实验一 缓存的设计与优化

1. 实验目的

1. 了解 gem5 存储结构和替换策略的实现，学习创建 SimObject。
2. 练习缓存设计。

2 准备工作

2.0 概述

本次 Cache 实验将实现一个 SimObject，SimObject 对象是 gem5 中所有对象的基本类。

快速浏览 learning gem5 文档第二部分中的一个小节 gem5: Creating a very simple SimObject，理解如何创建 SimObject 对象并加入编译。

2.1 新建一个 Python 类

每个 SimObject 都有一个与之关联的 Python 类。

`type`：包装的 C++ 类，通常与类名相同。

`cxx_header`：包含用作 type 参数的类声明的文件。

`cxx_class`：指定新创建的 SimObject 在 gem5 命名空间中声明。

2.2 用 C++ 实现 SimObject

创建 `hello_object.hh` 和 `hello_object.cc`

按照惯例，gem5 将所有头文件与文件名及其所在目录一起包装在 `#ifndef/#endif` 中，因此没有循环包含。

所有 SimObjects 的构造函数都假定它将采用一个参数对象。这个参数对象由构建系统自动创建，并且基于 SimObject 的 `Python` 类.

2.3 注册 SimObject 和 C++ 文件

为了编译 C++ 文件和解析 Python 文件，我们需要将这些文件告知构建系统。

gem5 使用 SCons 作为构建系统，因此只需在包含 SimObject 代码的目录中创建一个 SConscript 文件。

这个文件只是一个普通的 Python 文件：

```
Import('*')

SimObject('HelloObject.py', sim_objects=['HelloObject'])
Source('hello_object.cc')
```

2.4 构建 gem5 并创建配置脚本

重新编译。

接下来是创建 Python 配置文件来实例化上面创建的对象。

导入 m5 和编译的所有对象，实例化 Root 对象，实例化 HelloObject 对象。

最后在 `m5` 模块上调用 `instantiate` 并实际运行仿真。

运行，结果为：

```
xxa@ubuntu:~/Desktop/gem5-stable$ sudo build/X86/gem5.opt configs/learning_gem5/part2/run_hello.py
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.2.1.0
gem5 compiled May  8 2023 05:23:50
gem5 started May  8 2023 05:28:24
gem5 executing on ubuntu, pid 38529
command line: build/X86/gem5.opt configs/learning_gem5/part2/run_hello.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
Hello World! From a SimObject!
Beginning simulation!
build/X86/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting simulation...
Exiting @ tick 18446744073709551615 because simulate() limit reached
```

3. 实现 NMRU 策略

3.0 复制源文件

仿照 LRU 策略，复制一份源文件，把其中任何 LRU 前缀重命名为 NMRU (区分大小写)

3.1 在 Python 中使用 C++ 类

SimObject 底层是 C++ 类。为了在 Python 中使用这些类， 需要创建 Python 对象，和 C++ 类联系起来。对于 NMRU 替换策略，我们首先继承 `BaseReplacementPolicy` 类，然后设置类的名称和 C++ 文件路径.

在 `/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"
```

3.2 修改源文件代码实现 NMRU

修改源文件代码实现 NMRU（MRU 的反面，**从最近没有使用的块中随机选择一个进行替换**），参考 random 策略的实现。

只需修改 `nmru_rp.cc` 文件下的 `getVictim` 函数，思路为：

1. 找到 lastTouchTick 最大的 candidate
2. 将其移除
3. 从剩下的 candidates 中随机选择一个作为 victim

```
ReplaceableEntry*
NMRU::getVictim(const ReplacementCandidates& candidates) const
{
```

```
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);

    // Find the candidate with the biggest lastTouchTick
    auto max_lastTouchTick = std::static_pointer_cast<NMRUReplData>
(candidates[0]->replacementData)->lastTouchTick;
    size_t max_index = 0;
    for (size_t i = 1; i < candidates.size(); ++i) {
        auto candidate_lastTouchTick = std::static_pointer_cast<NMRUReplData>
(candidates[i]->replacementData)->lastTouchTick;
        if (candidate_lastTouchTick > max_lastTouchTick) {
            max_lastTouchTick = candidate_lastTouchTick;
            max_index = i;
        }
    }

