山东大学 · 泰山学堂

计算机体系结构课程实验报告

深入分析 gem5 模拟器

学生姓名 冯宗浩

学　　号 201300130013

指导教师 戴鸿君

2016 年 6 月 19 日

目录

1 实验目的 **3**

2 实验环境 **3**

3 实验内容 **3**

3.1 概述 3

3.2 环境搭建 3

3.3 SE 模式分析与系统调用实现 4

3.4 CPU 模型分析 6

3.4.1 AtomicSimpleCPU 模型和 TimingSimpleCPU 模型 6

3.4.2 InOrderCPU 模型 6

3.4.3 O3CPU 模型 7

3.4.4 对照实验 7

3.5 经典内存系统分析 9

3.6 Ruby 内存系统缓存一致性协议分析 10

3.6.1 MESI_CMP_directory 10

3.6.2 MOESI_CMP_directory 10

3.6.3 MOESI_CMP_token 10

3.6.4 MOESI_hammer 11

3.6.5 对照实验 11

3.7 新型缓存替换算法实现 11

4 结论分析与体会 **12**

1. 实验目的

1. 分析 gem5 SE 模式的内部原理，并尝试实现一些未完成的系统调用；
2. 分析 gem5 的 AtomicSimpleCPU 和 TimingSimpleCPU 模型；
3. 分析 gem5 的 InOrderCPU 模型；
4. 分析 gem5 的 O3CPU 模型；
5. 分析 gem5 的经典内存系统；
6. 分析 gem5 ruby 内存系统的 MESI_CMP_directory 缓存一致性协议；
7. 分析 gem5 ruby 内存系统的 MOESI_CMP_directory 缓存一致性协议；
8. 分析 gem5 ruby 内存系统的 MOESI_CMP_token 缓存一致性协议；
9. 分析 gem5 ruby 内存系统的 MOESI_hammer 缓存一致性协议；
10. 实现一个新的缓存替换算法，比如 DIP 和 DRRIP。

2. 实验环境

- Ubuntu 16.04 LTS
- Gem5 2.0

3. 实验内容

3.1. 概述

Gem5 是一款模块化的计算机系统模拟平台，支持模拟多种系统级别架构以及处理器微架构，在计算机体系结构研究等领域有广泛应用。Gem5 主要由 C++ 语言编写，辅以 python 语言编写的模拟配置脚本，既拥有较强的性能，又保证了良好的易用性。同时，Gem5 还是一款按照 BSD 协议开源的软件，这意味着我们不仅可以通过阅读源码来更好地理解其工作原理，还可以对其进行修改来添加需要的功能。

3.2. 环境搭建

我们推荐在 linux 平台下运行 gem5。下面以 ubuntu 系统为例，介绍 gem5 的安装过程：[1]

```
1 # 安装依赖组件
2 sudo apt-get update
3 sudo apt-get upgrade -y
4 sudo apt-get install -y m4 gcc g++ scons swig python python-dev mercurial zlib1g-dev
  libgoogle-perftools-dev
5
6 # 获取源代码
7 hg clone http://repo.gem5.org/gem5
8 cd gem5/
9
```

```
# 编译
scons build/X86/gem5.opt -j2
# 测试运行
build/X86/gem5.opt configs/example/se.py -c tests/test-progs/hello/bin/x86/linux/hello
```

如果想使用 gem5 模拟不同的的指令集架构 (ISA)，必须分别进行编译，此处我们编译的是 x86 架构。如果在编译过程中遇到问题，一般可通过升级编译器版本或扩大内存解决。

编译成功后，即可使用 gem5 模拟相应指令集架构的程序。模拟 gem5 自带的 x86 版 hello world 测试程序的结果如下：

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Jun 14 2016 13:12:15
gem5 started Jun 16 2016 15:30:35
gem5 executing on ubuntu, pid 3403
command line: build/X86/gem5.opt configs/example/se.py -c tests/test-progs/hello/bin/x86/linux/hello

