LAB3

NMRU的实现

思想：从最近没有使用的块中随机选择一个进行替换。

修改 `getVictim` 函数，将最近5次没有使用的块放入一个数组中，从该数组中随机选择一个作为替换块，如果数组为空，则从所有块中随机选择一个。

NMRU的getVictim()

C++ | 复制代码

```cpp
ReplaceableEntry*
NMRU::getVictim(const ReplacementCandidates& candidates) const
{
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);
    std::vector<ReplaceableEntry*> nmruCandidates;
    // record current tick
    Tick currentTick = curTick();
    for (const auto& candidate : candidates) {
        if ((currentTick - std::static_pointer_cast<NMRUReplData>(
            candidate->replacementData)->lastTouchTick) >= 5) {
            nmruCandidates.push_back(candidate);
        }
    }
    ReplaceableEntry* victim;
    if (nmruCandidates.size() > 0) {
        int randomIndex = random_mt.random<int>(0, nmruCandidates.size() - 1);
        victim = nmruCandidates[randomIndex];
    } else {
        int randomIndex = random_mt.random<int>(0, candidates.size() - 1);
        victim = candidates[randomIndex];
    }
    return victim;
}
```

创建SimObject

1. 实现 `nmru_rp.cc` `nmru_rp.hh`
2. 编辑 `/src/mem/cache/replacement_policies` 路径下的 `ReplacementPolicies.py`，添加：

Python | 复制代码

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

3. 将以下语句添加到 `/src/mem/cache/replacement_policies/SConscript`，

```python
dcache_class( ... , replacement_policy=NMRURP())
dcache_class.replacement_policy = NMRURP()
```

4. 将下面的内容添加到 `nmru_rp.hh`

```cpp
#ifndef __PARAMS__NMRURP__
#define __PARAMS__NMRURP__

namespace gem5 {
namespace replacement_policy {
class NMRU;
} // namespace replacement_policy
} // namespace gem5

#include "params/BaseReplacementPolicy.hh"

namespace gem5
{
struct NMRURPParams
    : public BaseReplacementPolicyParams
{
    gem5::replacement_policy::NMRU * create() const;
};

} // namespace gem5

#endif // __PARAMS__NMRURP__
```

5. 注册C++文件：将以下语句添加到 `/src/mem/cache/replacement_policies/SConscript`

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

6. 编译 `scons build/X86/gem5.opt`

配置修改

配置替换策略：

修改 `~/gem5/configs/common/CacheConfig.py` 文件，分别另存为 `CacheConfig_lip.py CacheConfig_lru.py CacheConfig_nmru.py CacheConfig_random.py`

修改方式如下：

Python | 复制代码

```python
#CacheConfig.py
for i in range(options.num_cpus):
        if options.caches:
            icache = icache_class(**_get_cache_opts('l1i', options))
            # FIXL:
            dcache = dcache_class(**_get_cache_opts('l1d', options))

#CacheConfig_lip.py
            dcache = dcache_class(**_get_cache_opts('l1d', options), replacement_policy=LIPRP())
#CacheConfig_lru.py
                        dcache = dcache_class(**_get_cache_opts('l1d', options), replacement_policy=LRURP())
#CacheConfig_nmru.py
            dcache = dcache_class(**_get_cache_opts('l1d', options), replacement_policy=NMRURP())
#CacheConfig_random.py
            dcache = dcache_class(**_get_cache_opts('l1d', options), replacement_policy=RandomRP())
```

修改 `~/gem5/configs/example/se.py` 分别另存为 `se_lab3_lip.py se_lab3_lru.py se_lab3_nmru.py se_lab3_random.py`