    // Remove the choosen candidate
    std::vector<ReplaceableEntry*> new_candidates(candidates.begin(),
candidates.end());
    new_candidates.erase(new_candidates.begin() + max_index);

    //Select a random entry from the remaining candidates
    ReplaceableEntry* victim = new_candidates[random_mt.random<unsigned>(0,
new_candidates.size() - 1)];

    return victim;
}
```

需要添加头文件：

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

#include <cassert>
#include <memory>
#include <vector>

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

3.3 注册 C++ 文件

将以下语句添加到 `/src/mem/cache/replacement_policies/SConscript`：

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

并且将 `'NMRURP'` 添加到 sim_objects 中。

3.4 编译并在 se.py 中配置替换策略

编译：

```
scons build/X86/gem5.opt -j9
CPU_MODELS=AtomicSimpleCPU,TimingSimpleCPU,O3CPU,MinorCPU
```

将 Cache 的替换策略写成一个参数。

向 `configs/common/ObjectList` 中添加：

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

向 `configs/common/Options.py` 加入：

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

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

修改 `configs/common/CacheConfig.py`：

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

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

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)())
```

运行：

```
build/X86/gem5.opt configs/example/se.py --help
```

可以看到如下内容：

```
  --l1d_repl {BIPRP,BRRIPRP,DRRIPRP,DuelingRP,FIFORP,LFURP,LIPRP,LRURP,MRURP,NMRURP,NRURP,RRIP
RP,RandomRP,SHiPMemRP,SHiPPCRP,SecondChanceRP,TreePLRURP,WeightedLRURP}
                        replacement policy for l1
  --l2_repl {BIPRP,BRRIPRP,DRRIPRP,DuelingRP,FIFORP,LFURP,LIPRP,LRURP,MRURP,NMRURP,NRURP,RRIPR
P,RandomRP,SHiPMemRP,SHiPPCRP,SecondChanceRP,TreePLRURP,WeightedLRURP}
                        replacement policy for l2
```

运行：

```
build/X86/gem5.opt \
        configs/example/se.py \
        --cpu-type=DerivO3CPU \
        --mem-type=DDR3_1600_8x8 \
        --caches \
        --l1d_size=64kB \
        --l1i_size=64kB \
        --cpu-clock=1GHz \
        --l1d_repl=NMRURP \
        --cmd=/home/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/lfsr
```

结果：

```
gem5 version 21.2.1.0
gem5 compiled May  8 2023 07:38:28
gem5 started May  8 2023 19:50:17
gem5 executing on ubuntu, pid 11498
command line: build/X86/gem5.opt configs/example/se.py --cpu-type=DerivO3CPU --mem-type=DDR3_1
600_8x8 --caches --l1d_size=64kB --l1i_size=64kB --cpu-clock=1GHz --l1d_repl=NMRURP --cmd=/hom
e/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/lfsr

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
build/X86/mem/mem_interface.cc:791: warn: DRAM device capacity (8192 Mbytes) does not match th
e address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
build/X86/sim/simulate.cc:194: info: Entering event queue @ 0.  Starting simulation...
build/X86/arch/x86/cpuid.cc:180: warn: x86 cpuid family 0x0000: unimplemented function 13
build/X86/sim/mem_state.cc:443: info: Increasing stack size by one page.
build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
Exiting @ tick 32409000 because exiting with last active thread context
```

4. 设计基于乱序 O3CPU 处理器的 cache

4.1 题目一

4.1.0 配置

使用实验二的测试程序 `mm.c`。基本参数假设：使用 O3CPU，L1D、L1I 大小为 64kB，L2 为 2MB。系统频率和 CPU 频率 2GHz，issuewidth = 8

测试配置如下：

配置号	替换策略	L1D Cache 相联度
0	Random	4
1	Random	8
2	Random	16
3	NMRU	4
4	NMRU	8
5	NMRU	16
6	LIP	4
7	LIP	8
8	LIP	16

