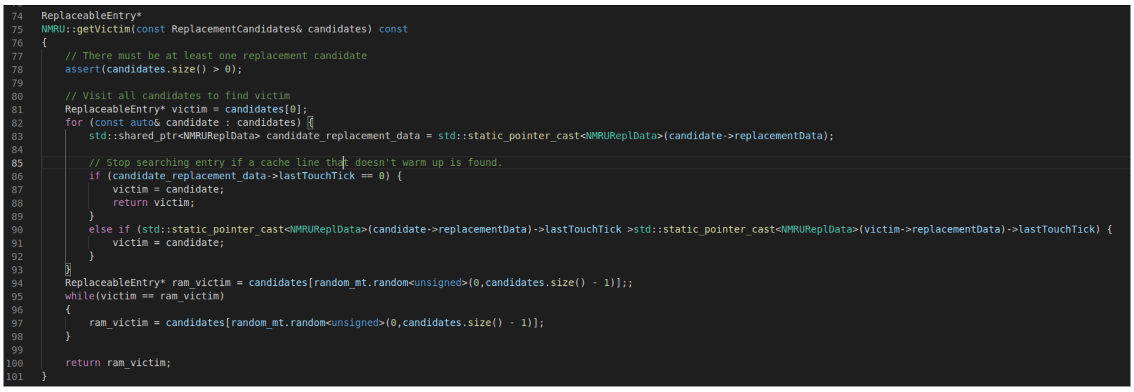
CALab3Report

PB19051183 吴承泽

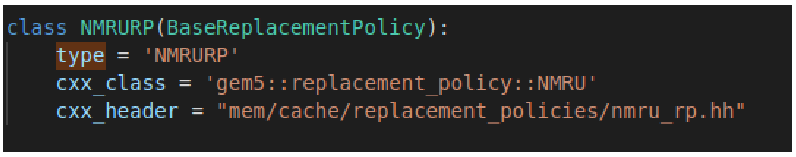
实现NMRU策略

1. 在`/src/mem/cache/replacement_policies/`中将复制一份`lru_rp.cc`与`lru_rp.hh`，命名为`nmru_rp.cc`与`nmru_rp.hh`，修改`nmru_rp`如下：



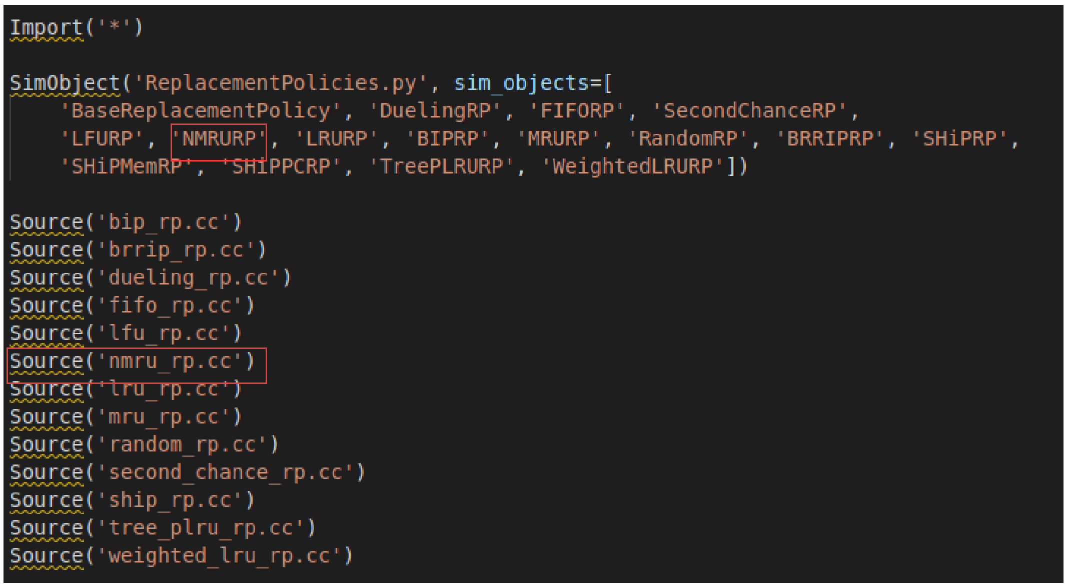
```cpp
ReplaceableEntry*
NMRU::getVictim(const ReplacementCandidates& candidates) const
{
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);

    // Visit all candidates to find victim
    ReplaceableEntry* victim = 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;
            return victim;
        }
        else if (std::static_pointer_cast<NMRUReplData>(candidate->replacementData)->lastTouchTick >std::static_pointer_cast<NMRUReplData>(victim->replacementData)->lastTouchTick) {
            victim = candidate;
        }
    }
    ReplaceableEntry* ram_victim = candidates[random_mt.random<unsigned>(0,candidates.size() - 1)];;
    while(victim == ram_victim)
    {
        ram_victim = candidates[random_mt.random<unsigned>(0,candidates.size() - 1)];
    }

    return ram_victim;
}
```