修改方式

```python
#se.py
from common import CacheConfig
#...
else:
    #...
    CacheConfig.config_cache(args, system)
    #...

#se_lab3_lip.py
from common import CacheConfig_lip
CacheConfig_lip.config_cache(args, system)
#se_lab3_lru.py
from common import CacheConfig_lru
CacheConfig_lru.config_cache(args, system)
#se_lab3_nmru.py
from common import CacheConfig_nmru
CacheConfig_nmru.config_cache(args, system)
#se_lab3_random.py
from common import CacheConfig_random
CacheConfig_random.config_cache(args, system)
```

cache相联度、替换策略对访存的影响

编写测试脚本，分别使用 `lip lru nmru random` 四种替换策略下，相联度分别为 `2 4 8 16` 的测试结果进行分析。

```bash
#!/bin/bash

GEM5_BASE=~/gem5
GEM5_OPT=~/gem5/build/X86/gem5.opt
SRC_DIR=~/lab3/src
RESULT_DIR=~/lab3/result
SE_PY=~/gem5/configs/example/se_lab3

M5OUT=m5out
OUT_TXT=${M5OUT}/stats.txt
OUT_INI=${M5OUT}/config.ini
rm -rf ${RESULT_DIR}/*
FILE=mm
for REL in nmru random lip lru; do
    for ASSOC in 2 4 6 8; do
        ${GEM5_OPT} ${SE_PY}_${REL}.py --cmd=${SRC_DIR}/${FILE} --cpu-type=DerivO3CPU \
            --l1d_size=64kB --l1d_assoc=${ASSOC} --l1i_size=64kB --caches
        mkdir -p ${RESULT_DIR}/${REL}/${ASSOC}
        cp  ${OUT_TXT} ${OUT_INI} ${RESULT_DIR}/${REL}/${ASSOC}
    done
done
```

结果分析。通过分析 `system.cpu.cpi` 和 `system.cpu.dcache.ReadReq.missRate::cpu.data` 来比较不同配置的性能。前者为CPI与性能成反比，后者为dcache缺失率，同样和配置性能成反比。

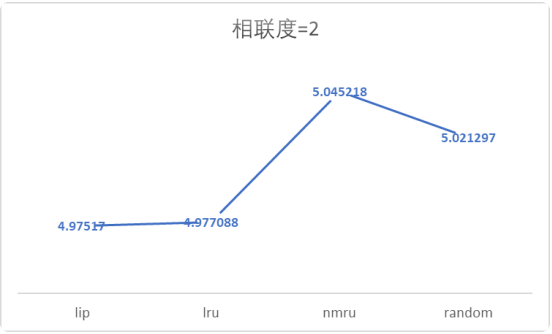
使用如下脚本文件获取测试结果：

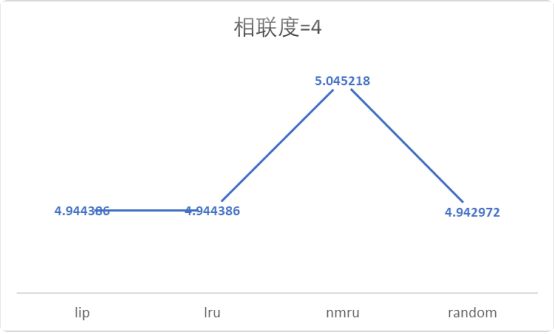
```bash
> stats.txt
for FILE in lip lru nmru random; do
    for DIR in 2 4 6 8;do
        cat ./result/${FILE}/${DIR}/stats.txt | grep system.cpu.cpi >> stats.txt
        cat ./result/${FILE}/${DIR}/stats.txt | grep system.cpu.dcache.ReadReq.missRate::cpu.data >> stats.txt
    done
done

cat stats.txt | awk '{print $2}'|xargs -n2;
```

