实验3 缓存的设计与优化

1 NMRU 实现方法

仿照MRU策略的实现方式，根据`mru_rp.hh`和`mru_rp.cc`构造`nmru_rp.hh`和`nmru_rp.cc`。`nmru_rp.hh`中描述了`ReplData`的数据结构，定义了数据结构上操作方式的接口。与`nmru_rp.hh`相同，我们仍然维护`lastTouchTick`这一数据结构来记录cache block最近一次被访问的时钟周期，初始为0（代表该cache block是invalid的）。`nmru_rp.hh`如下所示：

```
#ifndef __MEM_CACHE_REPLACEMENT_POLICIES_NMRU_RP_HH__
#define __MEM_CACHE_REPLACEMENT_POLICIES_NMRU_RP_HH__

#include "mem/cache/replacement_policies/base.hh"

namespace gem5
{

struct NMRURPParams;

GEM5_DEPRECATED_NAMESPACE(ReplacementPolicy, replacement_policy);
namespace replacement_policy
{

class NMRU : public Base
{
  protected:
    struct NMRUReplData : ReplacementData
    {
        Tick lastTouchTick;
        NMRUReplData() : lastTouchTick(0) {}
    };

  public:
    typedef NMRURPParams Params;
    NMRU(const Params &p);
    ~NMRU() = default;

    void invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
                                                                override;
    void touch(const std::shared_ptr<ReplacementData>& replacement_data) const
                                                                override;
    void reset(const std::shared_ptr<ReplacementData>& replacement_data) const
                                                                override;
    ReplaceableEntry* getVictim(const ReplacementCandidates& candidates) const
                                                                override;
    std::shared_ptr<ReplacementData> instantiateEntry() override;
};

} // namespace replacement_policy
} // namespace gem5

#endif // __MEM_CACHE_REPLACEMENT_POLICIES_NMRU_RP_HH__
```

在`nmru_rp.cc`中对`nmru_rp.hh`中的虚函数进行了实现，其中对数据结构`lastTouchTick`进行维护的各接口仍然保持了`mru_rp.cc`中的实现方式，仅修改`NMRU::getVictim`函数的实现以达成不同的替换策略。我们注意到，当cache之间的传输带宽较大时，的确可能出现多个cache block被同时访问的情况。在该情况下，满足mru条件的cache block并不是唯一的。但是在我们的模拟中，L1 d-cache和L2 cache的`block_size`均被设置为`block_size=64`（参见`config.init`），为了避免多次生成随机数带来的时间消耗，我们采用了以下的方式。即，假设mru是唯一的，找到其标号后在生成一个与该标号不同的随机编号作为被替换块的编号。`nmru_rp.cc`如下所示：

```
#include "mem/cache/replacement_policies/nmru_rp.hh"

#include <cassert>
#include <memory>

#include "params/NMRURP.hh"
#include "base/random.hh"
#include "sim/cur_tick.hh"

namespace gem5
{

GEM5_DEPRECATED_NAMESPACE(ReplacementPolicy, replacement_policy);
namespace replacement_policy
{

NMRU::NMRU(const Params &p)
  : Base(p)
{
}

void
NMRU::invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
{
    std::static_pointer_cast<NMRUReplData>(
        replacement_data)->lastTouchTick = Tick(0);
}

void
NMRU::touch(const std::shared_ptr<ReplacementData>& replacement_data) const
{
    std::static_pointer_cast<NMRUReplData>(
        replacement_data)->lastTouchTick = curTick();
}

void
NMRU::reset(const std::shared_ptr<ReplacementData>& replacement_data) const
{
    std::static_pointer_cast<NMRUReplData>(
        replacement_data)->lastTouchTick = curTick();
}

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

```
    ReplaceableEntry* victim;
    bool find = false;
    int counter = 0;
    int mru_index = 0; // notice that there is only one mru
    ReplaceableEntry* mru_entry = candidates[0];

    for (const auto& candidate : candidates) {
        std::shared_ptr<NMRUReplData> candidate_replacement_data =
            std::static_pointer_cast<NMRUReplData>(candidate->replacementData);

        // Stop searching entry if a cache line that doesn't warm up is found.
        if (candidate_replacement_data->lastTouchTick == 0) {
            victim = candidate;
            find = true;
            break;
        }
        else if (candidate_replacement_data->lastTouchTick >
                std::static_pointer_cast<NMRUReplData>(
                    mru_entry->replacementData)->lastTouchTick) {
            mru_entry = candidate;
            mru_index = counter;
        }
        counter++;
    }