2. 修改`/src/mem/cache/replacement_policies/ReplacementPolicies.py`，添加如下代码：



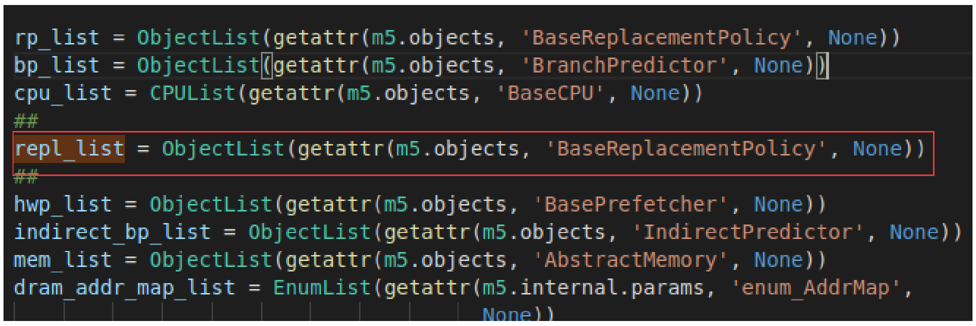
```python
class NMRURP(BaseReplacementPolicy):
    type = 'NMRURP'
    cxx_class = 'gem5::replacement_policy::NMRU'
    cxx_header = "mem/cache/replacement_policies/nmru_rp.hh"
```

3. 修改`/src/mem/cache/replacement_policies/SConscript`，添加如下代码：



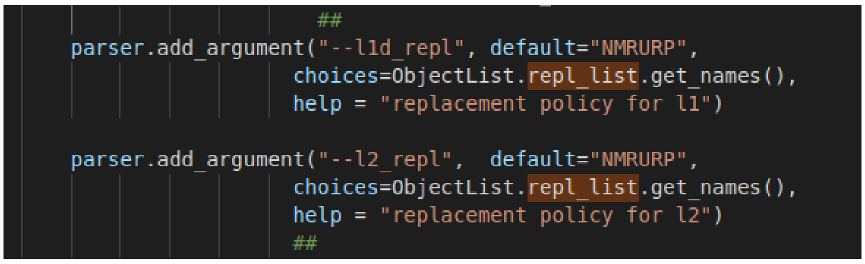
```python
Import('*')

SimObject('ReplacementPolicies.py', sim_objects=[
    'BaseReplacementPolicy', 'DuelingRP', 'FIFORP', 'SecondChanceRP',
    'LFURP', 'NMRURP', 'LRURP', 'BIPRP', 'MRURP', 'RandomRP', 'BRRIPRP', 'SHiPRP',
    'SHiPMemRP', 'SHiPPCRP', 'TreePLRURP', 'WeightedLRURP'])

Source('bip_rp.cc')
Source('brrip_rp.cc')
Source('dueling_rp.cc')
Source('fifo_rp.cc')
Source('lfu_rp.cc')
Source('nmru_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')
```