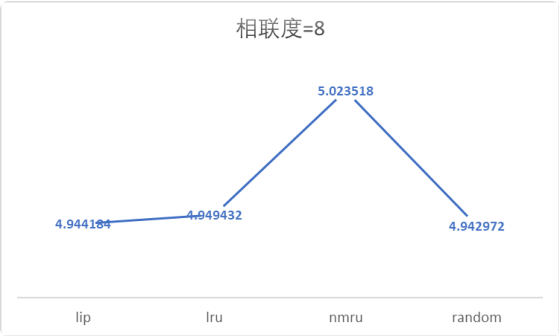
结果整理如下：

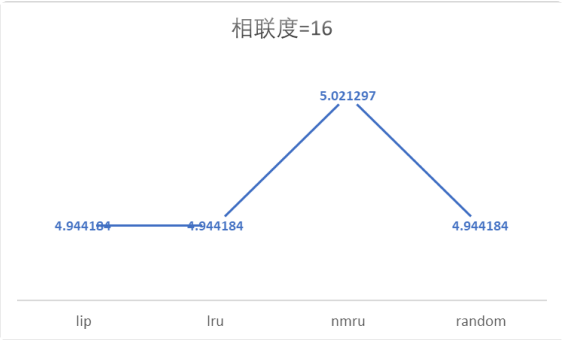
替换策略	相联度	CPI	dcache缺失率
lip	2	4.97517	0.151158
	4	4.944386	0.149037
	8	4.944184	0.149037
	16	4.944184	0.149037
lru	2	4.977088	0.151567
	4	4.944386	0.149037
	8	4.949432	0.149037
	16	4.944184	0.149037
nmru	2	5.045218	0.157137
	4	5.045218	0.156483
	8	5.023518	0.158151
	16	5.021297	0.157782
random	2	4.975473	0.152377
	4	4.942972	0.149037
	8	4.942972	0.149037
	16	4.944184	0.149037

在相同的相联度下，不同替换策略之间性能差异：



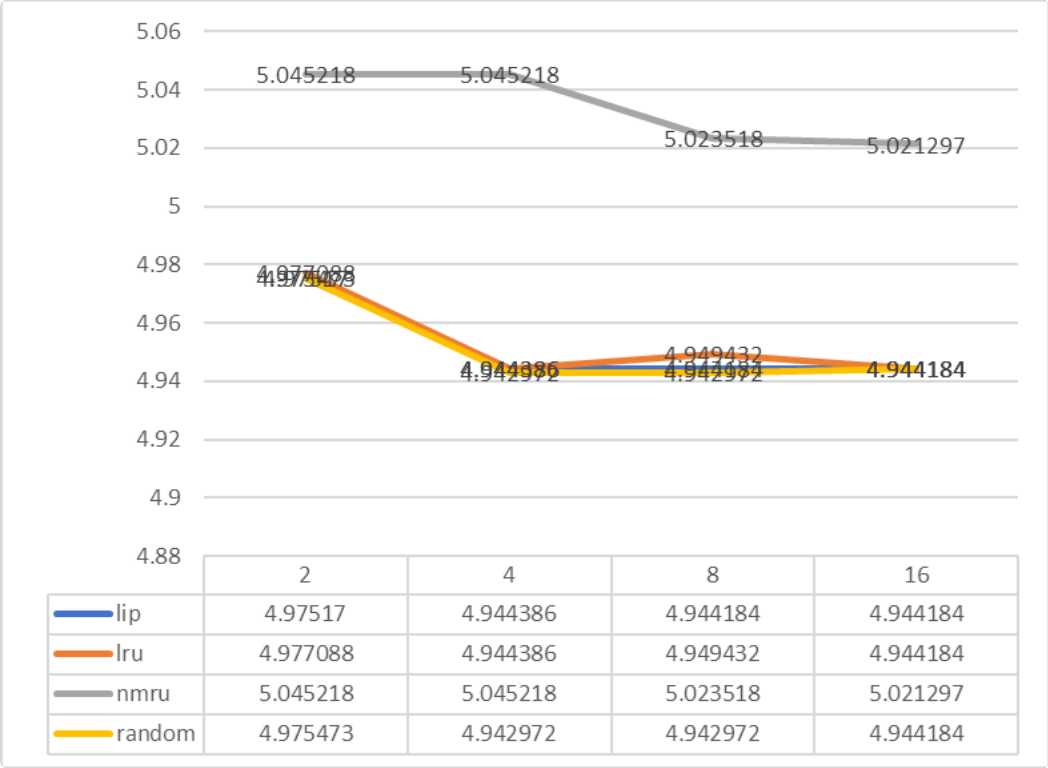






- 当相联度=2时，lip替换策略最佳，而当相联度>2时，random策略最佳。
- lru受相联度影响不大，性能总体都较好。
- nmru受相联度影响也不大，性能始终为最差

在相同的替换策略下，性能受相联度影响情况：



	2	4	8	16
lip	4.97517	4.944386	4.944184	4.944184
lru	4.977088	4.944386	4.949432	4.944184
nmru	5.045218	5.045218	5.023518	5.021297
random	4.975473	4.942972	4.942972	4.944184