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 5942000 because target called exit()
```

模拟结束后，gem5 目录下的 m5out 文件夹中会记录此次模拟的统计数据，其中 config.ini 与 config.json 内容相同，为此次模拟的完整参数列表，如使用的 CPU 模型、频率以及内存类型等；stats.txt 则记录了此次模拟的详细统计数据，如模拟时间、内存占用等 [4]；而安装 pydot 后，gem5 还能对此次模拟的配置进行可视化，如图. 1所示。

3.3. SE 模式分析与系统调用实现

Gem5 支持 ARM, Alpha, MIPS, SPARC, Power, x86 等多种指令集架构，它可以在 Full System (FS) 模式下模拟一个拥有硬件和软件的完整的系统，也可以在 Syscall Emulation (SE) 模式下通过直接提供系统服务来模拟用户程序。在 SE 模式中，gem5 能够模拟大多数常用的系统调用。当程序执行一个系统调用时，gem5 捕获并模拟这个调用，常用的处理方式是将其传给宿主操作系统。由于目前在 SE 模式中没有线程调度器，因此线程必须静态映射到处理器的核上，这限制了它在多线程程序上的应用。[2]

分析 gem5 源码后可以发现，gem5 支持的系统调用数量较为有限，一方面是因为部分系统调用的使用频率较低；另一方面是因为部分需要使用系统服务或 I/O 设备的系统调用只能在 FS 模式下运行，例如与网络相关的系统调用。通过浏览源码的 `src/arch/x86/linux/process.cc`

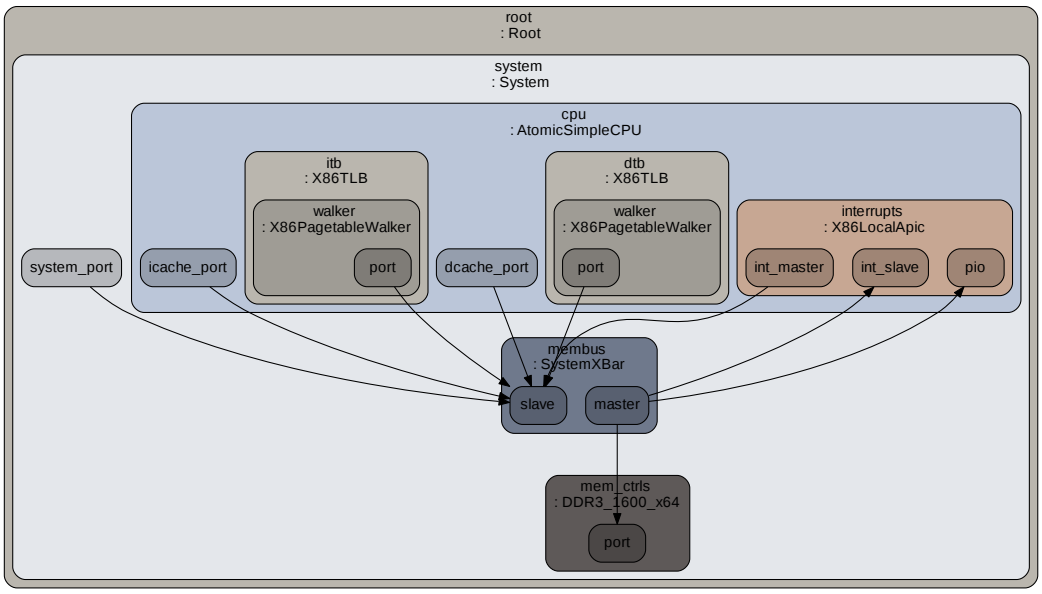

图. 1. 模拟配置可视化

文件，可以找到 x86 架构所有的系统调用及其对应的处理函数的列表 syscallDescs，其中已实现的系统调用对应的处理函数以 “系统调用名 + Func” 命名，如系统调用 read 对应的处理函数名为 readFunc；而未实现的系统调用对应的函数统一被设为 unimplementedFunc，该函数在执行时会提示系统调用未实现的错误。

在本实验中，我们选择在 x86 架构下实现系统调用 rmdir。该系统调用位于 unistd.h 头文件中，其定义为 `int rmdir(const char *pathname)`，功能是删除路径为 pathname 的空目录。若操作成功，则返回 0；否则返回 -1，并将 errno 设为相应的错误信息。参照目前已有的系统调用处理代码，我们编写了 rmdir 的处理函数 rmdirFunc，并将其置于 `src/sim/syscall_emul.hh` 文件中，再将 process.cc 中 rmdir 对应的函数进行修改，重新编译 gem5 后，即可得到支持 rmdir 系统调用的 gem5。

rmdirFunc 的代码如下所示：