4. 在`configs/common/ObjectList.py`添加如下语句：



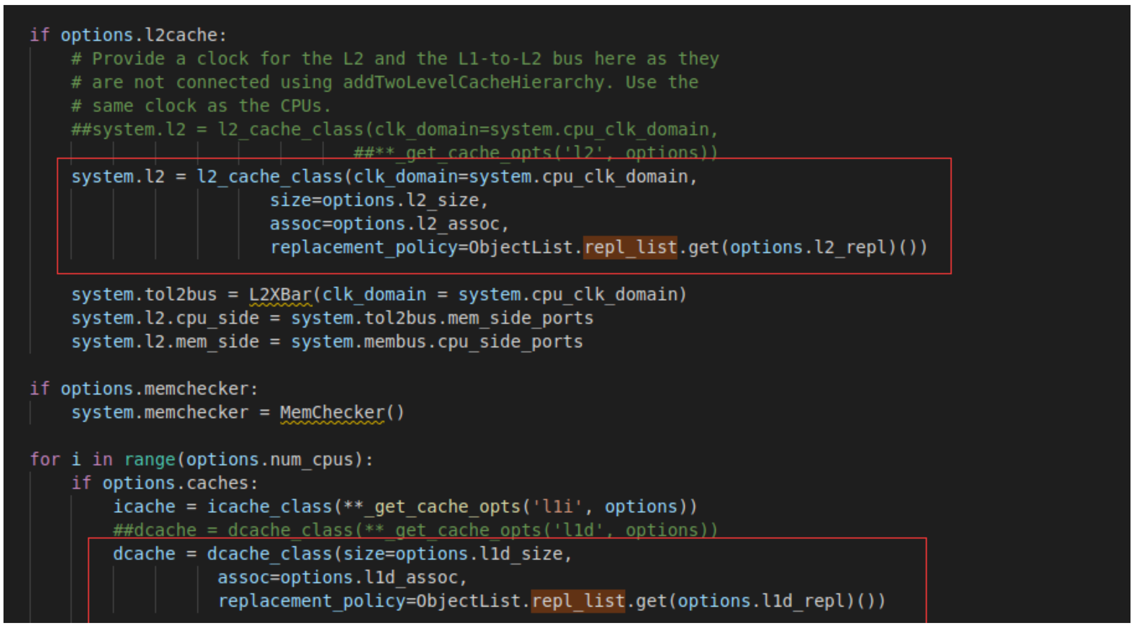
```python
rp_list = ObjectList(getattr(m5.objects, 'BaseReplacementPolicy', None))
bp_list = ObjectList(getattr(m5.objects, 'BranchPredictor', None))
cpu_list = CPUList(getattr(m5.objects, 'BaseCPU', None))
##
repl_list = ObjectList(getattr(m5.objects, 'BaseReplacementPolicy', None))
##
hwp_list = ObjectList(getattr(m5.objects, 'BasePrefetcher', None))
indirect_bp_list = ObjectList(getattr(m5.objects, 'IndirectPredictor', None))
mem_list = ObjectList(getattr(m5.objects, 'AbstractMemory', None))
dram_addr_map_list = EnumList(getattr(m5.internal.params, 'enum_AddrMap',
                                      None))
```