- 对nmru，性能随相联度的增大而提升，有4变为8时提升最明显
- 对其它3种替换策略，相联度由2变为4时性能明显提升，由4变为8时略微降低。这三种策略在相联度大于4后，受相联度影响总体不大
- 总体来看，nmru的性能总是低于其余三种。

性能最佳的配置为：替换策略为random,相联度选择为4。出现该结果的原因是，选择的cache大小较小，其余三种由于计算量较大，性能反而不如几乎没有计算量的random策略。而对于random策略，在相联度高的情况下，更多的数据块存在相关性，在缓存中这些数据块的存在会对缓存命中率产生较大的影响。如果随机选择一个相关性较高的数据块被替换出去，缓存命中率可能会急剧下降，影响整个系统的性能。因此random策略在相联度不太大时的性能优于lru等策略。

一个实际情况

考虑一个实际情况，O3CPU 2.2GHz，issuewidth = 8。策略限制如下：

	Random	NMRU	LIP
Max assoc.	16	8	8
Lookup time	100ps	500ps	555ps

500ps和555ps转换为2.2GHz下的周期数均为2（向上取整）。而100ps转换为周期数为1.因此后两个测试不用修改 `Caches.py` 文件，因为 `tag_latency` 默认就是2

先用如下脚本获得后两个测试的结果：

run2.sh　Bash　复制代码

```
#!/bin/bash

GEM5_BASE=~/gem5
GEM5_OPT=~/gem5/build/X86/gem5.opt
SRC_DIR=~/lab3/src
RESULT_DIR=~/lab3/result2
SE_PY=~/gem5/configs/example/se_lab3

M5OUT=m5out
OUT_TXT=${M5OUT}/stats.txt
OUT_INI=${M5OUT}/config.ini
rm -rf ${RESULT_DIR}/*
FILE=mm

for REL in nmru lip; do
    for ASSOC in 8; do
        ${GEM5_OPT} ${SE_PY}_${REL}.py --cmd=${SRC_DIR}/${FILE} --cpu-type=DerivO3CPU \
            --l1d_size=16kB --l1d_assoc=${ASSOC} --l1i_size=64kB --caches \
            --sys-clock=2.2GHz --cpu-clock=2.2GHz
        mkdir -p ${RESULT_DIR}/${REL}/${ASSOC}
        cp  ${OUT_TXT} ${OUT_INI} ${RESULT_DIR}/${REL}/${ASSOC}
    done
done
```

结果如下：

	CPI	dcache缺失率
lip	4.844209	0.148506
nmru	4.925814	0.159865

而对第一个测试，需要将 `~/gem5/configs/conmmon/Caches.py` 做如下修改：

Python | 复制代码

```python
#修改前
class L1_DCache(L1Cache):
    pass
#修改后
class L1_DCache(L1Cache):
    assoc = 2
    tag_latency = 1
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20
```

重新编译 `scons build/X86/gem5.opt`

使用以下脚本文件活得测试结果：

Bash | 复制代码

```bash
REL=random
ASSOC=16
${GEM5_OPT} ${SE_PY}_${REL}.py --cmd=${SRC_DIR}/${FILE} --cpu-type=DerivO3CPU \
            --l1d_size=16kB --l1d_assoc=${ASSOC} --l1i_size=64kB --caches \
            --sys-clock=2.2GHz --cpu-clock=2.2GHz
        mkdir -p ${RESULT_DIR}/${REL}/${ASSOC}
        cp  ${OUT_TXT} ${OUT_INI} ${RESULT_DIR}/${REL}/${ASSOC}
```

从结果种筛选性能分析需求数据。

最终结果如下：

	CPI	dcache缺失率
lip	4.960484	0.160772

nmru	5.036437	0.177673
random	5.04002	0.174367

如上所示：lip替换策略最佳。

原因：LIP策略使缓存中的数据尽可能不被替换出去，在缓存大小有限的情况下提高缓存命中率。LIP替换策略通过保持缓存中相联度一定，来提高缓存命中率。总之LIP策略在cache较小时性能更优。如果cache大一点，这种优异差距将减少。

例如，下面是将l1d_size有16kB改为64kB的测试结果：

	CPI	dcache缺失率
lip	4.814686	0.149133
nmru	4.867878	0.156592
random	4.814686	0.149133

此时lip策略和random策略性能相同。