>(
        candidate->replacementData)->lastTouchTick == Tick(0)) {
            victim = candidate;
            break;
        }
    }

    return victim;
}
```

其中，代码新include了一个函数库random，上述代码的逻辑为首先遍历candidates数组，找到最新使用过的设为last，然后只要candidates的大小大于1，我们就随机从candidates中选择一个，如果恰好选到了last则重新再抽一次直到选到一个不是last的。也就完成了随机选择一个非最近使用的替换策略。

5. 编译完成之后，再`se.py`中添加新的语句来手动设置选择使用的更新策略，这里我选择使用的语句为：

```
system.cpu[0].dcache.replacement_policy = NMRURP() || LIPRP() || RandomRP()
system.l2.replacement_policy = NMRURP() || LIPRP() || RandomRP()
```

二、模拟结果展示和分析

对于相同的benchmark mm，不同配置下的numCycles如下所示：

	Random	NMRU	LIP
4	3501193	3501179	3501177
8	3501171	3501162	3501178
16	3501150	3501172	3501224

对应的dCache缓存失效率如下所示：

	Random	NMRU	LIP
4	0.071701	0.059146	0.099742
8	0.068853	0.062425	0.140563
16	0.068370	0.063753	0.143493

通过纵向对比不同替换策略，我们发现，对于Random策略来说，组相联程度增加并没有过于影响到程序整体的性能。这是因为对于随机替换而言，不同的组相联对于随机选择的影响并不大，每个块都是以等概率的方法被选中替换的。性能在变好的原因可能是组相联度增加后，访问数据的规律性没有那么强了，从而使得随机替换策略性能上升。

而对于NMRU来说，随着组相联的增加性能先上升，然后又有所下降。NMRU的替换策略为从最近没有使用的块中随机选择一个进行替换，由于不能像LRU那样准确替换掉最近没有使用的块，所以在做替换时有可能换掉未来即将要用的块。随着组相联程度增加，程序的内存局部性体现在缓存中，数据之间分布更加分散，所以性能会因此下降，这一点也可以从DCache的失效率看出。但是运行程序的总性能并不是直接下降，我们发现l2Cache的失效率反而下降了。也就是随着组相联NMRU虽然换走“最关键”的那些块，但是实际上还是换掉了一些“较为重要”的块，从而引发了更多的l2失效。

LIP反而随着组相联的增加性能下降。因为LIP的替换策略基于LRU替换策略的基本原则，但在插入新块时将新块被插入到最近最少使用的位置，当新块被重复访问时，它们将逐渐向MRU位置移动，直到它们成为最近访问的块。这样的话，当一个新的块被加入进来之后，如果没有进行再次的访问，那么很有可能在下次就被替换掉了。这样在组相联的程度提升之后，可能同一个块的访问并没有那么频繁，也就是一个新块插入之后可能只访问了一次后面就访问别的地方了，在缓存发生替换时，未来可能还要访问的块就被替换掉了。

通过横向对比：在16路组相联的情况下，Random的替换策略有最好的性能；而在8路组相联的情况下，反而是NMRU策略有更好的性能；而在4路组相联的情况下，反而是LIP策略有更好的性能。由此可得组相联程度并非对一个策略越高越好，它需要搭配着不同的策略进行适配。

性能最好的配置为：16路组相联RandomRP替换策略。（实际上各个运行性能的差别并不是很大，仅有少量的区别）

三、实验问题分析

$$\text{tag\_latency (cycles)} = \frac{\text{lookup time (ps)}}{T_{\text{CPU}}} = \text{lookup time(ps)} * f_{\text{CPU}}(GHz)$$

	Random	**NMRU**	**LIP**
Max assoc	16	8	8
Lookup time	100ps	500ps	555ps
tag_latency	1	2	2

	NumCycles	**SimSeconds**	**dCacheMissRate:overall**	**l2MissRate:overall**
Random 16	3557564	0.001619	0.061123	0.125264
NMRU 8	3563472	0.001621	0.058668	0.134514
LIP 8	3563472	0.001621	0.105604	0.077969

在稳定状态下（即缓存的每一个块都合法载入），Random策略的算法时间复杂度为O(1)、而NMRU和LIP策略都不可避免的要对整个组进行遍历，所以策略的时间复杂度为O(n)，其中n为该组中块的个数。而我们通过计算可以得到Random的tag_latency最小，同时Random的时间复杂度最低，则Random替换策略更优。