注：L1D 和 L2 的替换策略一致

> 题目要求：完成完成上面要求的替换策略、相联度的模拟。描述模拟的所有配置和结果，讨论变量的影响。给出性能最佳的配置，分析性能提升原因。

以配置 0 为例，命令为：

```
outfile="mm"
cmd="/home/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/mm"

build/X86/gem5.opt \
    configs/example/se1.py \
    --cpu-type=O3CPU \
    --mem-type=DDR3_1600_8x8 \
    --caches \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --l2cache \
    --l2_size=2MB \
    --sys-clock=2GHz \
    --cpu-clock=2GHz \
    --l1d_repl=RandomRP \
    --l2_repl=RandomRP \
    --l1d_assoc=4 \
    --cmd=$cmd
cp -r m5out "/home/xxa/Desktop/tmpfile/${outfile}0"
```

完整测试脚本见附件。

4.1.1 运行结果

配置号	Ticks	dcache.overallAccesses::total	dcache.overallHits::total	dcache.HitRate	system.cpu.dcache.replacements	l2.HitRate
0	1748753000	2888123	2720273	0.9418827	27706	0.8710694
1	1748746000	2888123	2712129	0.9390628	28648	0.8750995
2	1748740000	2888124	2712740	0.9392741	28601	0.8749045
3	1748750500	2888123	2729509	0.9450806	25302	0.8594998
4	1748745500	2888124	2719497	0.9416137	27167	0.8686442
5	1748742000	2888134	2714904	0.9400201	28300	0.8736111
6	1748750000	2888124	2690831	0.9316882	33033	0.8909649
7	1748751000	2888124	2583381	0.8944841	47090	0.9225164
8	1748774000	2888124	2580460	0.8934727	47720	0.9235084

配置号	dcache.demandMissLatency::total	l2.demandMissLatency::total	SUM
0	2872658500	268745500	3141404000
1	2968032500	267940000	3235972500
2	2954121000	268061000	3222182000
3	2763635500	268273500	3031909000
4	2879382500	268435500	3147818000
5	2926561500	268529000	3195090500
6	3290052000	269423500	3559475500
7	4695600000	268791500	4964391500
8	4728280500	269450000	4997730500

4.1.2 分析

关于 LIP:

> The LRU Insertion Policy consists of a LRU replacement policy that instead of inserting blocks with the most recent last touch timestamp, it inserts them as the LRU entry. On subsequent touches to the block, its timestamp is updated to be the MRU, as in LRU. It can also be seen as a BIP where the likelihood of inserting a new block as the most recently used is 0%.

在常规的LRU策略中，新进入的缓存块（也就是最近访问的数据块）被认为是最新的（最近最常使用），因此在需要替换缓存块时，它们不太可能被替换。

然而，在LRU插入策略（LIP）中，新进入的缓存块被认为是最旧的（最近最少使用）。也就是说，当我们需要在缓存中为新数据腾出空间时，最新进入的缓存块将被首先考虑替换。不过，如果这个新进入的缓存块在之后被再次访问，那么它的状态就会更新为最新的（最近最常使用），就像在常规的LRU策略中一样。

替换策略 NMRU 和 LIP，大体上都呈现出随着相联度的提高，L1D Cache 的换入换出次数逐渐增大，Miss 的总延迟逐渐增大。替换策略 Random，呈现出随着相联度的提高， L1D Cache 的换入换出次数先增大后减小，Miss 的总延迟先增大后减小。

首先，在替换策略为 Random 或 NMRU 时，随着 dcache 相联度增加，运行时间略有下降；在替换策略为 LIP 时，随着 dcache 相联度的增加，运行时间略有上升。但是上述变化均非常小，变化比例小于 10^{-6} .