在`configs/common/Options.py`的`AddNoISAOption`类中添加：



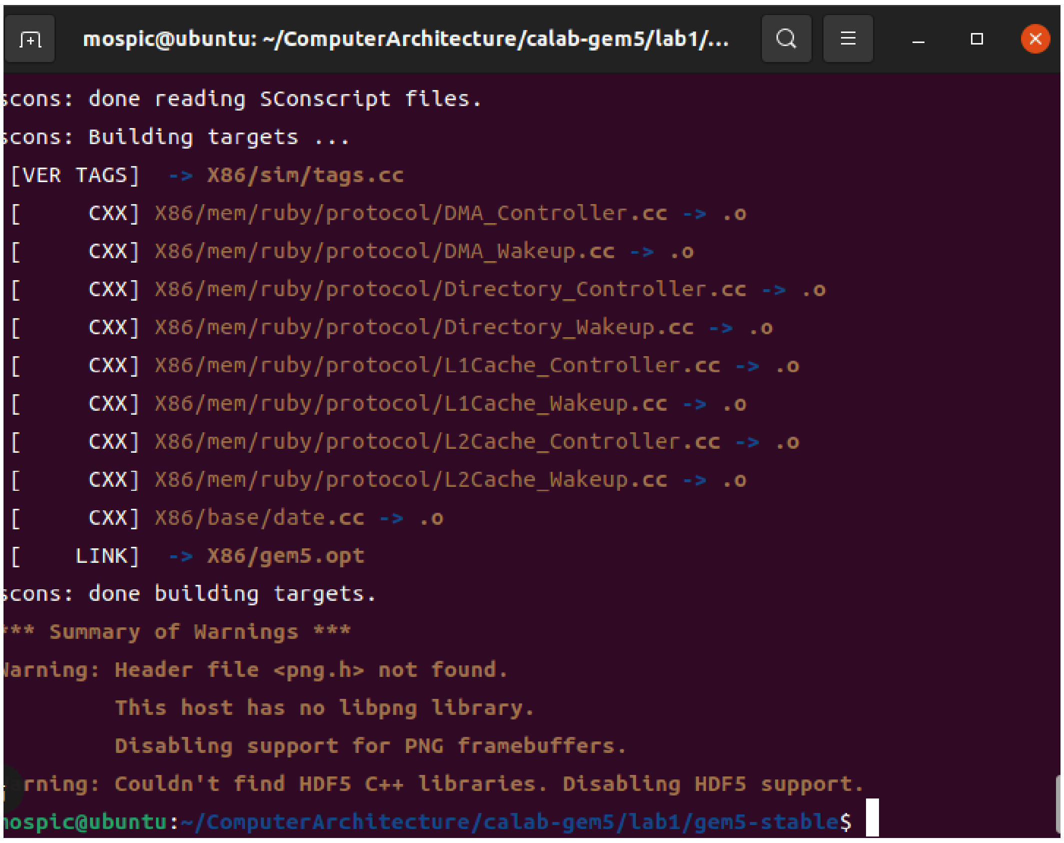
```
                    ##
    parser.add_argument("--l1d_repl", default="NMRURP",
                        choices=ObjectList.repl_list.get_names(),
                        help = "replacement policy for l1")

    parser.add_argument("--l2_repl",  default="NMRURP",
                        choices=ObjectList.repl_list.get_names(),
                        help = "replacement policy for l2")
                        ##
```

在 `configs/common/CacheConfig.py` 修改 `dcache` 与 `system.l2`：



```
if options.l2cache:
    # Provide a clock for the L2 and the L1-to-L2 bus here as they
    # are not connected using addTwoLevelCacheHierarchy. Use the
    # same clock as the CPUs.
    ##system.l2 = l2_cache_class(clk_domain=system.cpu_clk_domain,
                               ##**_get_cache_opts('l2', options))
    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)())

    system.tol2bus = L2XBar(clk_domain = system.cpu_clk_domain)
    system.l2.cpu_side = system.tol2bus.mem_side_ports
    system.l2.mem_side = system.membus.cpu_side_ports

if options.memchecker:
    system.memchecker = MemChecker()

for i in range(options.num_cpus):
    if options.caches:
        icache = icache_class(**_get_cache_opts('l1i', options))
        ##dcache = dcache_class(**_get_cache_opts('l1d', options))
        dcache = dcache_class(size=options.l1d_size,
                     assoc=options.l1d_assoc,
                     replacement_policy=ObjectList.repl_list.get(options.l1d_repl)())
```

编译gem5:

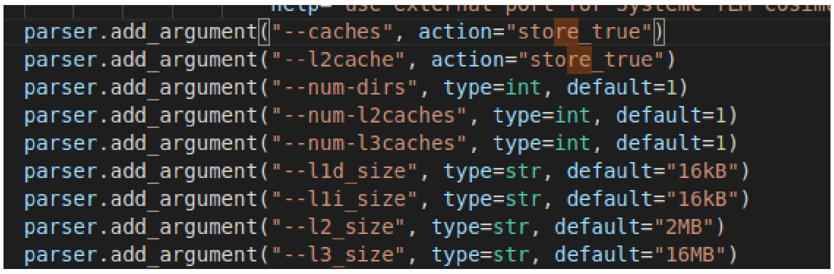


```
mospic@ubuntu: ~/ComputerArchitecture/calab-gem5/lab1/...
scons: done reading SConscript files.
scons: Building targets ...
[VER TAGS]  -> X86/sim/tags.cc
[     CXX] X86/mem/ruby/protocol/DMA_Controller.cc -> .o
[     CXX] X86/mem/ruby/protocol/DMA_Wakeup.cc -> .o
[     CXX] X86/mem/ruby/protocol/Directory_Controller.cc -> .o
[     CXX] X86/mem/ruby/protocol/Directory_Wakeup.cc -> .o
[     CXX] X86/mem/ruby/protocol/L1Cache_Controller.cc -> .o
[     CXX] X86/mem/ruby/protocol/L1Cache_Wakeup.cc -> .o
[     CXX] X86/mem/ruby/protocol/L2Cache_Controller.cc -> .o
[     CXX] X86/mem/ruby/protocol/L2Cache_Wakeup.cc -> .o
[     CXX] X86/base/date.cc -> .o
[    LINK]  -> X86/gem5.opt
scons: done building targets.
*** Summary of Warnings ***
Warning: Header file <png.h> not found.
         This host has no libpng library.
         Disabling support for PNG framebuffers.
rning: Couldn't find HDF5 C++ libraries. Disabling HDF5 support.
mospic@ubuntu:~/ComputerArchitecture/calab-gem5/lab1/gem5-stable$
```

设计基于乱序 O3CPU 处理器的 cache

1、

修改*l1cache*大小均为*16KB*：



```
parser.add_argument("--caches", action="store_true")
parser.add_argument("--l2cache", action="store_true")
parser.add_argument("--num-dirs", type=int, default=1)
parser.add_argument("--num-l2caches", type=int, default=1)
parser.add_argument("--num-l3caches", type=int, default=1)
parser.add_argument("--l1d_size", type=str, default="16kB")
parser.add_argument("--l1i_size", type=str, default="16kB")
parser.add_argument("--l2_size", type=str, default="2MB")
parser.add_argument("--l3_size", type=str, default="16MB")
```

运行命令如下（以模拟`mm`，使用`NMRURP`的替换策略，设置`assoc`为2为例）：

```
build/X86/gem5.opt configs/example/se.py --cmd=lab2-benchmark/cs251a-microbench-master/mm --cpu-type=DerivO3CPU --cpu-clock='1GHz' --IssueWidth=8 --caches --l1d_repl='LIPRP' --assoc=2
```

基于不同的相联度与不同的替换策略：可以得到一共四类替换策略**Random、NMRU、LRU、LIP与assoc=2\4\8\16**一共十六项实验结果，分析如下：

不同替换策略，相联度不同得到的实验结果如下所示（**dcache='16kB'**）：

simTicks:

strategy\assoc	**2**	**4**	**8**	**16**
LIP	64306000	62984000	62639000	62137000
LRU	63965000	63628000	62639000	62271000
NMRU	63965000	63528000	63010000	62686000
Random	64100000	63754000	63069000	62695000

dcache.overallMissRate::cpu.data:

strategy\assoc	**2**	**4**	**8**	**16**
LIP	0.102776	0.096504	0.094335	0.094871
LRU	0.102359	0.097390	0.092527	0.092402
NMRU	0.102359	0.100569	0.099700	0.097742
Random	0.107000	0.102649	0.097569	0.099087

dcache.replacements:

strategy\assoc	**2**	**4**	**8**	**16**
LIP	287	242	223	224
LRU	289	244	223	215
NMRU	289	265	261	248
Random	312	277	248	250

在**同一种替换策略下，不同的相联度中**，失效率大致满足：

$$MissRate_{assoc=2} > MissRate_{assoc=4} > MissRate_{assoc=16} \approx MissRate_{assoc=8}$$

替换块数大致满足：

$$Replacements_{assoc=2} > Replacements_{assoc=4} > Replacements_{assoc=16} \approx Replacements_{assoc=8}$$

由lab2的分析，程序`mm.c`的*memory locality*比较强，因此在`assoc`增加时，不同块放在不同组的概率变大，*MissRate*应减少。