    // notice that hashed value won't wrap back to mru_index
    if(!find){
        victim = candidates[(mru_index+random_mt.random<unsigned>(1,
candidates.size() - 1))%candidates.size()];
    }

    return victim;
}

std::shared_ptr<ReplacementData>
NMRU::instantiateEntry()
{
    return std::shared_ptr<ReplacementData>(new NMRUReplData());
}

} // namespace replacement_policy
} // namespace gem5
```

最后在`ReplacementPolicies.py`中加入

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

以在python中使用该C++类。以及修改`SConscript`文件进行注册（注意：需要修改两处）：

```
Import('*')

SimObject('ReplacementPolicies.py', sim_objects=[
    'BaseReplacementPolicy', 'DuelingRP', 'FIFORP', 'SecondChanceRP',
```

```
    'LFURP', 'LRURP', 'BIPRP', 'MRURP', 'RandomRP', 'BRRIPRP', 'SHiPRP',
    'SHiPMemRP', 'SHiPPCRP', 'TreePLRURP', 'WeightedLRURP', 'NMRURP'])

Source('bip_rp.cc')
Source('brrip_rp.cc')
Source('dueling_rp.cc')
Source('fifo_rp.cc')
Source('lfu_rp.cc')
Source('lru_rp.cc')
Source('mru_rp.cc')
Source('random_rp.cc')
Source('second_chance_rp.cc')
Source('ship_rp.cc')
Source('tree_plru_rp.cc')
Source('weighted_lru_rp.cc')
Source('nmru_rp.cc')
```

2 修改配置方法

2.1 增加命令行可选参数

在`config/common`目录下，作以下操作。在`ObjectList.py`中加入

```
repl_list = ObjectList(getattr(m5.objects, 'BaseReplacementPolicy', None))
```

在`Options.py`中加入

```
    parser.add_argument("--l1d_repl", action='store', type=str, default="LRURP",
                        choices=ObjectList.repl_list.get_names(),
                        help = "replacement policy for l1")
    parser.add_argument("--l2_repl", action='store', type=str, default="LRURP",
                        choices=ObjectList.repl_list.get_names(),
                        help = "replacement policy for l2")
```

在`CacheConfig.py`中对`system.l2`和`dcache`的实例化过程进行修改，如下所示

```
system.l2 = l2_cache_class(clk_domain=system.cpu_clk_domain,
                           size=options.l2_size,
                           assoc=options.l2_assoc,

replacement_policy=ObjectList.repl_list.get(options.l2_repl)())
```

```
dcache = dcache_class(size=options.l1d_size,
                      assoc=options.l1d_assoc,

 replacement_policy=ObjectList.repl_list.get(options.l1d_repl)())
```

2.2 手动修改`tag_latency`

在`config/common`目录下的`Cache.py`中修改`L2Cache`的`tag_lantecy`（由于`L1Cache`的`tag_latency`较小，依照对照实验中的单一变量原则，不做修改），如下所示：

```
class L2Cache(Cache):
    assoc = 8
    tag_latency = 20
    data_latency = 20
    response_latency = 20 #手动修改为1、2
    mshrs = 20
    tgts_per_mshr = 12
    write_buffers = 8
```

3 模拟结果展示与分析

3.1 题目1

保持`Cache.py`中`L2Cache`的`tag_lantecy`为默认值（20），执行以下脚本，结果保存在`result_20`目录下。脚本如下：

```
GEM5=/home/mingkai/Course/calab-gem5/gem5-stable
SRC=/home/mingkai/Course/calab-gem5/benchmark/mm
RESULT_DIR=/home/mingkai/Course/calab-gem5/lab3/result
TARGET=configs/example/se.py