```
SyscallReturn rmdirFunc(SyscallDesc *desc, int num, LiveProcess *p, ThreadContext *tc)
{
    string path;
    int index = 0;
    if (!tc->getMemProxy().tryReadString(path, p->getSyscallArg(tc, index)))
        return -EFAULT;

    // 获取路径
    path = p->fullPath(path);

    int result = rmdir(path.c_str());
    return (result == -1) ? -errno : result;
}
```

为了测试修改后的 gem5 是否真正实现了对 rmdir 系统调用的支持，我们编写了一个简单的测试程序，并将其分别在修改前后的 gem5 中进行模拟。测试程序简单直接地调用了 rmdir 系统调用，其代码如下：

```
#include <unistd.h>
#include <stdio.h>
int main() {
    if (rmdir("test") == 0) {
        puts("rmdir success");
    } else {
        puts("rmdir fail");
    }
    return 0;
}
```

需要注意的是 gem5 模拟的程序必须由静态编译生成，即程序不能依赖任何外部动态库，只需在编译时添加 `-static` 参数即可，编译完成后可使用 `ldd <FILE>` 命令来验证是否存在依赖。通过实际测试，发现在未修改的 gem5 下模拟该程序会出现 `fatal: syscall rmdir (#84) unimplemented` 异常，模拟随即终止。而在修改后的 gem5 中，程序能够正常运行，这证明我们成功地为 gem5 添加了对 rmdir 系统调用的支持。

3.4. CPU 模型分析

Gem5 支持 AtomicSimpleCPU, TimingSimpleCPU, InOrderCPU 和 O3CPU 等多种 CPU 模型，下面对它们的特点分别进行分析，并通过模拟同一个程序来进行横向对比。

3.4.1. AtomicSimpleCPU 模型和 TimingSimpleCPU 模型

AtomicSimpleCPU 和 TimingSimpleCPU 模型都继承自 BaseSimpleCPU 模型，它们都属于不支持流水线的有序指令模型，每个周期仅执行一条指令。它们的主要区别在于访存方式不同，AtomicSimpleCPU 模型不计算访存代价，即所有的访存操作都是立即完成，没有任何阻塞；而 TimingSimpleCPU 模型每次只允许处理一个访存请求，使用了存储器访问时序模型，在访问缓存时会在队列中停顿并等待内存系统返回处理的优先级。

3.4.2. InOrderCPU 模型

InOrderCPU 模型是一个顺序执行指令的流水线模型，注重于指令的计时以及模拟精度。该模型的特点在于把 CPU 可能访问的组件，如 ALU、分支预测器等抽象为可供每个指令请求的资源。使用该模型可以方便地修改定制流水线而不必重新设计完整的 CPU 模型，大大降低了难度和时间成本，可方便地用于模拟不同的流水线周期、发射宽度以及硬件线程数目。

3.4.3. O3CPU 模型

O3 (Out-of-Order) CPU 模型是一个乱序执行的流水线模型，模拟了指令、功能单元、内存访问以及流水线周期之间的依赖关系。参数化的流水线资源，如加载/存储队列和重排序缓存等，让 O3CPU 模型能够模拟不同尺度的架构。同时在 O3CPU 模型中，指令只会于解除所有依赖后在执行周期时执行。

3.4.4. 对照实验

为了比较这几种 CPU 模型的性能差异，我们编写了一个测试程序，并分别使用不同模型模拟该程序。我们选择的测试程序为 Eratosthenes 素数筛程序，该程序的特点是包含了大量的乱序访存操作。我们将程序的功能设定为计算 1,000,000 以内的素数个数，这样每次模拟的用时适中，能够恰当地反映出不同 CPU 模型的特性。

Eratosthenes 素数筛的代码如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MX 1000001

_Bool prime[MX];

void sieve() {
    for (int i = 2; i < MX; ++i) {
        prime[i] = 1;
    }

    for (int i = 2; i <= sqrt(MX); ++i) {
        if (prime[i]) {
            for (int j = i + i; j < MX; j += i) {
                prime[j] = 0;
            }
        }
    }
}

int main() {
    sieve();

    int cnt = 0;
    for (int i = 2; i < MX; ++i) {
        cnt += prime[i];
    }
    printf("%d\n", cnt);

    return 0;
}
```