而当`assoc`增加至16时，由于*memory regularity*较强且每组的缓存个数较少，部分数据倾向于放入同一组导致的*Replacements*易增加，同时导致*MissRate*中组数增加对*MissRate*的影响与频繁映射至同一组对*MissRate*的影响持平，*MissRate*依据不同的替换策略在`assoc=8`与`assoc=16`有着不同的规律。

*Replacements*与Cache查询次数和失效率的乘积正相关，而对于同一个程序，其他参数相同的情况下Cache的查询次数大致相同，因此对于失效率的分析在Replacements处同样适用。

在**不同的替换策略下，相同相联度**，失效率大致满足（`assoc=2/4`较小时）：

$$MissRate_{LIP} \approx MissRate_{LRU} < MissRate_{NMRU} < MissRate_{Random}$$

替换块数大致满足：

$$Replacements_{LIP} \approx Replacements_{LRU} < Replacements_{NMRU} < Replacements_{Random}$$

- **LIP**策略是结合了LRU和MRU算法，将部分行替换入MRU端，驻留时间在cache的时间比传统的LRU更长，对于特定循环序列工作集相较LRU有着较好的优化，对`mm.c`程序中大量的`for`循环有着较好的优化；
- **LRU**策略是最近最少使用算法，对于对于连续缓存的工作场景下容易表现一般，因为部分块的使用时间间隔略久，LRU替换策略可能会将即将要使用的块替换出去（即局部性较强），但在相联度较高时能有效避免一部分的块替换，当然LRU算法上的开销也会增加；
- **NMRU**策略是最近未使用块中随机选择一个，根据*memory locality*经验结论，曾经访问过的块在现在更易被访问，因此相较于LRU策略有更大的随机性，因此更容易产生*Cache Miss*，从而导致性能下降；
- **Random**策略完全随机的抽取Cache，有概率抽取到最近使用或最常使用的块，其*CacheMiss*率为最高的，性能也为最差的。

性能最佳的配置如下：`--l1d_repl='LIP' --assoc=16`，其*simTicks*在16组实验中最小。

性能提升的原因：理论上`assoc`越大，相联度越大，*MissRate*越小，因此访存内存的开销占比越小，而测试中使用的**LIP**策略在*Replacements*和*MissRate*表现均优异。

2、

考虑实际情况：**O3CPU 2.2GHz，issuewidth=8**:

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

模拟命令如下（使用O3CPU cpu_clock为2.2GHz 以LIP为例 assoc=8）：

```
build/X86/gem5.opt configs/example/se.py --cmd=lab2-benchmark/cs251a-microbench-master/mm --cpu-type=O3CPU --
cpu-clock='2.2GHz' --IssueWidth=8 --caches --l1d_repl='LIPRP' --assoc=8
```

最优的替换策略为：**lookup time=555ps; Max assoc=8 replacements_policy=LIP**

因为不同的策略下，性能开销的主要差异在执行程序对dcache的查找时间与对内存的访存时间，因此欲得到其中最优的替换策略，即比较上述二者的开销即可：

- 首先比较*NMRU*和*Random*，该配置的*Random*与该配置的*NMRU*的实验结果如下所示：
  $OverallMissLatency(Random) - OverallMissLatency(NMRU) = 58994655 - 57678610 \approx 1 * 10^6 ps$,

  而

  $(Lookuptime(NMRU) * dcache.overallaccess(NMRU) - Lookuptime(Random) * dcache.overallaccess(Random)) \approx 3.6 * 1$

  NMRU的时间开销之和大于Random时间开销之和，因此*Random*性能好于*NMRU*。
- 其次比较*LIP*与*Random*，该配置的*Random*与该配置的*LIP*的实验结果如下所示：

  $OverallMissLatency(Random) - OverallMissLatency(LIP) = 58994655 - 53931650 \approx 5 * 10^6 ps$,

  而

  $(Lookuptime(LIP) * dcache.overallaccess(LIP) - Lookuptime(Random) * dcache.overallaccess(Random)) \approx 4 * 10^6 ps$

  Random的时间开销之和大于LIP时间开销之和，因此*LIP*性能好于*Random*。

综上所述：**LIP**替换策略更优。