总的来说，相同组相联度下，dcache 命中率 NMRU > Random > LIP，Cache 块的换入换出次数：NRUM < Random < LIP

考虑运行时间，粗略来说呈现出：NMRU < Random < LIP，但是上述八种配置中，**运行时间最短的是替换策略选择为 Random，l1D cache 相联度选择为 16。**因为随着相联度增加到 16，Random 策略的时间降低幅度最大。

分析上述表现的原因，我认为有以下三条：

- NRUM 是除了刚换入的块以外随机换出一块，相比 Random 替换策略，保留了最新的块，在相联度比较低时这种策略效果更优，体现出测试的二进制文件是数据局部性好，尤其是时间局部性。所以 NMRU 的 Cache 块换入换出次数最少。但是相联度升高后，注意到，随着相联度的提高，NMRU 策略 dcache 的命中率逐渐降低，而 Random 策略是先降低后升高，所以相联度为 16 时，Random 策略得益于 dcache 的命中率升高，所以运行时间下降幅度更大。
- LIP 是 LRU Insertion Policy，新进入的块被视为最旧的，所以当我们需要在缓存中为新数据腾出空间时，最新进入的缓存块将被首先考虑替换。而测试文件具有比较好的时间局部性，所以这种策略表现并不好，从 LIP 策略 dcache 命中率为三种策略最低、Cache 块换入换出次数最多就可以看出来。
- 此外，对于提升组相联度的影响，由于 L1D Cache 足够大，对于 Cache 命中率和 Cache 块的换入换出次数来说，组相联度越高，这两个指标都变小，对于总运行时间来说应该是负面影响，但实际是随着相联度的升高，运行时间越来越小。我认为提升相联度对于运行时间的正面影响有：
    - **减少冲突Miss**：当相联度增加时，每个索引可以指向的缓存行数增加，这使得更多的数据块可以在同一个索引下被存储。因此，提高相联度可以减少由于两个或多个数据块需要映射到同一个缓存行而发生的冲突 miss。
    - **改善数据局部性**：提高相联度可以更好地利用程序的局部性。具有高相联度的缓存可以存储更多的数据块，这有助于利用程序的空间局部性。此外，提高相联度也有助于利用程序的时间局部性，因为数据块在被替换出缓存之前可以在缓存中停留更长的时间。

    但还是比较反常

不过上述测试结果中，9 种配置的指标差别不大，究其原因，我认为是 L1D Cache 比较大，所以我将 L1D Cache 减小到 1kB，重新测试，结果如下：

配置号	Ticks	dcache.overallAccesses::total	dcache.overallHits::total	dcache.HitRate	l2.HitRate
0	2110660500	2888003	585915	0.2028789	0.9917828
1	1817130000	2888216	1407911	0.4874674	0.9855115
2	1782572500	2888188	1803091	0.6242983	0.9832019
3	2104889500	2888007	522589	0.1809514	0.9918803
4	1812950000	2888264	1417140	0.4906546	0.9850982

配置号	Ticks	dcache.overallAccesses::total	dcache.overallHits::total	dcache.HitRate	l2.HitRate
5	1778300000	2888221	1826895	0.6325329	0.9826803
6	2152705000	2888044	527045	0.1824911	0.9922501
7	1997513000	2888144	859358	0.2975468	0.9914313
8	1972502500	2888244	1060348	0.3671255	0.9912543

首先可以看出，在 Cache 容量比较小时，提升相联度有助于提高命中率，进而提升效率。

从运行时间来看，仍然呈现出 NMRU > Random > LIP，尤其在组相联度比较高时，LIP 的运行时间比另外两种配置要长很多，因为其 L1D Cache 的命中率比较低。

但是还有一点与之前不同的是，此时是替换策略选择为 NMRU、相联度为 16 时运行时间最短。这还是体现出测试的二进制文件具有较好的空间局部性，刚访问的 Cache 块会被再次访问，所以 NMRU 保留最近访问过的块是有助于提升 Cache 命中率。

但是注意到，即使是之前 L1D Cache 大小为 64kB 时，NRUM 的 dcache 的命中率也是整体优于 Random 的，二者的 `dcache.overallAccesses::total` 不同，即缓存的访问次数不同，NMRU 的要略多于 Random，但是替换策略不会直接影响缓存的访问次数，与模拟器的内部实现细节有关。

所以对于**性能最佳的配置**这个问题，如果仅考虑本题给出的 9 中配置，那么运行时间最短的是**替换策略选择为 Random，l1D cache 相联度选择为 16。**