模拟时可通过 `--cpu-type` 参数来指定 CPU 模型。由于 InOrderCPU 模型和 O3CPU 模型必须配合缓存工作，因此所有的模拟都在开启二级缓存的情况下进行，只需在模拟时添加参数 `--caches --l2cache` 即可。实验结果如图. 2和图. 3所示。

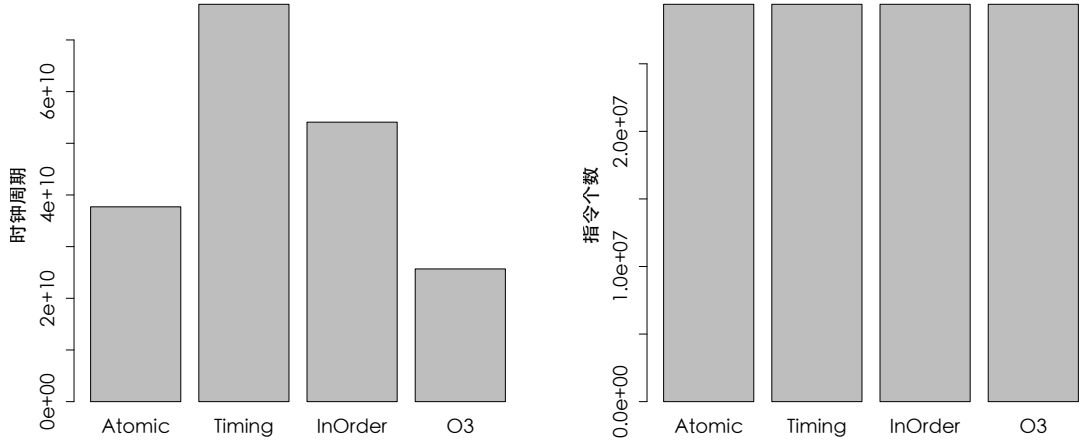

图. 2. 不同 CPU 模型的周期个数与指令个数

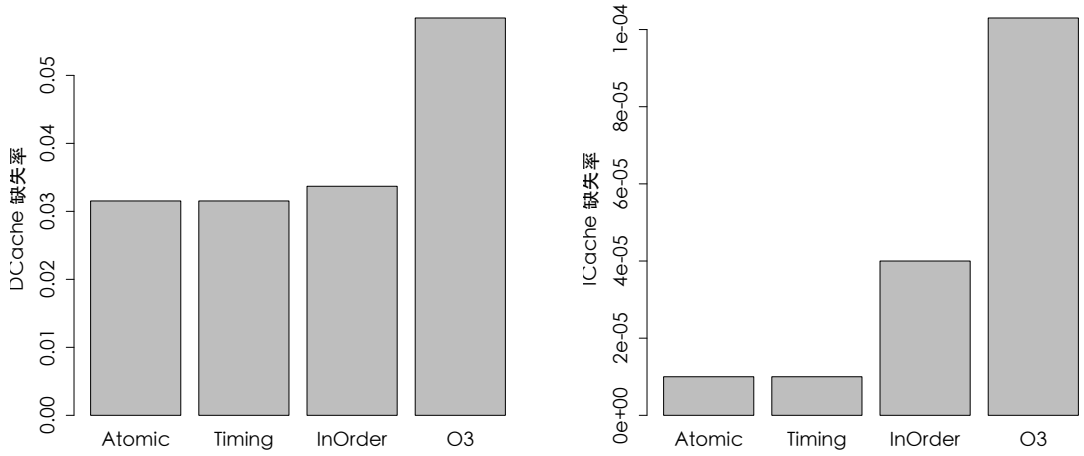

图. 3. 不同 CPU 模型的缓存缺失率

根据实验结果，可以发现各 CPU 模型执行的总指令个数大致相同，但经过的时钟周期数差别较大。这是由于不同 CPU 模型下同一种指令可能需要不同数量的时钟周期来完成，而这当中最主要的原因还是访存方式的不同：访存立即完成的 AtomicSimpleCPU 模型相比 TimingSimpleCPU 模型经过时钟周期数明显更少；而 InOrderCPU 模型由于引入了流水线，需要的时钟周期数有所减少；乱序执行流水线的 O3CPU 模型的性能则得到了进一步提升，甚至快于 AtomicSimpleCPU 模型，这也从侧面反应了流水线优化的威力。在缓存缺失率的表现上，非流水线模型优于流水线模型，这是由于前者的访存具有较强的规律性。同理，顺序流水线模型的缓存缺失率也优于乱序流水线模型。

3.5. 经典内存系统分析

Gem5 中提供了两种内存系统：经典内存系统来自 M5 模拟器，特点是快速并且易于配置；Ruby 内存系统来自 GEMS 模拟器，具有较强的灵活性，能够模拟多种缓存一致性系统。两种系统相辅相成，各自的优点和缺点也不尽相同。

经典内存系统的优点有：

- **快进**：经典内存系统支持 atomic 访存模式，该访存模式可以用于满足在执行过程中快进到程序的某一特定部分的需求；