rm -rf ${RESULT_DIR}/*

for REPL in RandomRP NMRURP LIPRP; do
    mkdir -p ${RESULT_DIR}/${REPL}
done

cd $GEM5

for REPL in RandomRP NMRURP LIPRP; do
    for ASSOC in 4 8 16;do
        build/X86/gem5.opt ${TARGET} --cmd=${SRC} --cpu-type=DerivO3CPU \
            --l1d_size=64kB --l1i_size=64kB --l1d_repl=${REPL} --
l1d_assoc=${ASSOC} --caches \
            --l2_size=2MB --l2cache --l2_repl=${REPL} \
            --sys-clock=2GHz --cpu-clock=2GHz --mem-type=DDR3_1600_8x8
        cp -r m5out/ ${RESULT_DIR}/${REPL}/ASSOC_${ASSOC}/
    done
done
```

模拟结果如下所示（simTicks）：

	LIPRP	NMRURP	RandomRP
4	1986576000	1986576000	1986576000
8	1986576000	1986576000	1986576000
16	1986994500	1986994500	1986576000

可以看到，不同的配置的最后结果相差并不是很大。对于`mm`，在使用LIP或者NMRU替换策略且相联度为16时，在以上的配置中表现最好。对于相联度而言，更大的相联度可以减少cache的miss rate，但可能会增大每次查询所需要的时间。对于替换策略而言，LIP是LRU Insertion Policy的简称，相较于传统的LRU算法，LIP在初始时将所有块均设置为LRU块，只有当其被访问时才会从LRU侧调度到MRU侧，该算法对于workload的需求大大超过cache承载能力的情况下均有更好的效果。NMRU替换掉不是MRU的任意一个block，相较于LRU算法，也类似的避免了在workload较大情况下的抖动。故而对于`mm`而言，由于其workload较大，且访问顺序不是随机的（即具有一定的locality性质），使用适当更大的相联度，可以在牺牲部分查找时间的情况下更好地减少miss rate，使用LIP/NMRU也对workload较大的情况具有一定的促进作用。由此，提高了总体的性能。

3.2 题目2

修改`Cache.py`中`L2Cache`的`tag_lantecy`为1，执行以下脚本，结果保存在`result_1`目录下。脚本如下：

```
GEM5=/home/mingkai/Course/calab-gem5/gem5-stable
SRC=/home/mingkai/Course/calab-gem5/benchmark/mm
RESULT_DIR=/home/mingkai/Course/calab-gem5/lab3/result
TARGET=configs/example/se.py

rm -rf ${RESULT_DIR}/*

for REPL in RandomRP; do
    mkdir -p ${RESULT_DIR}/${REPL}
done

cd $GEM5

for REPL in RandomRP; do
    for ASSOC in 4 8 16;do
        build/X86/gem5.opt ${TARGET} --cmd=${SRC} --cpu-type=DerivO3CPU \
            --l1d_size=64kB --l1i_size=64kB --l1d_repl=${REPL} --
l1d_assoc=${ASSOC} --caches \
            --l2_size=2MB --l2cache --l2_repl=${REPL} \
            --sys-clock=2.2GHz --cpu-clock=2.2GHz --mem-type=DDR3_1600_8x8
        cp -r m5out/ ${RESULT_DIR}/${REPL}/ASSOC_${ASSOC}/
    done
done
```

修改`Cache.py`中`L2Cache`的`tag_lantecy`为2，执行以下脚本，结果保存在`result_2`目录下。脚本如下：

```
GEM5=/home/mingkai/Course/calab-gem5/gem5-stable
SRC=/home/mingkai/Course/calab-gem5/benchmark/mm
RESULT_DIR=/home/mingkai/Course/calab-gem5/lab3/result
TARGET=configs/example/se.py
```

```
rm -rf ${RESULT_DIR}/*

for REPL in NMRURP LIPRP; do
    mkdir -p ${RESULT_DIR}/${REPL}
done

cd $GEM5

for REPL in NMRURP LIPRP; do
    for ASSOC in 4 8;do
        build/X86/gem5.opt ${TARGET} --cmd=${SRC} --cpu-type=DerivO3CPU \
            --l1d_size=64kB --l1i_size=64kB --l1d_repl=${REPL} --
l1d_assoc=${ASSOC} --caches \
            --l2_size=2MB --l2cache --l2_repl=${REPL} \
            --sys-clock=2.2GHz --cpu-clock=2.2GHz --mem-type=DDR3_1600_8x8
        cp -r m5out/ ${RESULT_DIR}/${REPL}/ASSOC_${ASSOC}/
    done
done
```

模拟结果如下所示：

	RP	assoc.	tag_latency	simTicks
1	Random	4	1	1797260465
2	Random	8	1	1797258645
3	Random	16	1	1797263650
4	NMRU	4	2	1799112770
5	NMRU	8	2	1799601440
6	LIP	4	2	1799112770
7	LIP	8	2	1799112770

配置2更优，理由如下：

在该实验中，tag_latency带来的影响占据了主导地位，由于RandomRP使用了更小的tag_latancy，其总体性能更优。对于相联度而言，更大的相联度可以减少cache的miss rate，但可能会增大每次查询所需要的时间。从上面的结果中我们可以看到，assoc.=8是比较合理的更大相联度。