4.2 题目二

4.2.0 配置

考虑一个实际情况，O3CPU 2.2GHz，issuewidth = 8。三种策略限制如下：

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

修改 `Max assoc`：`--l1d_assoc=16`，`--l1d_assoc=8`

对于 `Lookup time`，在 gem5 模拟器中，`tag_latency` 参数是以 CPU 周期为单位，所以需要做转换。CPU 频率是 2.2GHz，那么一个 CPU 时钟周期的时间就是 1/2.2G = 0.454545 ns = 454.545 ps。如果 `lookup time` 是 100 ps，那么对应的 `tag_latency` 就是 100/454.545 = 0.22，由于 `tag_latency` 必须是整数，那么向上取整为 1。

所以实际配置如下：

	Random	NMRU	LIP
Max assoc	16	8	8
tag_latency	1	2	2

对于修改 `tag_latency`，直接修改 `/configs/common` 下的 `Caches.py` 文件内容。

Random

修改 `tag_latency = 1`

```
build/X86/gem5.opt \
    configs/example/se1.py \
    --cpu-type=O3CPU \
    --mem-type=DDR3_1600_8x8 \
    --caches \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --l2cache \
    --l2_size=2MB \
    --cpu-clock=2.2GHz \
    --l1d_repl=RandomRP \
    --l2_repl=RandomRP \
    --l1d_assoc=16 \
    --cmd="/home/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/mm"
```

NMRU

修改 `tag_latency = 2`

```
build/X86/gem5.opt \
    configs/example/se1.py \
    --cpu-type=O3CPU \
    --mem-type=DDR3_1600_8x8 \
    --caches \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --l2cache \
    --l2_size=2MB \
    --cpu-clock=2.2GHz \
    --l1d_repl=NMRURP \
    --l2_repl=NMRURP \
    --l1d_assoc=8 \
    --cmd="/home/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/mm"
```

LIP

修改 `tag_latency = 2`

```
build/X86/gem5.opt \
    configs/example/se1.py \
    --cpu-type=O3CPU \
    --mem-type=DDR3_1600_8x8 \
    --caches \
    --l1d_size=64kB \
    --l1i_size=64kB \
    --l2cache \
    --l2_size=2MB \
    --cpu-clock=2.2GHz \
    --l1d_repl=LIPRP \
    --l2_repl=LIPRP \
    --l1d_assoc=8 \
    --cmd="/home/xxa/Desktop/Arch_Labs/Lab2/cs251a-microbench-master/mm"
```

4.2.1 运行结果

配置	Ticks	dcache.overallAccesses::total	dcache.overallHits::total	dcache.HitRate
Random	1617154175	2888146	2707496	0.9374512
NMRU	1619468305	2888126	2717262	0.9408391
LIP	1619474220	2888126	2583121	0.8943935

4.2.2 分析

三种策略的模拟运行时钟周期数：Random < NMRU < LIP

首先，基于之前的分析，LIP 是 LRU Insertion Policy，新进入的块被视为最旧的，所以当我们需要在缓存中为新数据腾出空间时，最新进入的缓存块将被首先考虑替换。而测试文件具有比较好的时间局部性，所以这种策略表现并不好，从 LIP 策略 dcache 命中率为三种策略最低就可以看出来。

对于 Random 策略的时间小于 NMRU，我认为是由于 Random 本身的开销较小，虽然 NMRU 的命中率策略高，但还是 Random 的整体效率更优一些。

5. 附录

- `se1.py`：配置文件，设置了 Issue Width = 8
- `nmru_rp.hh`、`nmru_rp.cc`：实现 NMRU 的 C++ 代码
- `CacheConfig.py`、`Caches.py`、`ObjectList.py`、`Options.py`、`SConscript`、`ReplacementPolicies.py`：配置 Cache 策略时修改的配置文件（具体修改内容均在实验报告中指出）
- `runbenchmarkvm.sh`：第一题的运行脚本文件
- `output1`：第一题的模拟输出文件，即 9 种配置对应的输出
- `output2`：第二题的输出文件，即 3 中配置对应的输出