- **速度**：经典内存系统不仅支持 atomic 访存模式，还支持 timing 访存模式，它们都快于 ruby 内存系统；
- **易于配置**：只需要简单修改 python 配置文件，就能创建任意的存储层次，同时经典内存系统的抽象缓存一致性协议也能自动扩展到任意的存储层次。

同时，经典内存系统也存在一些缺点：

- **缓存一致性的灵活性**：虽然经典内存系统的缓存一致性协议可以应用于不同的存储层次，但是经典内存系统被限定为使用 MOESI 协议，修改协议本身较为困难；
- **缓存一致性的精确性**：由于没有对临时状态进行建模，经典内存系统不能像 ruby 内存系统一样精确地模拟协议中的竞争等状况。

为了进行对比，我们分别使用经典内存系统和 ruby 内存系统模拟 3.4.4 中的素筛程序。经典内存系统是 gem5 的默认内存系统，若要转而使用 ruby 内存系统，只需在模拟时添加 `--ruby` 参数即可。实验结果如图. 4所示，可以看到两种内存系统的模拟用时相当，但经典内存系统模拟的时钟周期更多，这证明了其速度更快。

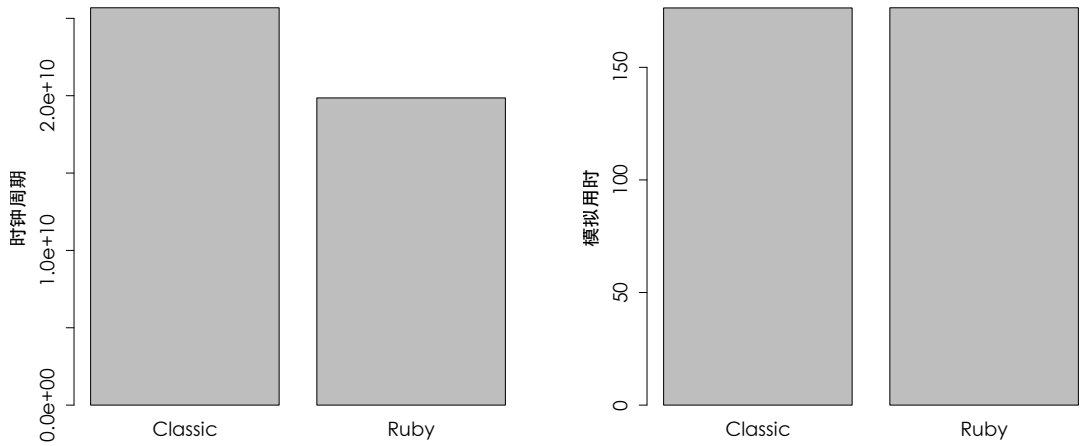

图. 4. 经典内存系统与 ruby 内存系统对比

3.6. Ruby 内存系统缓存一致性协议分析

3.6.1. MESI_CMP_directory

MESI_CMP_directory 缓存一致性协议又称 MESI_Two_Level 协议，是一个单芯片、两级缓存、严格包含（L1 缓存中的内容一定包含在 L2 缓存中）的协议。在本协议中，每个核拥有私有的 L1 缓存，而 L2 缓存在核之间共享。L1 缓存又被划分为指令缓存 (ICache) 和数据缓存 (DCache)。

该协议包含如下四种状态：

- **M**(Modified): 缓存块仅被一个 L1 缓存独占，没有其它的共享者。这块缓存既可以被读取，也可以被写入。这块缓存的数据是整个系统中唯一有效的拷贝，因为是脏的（已经被修改过)。
- **E**(Exclusive): 缓存块仅被一个 L1 缓存独占。与 M 状态大致相同，唯一的区别在于这块缓存还没有被写入。
- **S**(Shared): 缓存块被多个 L1 缓存或者 L2 缓存所共享，但是是只读的。
- **I**(Invalid): 缓存块已经无效，必须从存储器中读取。

该协议中的一个主要优化是如果一个 L1 缓存请求读取一块数据，而其它的 L1 缓存都没有这块数据，那么 L2 缓存控制器就把这块数据设为该 L1 缓存所独占（E 状态)。这么做是考虑到读取操作后有较大的概率进行写入操作，那么通过这个优化就节约了一次请求。这也正是 E 状态存在的意义。

3.6.2. MOESI_CMP_directory

MOSEI_CMP_directory 缓存一致性协议是一个多芯片、两级缓存、非包含（既不严格包含，也不独占）的协议。与 MESI 协议相比，MOESI 协议的控制器中还引入了额外的 O (Owned) 状态，用于表示相关块由该缓存拥有，在存储器中已经过时。在 MESI 协议中，如果尝试共享处于 M 状态的缓存块，则该块会被强制写回存储器中，而且状态会变为 S 状态；而 MOESI 协议的处理方式是将原缓存中这个块的状态改为 O 状态，其它新共享的块设为 S 状态，这样该块不必写到存储器中，只需要在发生缺失时让 O 状态的缓存提供该块即可。同时 MOESI 协议还引入了许多 MESI 协议中不存在的合并优化。

3.6.3. MOESI_CMP_token

MOSEI_CMP_token 缓存一致性协议是一个两级缓存协议。它通过显式交换与计数 token 来维护一致性权限。开始时每个缓存块分配了固定数量的 token。如果要写一个缓存块，处理器必须拥有这个块所有的 token；如果要读一个缓存块，则只需要拥有至少一个 token。该协议还拥有持续的消息支持来避免陷入饥饿。

3.6.4. MOESI_hammer

MOSEI_hammer 缓存一致性协议是一个单芯片、两级私有缓存、严格包含的协议。该协议是 AMD hammer 协议的实现，后者被用于 AMD Hammer 芯片（也就是 AMD 速龙处理器)。

3.6.5. 对照实验

由于 gem5 默认采用的缓存一致性协议为 MI_example，为测试不同的缓存一致性协议，必须在编译 gem5 时进行指定。例如若想使用 MOESI_CMP_directory 协议，则对应的编译指令应为 `scons PROTOCOL=MOESI_CMP_directory build/X86/gem5.opt`。在测试时我们采用了 gem5 的 ruby random test，该测试是专门设计为用于验证缓存一致性协议正确性的，只需在使用时选取 `configs/example/ruby_random_test.py` 配置文件即可。在本实验中我们将 CPU 核数设置为 4，测试负荷设置为 10000，相应的参数为 `-n 4 --maxloads=10000`。实验结果如图. 5 所示。

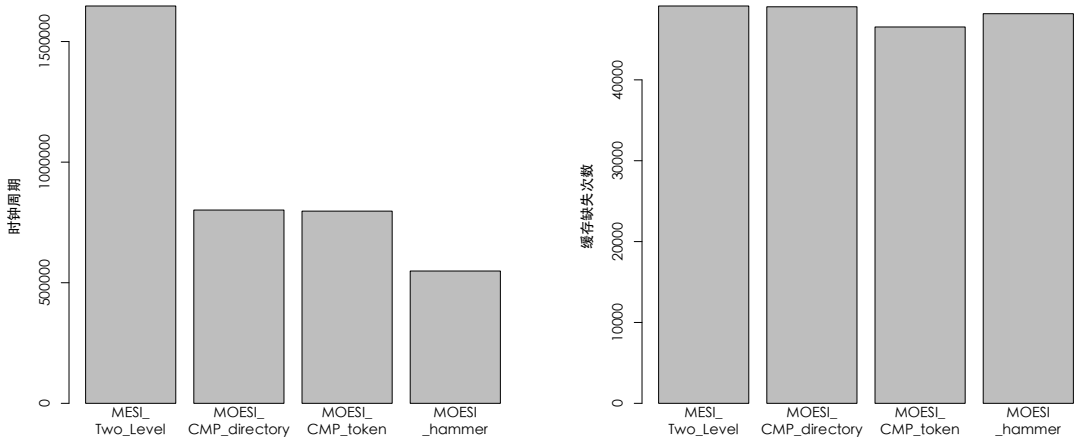

图. 5. 不同缓存一致性协议的对比

3.7. 新型缓存替换算法实现

在 gem5 中可以为不同的缓存对象单独指定缓存替换算法，这意味着 L1 和 L2 缓存，甚至指令缓存与数据缓存都可以分别使用不同的缓存替换算法。默认情况下 gem5 使用的缓存替换算法皆为 LRU (Least Recent Used)，即最久未使用算法。该算法的大致思路是在面临缓存缺失时优先替换最久时间没有使用过的缓存块。

NRU (Not Recent Used) 是一种 LRU 的近似算法，该算法为每块缓存设置一个标记位，用来表示该缓存块是否可能在最近被访问到。每当缓存命中时，该缓存块的标记位设为 0，表示最近可能会再次访问该缓存块；而当缓存缺失时，则顺序扫描所有缓存块，将第一个发现的标记为 1 的缓存块进行替换，若所有块的标记都为 0，则将所有块的标记置 1，再进行替换。

SRRIP (Static Re-Reference Interval Prediction) 将 NRU 算法进行了进一步扩展，把标记位的个数扩大至 m，以更加细致地区分缓存块的状态。与 NRU 相似，在缓存命中时该块标记为 0；而在缓存缺失时 SRRIP 寻找的是标记为 $2^m - 1$ 的缓存块，若找到则进行替换，否则将所有块的标记增加 1，继续进行寻找。对于新换入的缓存块，其标记设置为 $2^m - 2$，以防止那些很久才能被再次使用到的缓存块长期占用缓存空间。

如果程序的工作集大于缓存容量，则使用 SRRIP 可能引发频繁的换进换出，产生抖动问题。DRRIP (Dynamic Re-Reference Interval Prediction) 为了解决该问题对 SRRIP 进行了修改，当新换入缓存块时将其标记以较大概率设为 $2^m - 1$，或以较小概率设为 $2^m - 2$，从而避免了抖动。[3]

在本实验中我们选择实现 DRRIP 算法。在 gem5 中，缓存替换算法的实现位于 `src/mem/cache/tags` 文件夹中，可以发现 gem5 目前实现了 LRU、FALRU 和随机替换等多种缓存替换算法。参考已有算法的实现，我们可以自行编写 DRRIP 的实现代码，只需要重新实现 accessBlock 和 findVictim 等函数即可。accessBlock 函数的功能是访问一块给定地址的缓存，并更新替换信息；findVictim 函数的功能是在缓存缺失时选择一块被替换的缓存。实现完成后修改该文件夹中的 Tags.py 和 SConscript 把新文件添加到编译系统，之后重新编译 gem5 即可使用新的缓存替换算法。由于时间所限，目前我们 DRRIP 算法的实现尚不能正常工作，有待进一步完善。

4. 结论分析与体会

通过本次实验，我基本掌握了 gem5 模拟器的使用方法，并对其内部原理有了初步了解。同时在实验过程中，我对课程中学到的知识也有了更深刻的理解，收获很大。

引用

1. *Documentation - gem5.* http://gem5.org/Documentation.

2. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

3. Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

4. Jason Lowe-Power. *gem5 Tutorial.* http://www.lowepower.com/jason/learning_gem5/index.html.