



山东大学

SHANDONG UNIVERSITY

计算机系统原理第三次实验报告

小组成员及分工

姓名	班级	学号	分工
鲍泓睿	密码1班	202200460076	构建 gem5 获得内存地址序列, cache模拟器C++部分代码编写, 实验原理, cache模拟器正确性检验, 实验结果分析及相应部分的报告撰写
高钰超	密码1班	202200460136	cache模拟器python部分代码编写, 实验原理、实验数据可视化和实验结果分析及相应部分的报告撰写
陈万里	网安2班	202200460153	cache_visual工具封装与策略改进, cache可视化处理, 实验数据统计及相应部分报告撰写
郑傲宇	网安2班	202222460130	构建 gem5 与 ntt 可执行文件, 处理地址序列输出, cache模拟器正确性检验, 分析数据总结实验结论及相应部分的报告撰写

目录

目录

1 实验要求

2 实验环境

3 实验原理

3.1 gem5

3.1.1 gem5 是什么

3.1.2 本次实验用到的 gem5 特性

3.2 cache的基本概念及原理

3.3 cache的映射策略

3.3.1 直接映射 (Direct Mapping)

3.3.2 组相联 (Set Associative Mapping)

3.3.3 全相联 (Fully Associative Mapping)

3.4 cache的替换策略

3.4.1 最近最少使用 (LRU) 策略

3.4.2 先进先出 (FIFO) 策略

3.4.3 随机替换 (Random) 策略

3.4.4 最不常用 (LFU) 策略

4 实验内容

4.1 获得正确的输出

4.1.1 构建 gem5 时遇到的问题

4.1.2 编译 ARM 架构下的 ntt 可执行文件

4.1.3 处理输出

4.2 编写cache模拟器

4.2.1 处理地址序列文件

4.2.2 通过类class实现cache

4.2.2.1 Cache line

4.2.2.2 Cache

4.2.3 映射方式

4.2.3.1 直接映射

4.2.3.2 组相联

4.2.3.3 全相联

4.2.4 替换策略

4.2.4.1 最近最少使用 (LRU) 策略

4.2.4.2 先进先出 (FIFO) 策略

4.2.4.3 最不常用 (LFU) 策略

4.2.4.4 随机替换 (Random) 策略

5 数据处理与分析

5.1 模拟器封装

5.2 实验数据

6 实验结论

6.1 什么是仿真器，仿真器有什么作用/价值？

6.2 正确性分析

6.2.1 C++实现Cache模拟器正确性分析

6.2.2 可视化模拟器正确性分析 (Python)

6.3 Cache 大小对命中率的影响

6.3.1 原理分析

6.3.2 结果分析

6.4 不同组相连配置对命中率的影响

6.4.1 原理分析

6.3.2 结果分析

6.5 对 LRU 算法的分析

6.6 不同的代码实现与优化方法对 cache 命中率的影响

6.7 实验心得

7 附录

7.1 内存序列地址

7.2 Cache模拟器C++实现

7.3 Cache模拟器Python实现

7.4 完整实验数据

1 实验要求

- 实验目的

学习仿真器的基本功能，理解cache的工作原理，以及相应参数与算法对程序性能的影响。

- 实验内容

1. 通过gem5获得**快速数论变换NTT**程序的内存地址访问序列（trace）。
2. 编写cache模拟器，就是用软件的方法实现cache的工作原理，比如命中检查机制、内存地址和cache块的映射（全相联、组相联、直接映射）的实现和一些替换算法（Random,LRU）等。
3. 自行编写Cache模拟器，以获得的内存地址访问序列文件为输入，统计miss/hit情况。调整相关参数（cache块大小，cache大小）和算法（映射方式，替换算法），查看miss/hit的变化情况，并做出相应解释。在报告里尝试通过实验回答以下问题或者其他你认为有价值的问题：
 1. 什么是仿真器，仿真器有什么作用/价值；
 2. 如何验证你编写的cache仿真器的正确性（micro-benchmarking）；
 3. Cache的大小，组相连等配置对命中率的影响；
 4. LRU是否是一个好的cache替换算法；
 5. 不同的代码实现与优化方法对cache命中率有怎样的影响；

2 实验环境

处理器	Intel Core i7-12700H 6P8E
内存	DDR5 4800MHz Dual Channel 64GB
操作系统 内核	Linux 6.8.9-arch1-1 x86_64
编译器	gcc version 8.2.1 20180802 (GNU Toolchain for the A-profile Architecture 8.2-2018-08 (arm-rel-8.23))
编译选项	arm-linux-gnueabihf-gcc --static -o ntt ./ntt.c

3 实验原理

3.1 gem5

以下内容主要参考自 gem5 官方文档 (<https://www.gem5.org/>)。

3.1.1 gem5 是什么

gem5 是一个现代的计算机体系结构模拟研究平台，能够模拟多种指令集架构 (ISA) 和微架构，实现了系统模拟和仅应用模拟。本次实验使用了 gem5 的仅应用模拟功能，探查可执行文件运行时的内存读写序列。

3.1.2 本次实验用到的 gem5 特性

gem5 实现了一个事件驱动的内存系统，在这一系统的基础上，gem5 会为每一个 CPU 读写请求分配一个唯一的序号，这一特性称为 Memory Access Ordering。gem5 同时具有基于 trace 的 debug 功能，本次实验即通过这一 debug 功能和其内存特性获取 ntt 可执行文件的内存地址访问序列。

(参考: https://www.gem5.org/documentation/general_docs/memory_system/gem5_memory_system/, https://www.gem5.org/documentation/general_docs/debugging_and_testing/debugging/trace_based_debugging/)

3.2 cache的基本概念及原理

- 基本概念

缓存 (Cache) 是一种用于临时存储数据的高速存储设备或存储空间，其目的是减少对于较慢存储设备 (如主存储器或硬盘驱动器) 的访问次数，从而提高系统的性能和响应速度。在计算机系统中，缓存通常是位于处理器 (CPU) 和主存储器之间的一层存储结构，以及位于主存储器和外部存储 (如硬盘) 之间的另一层存储结构。

- 基本原理

1. 局部性原理：缓存的设计基于计算机程序的局部性原理，其中包括时间局部性和空间局部性。时间局部性指一个数据项一旦被访问，可能在不久的将来再次被访问；而空间局部性指一个数据项被访问时，相邻的数据项也可能被访问。因此，将最近使用或相关的数据项存储在缓存中，可以有效利用这种局部性原理提高访问效率。
2. 缓存命中和缓存未命中：当处理器请求数据时，如果所需数据已经在缓存中，则称为缓存命中 (Cache Hit)，处理器可以直接从缓存中读取数据；如果所需数据不在缓存中，则称为缓存未命中 (Cache Miss)，此时需要从更慢的存储层 (如主存储器或磁盘) 中加载数据到缓存中，并返回给处理器。
3. 缓存层次结构：为了更好地利用局部性原理，计算机系统通常采用多层次的缓存结构，例如 L1、L2、L3 缓存等。其中，L1 缓存通常位于处理器核心内部，速度最快但容量较小；L2 和 L3 缓存则通常位于处理器芯片上，速度介于 L1 和主存之间，容量比 L1 大；而主存则是速度较慢但容量较大的存储层次。
4. 替换策略：当缓存已满而需要替换其中的某些数据项时，缓存系统需要选择合适的替换策略。常见的替换策略包括随机替换 (random)、最近最少使用 (LRU)、先进先出 (FIFO)、最少使用 (LFU) 等。
5. 写策略：当缓存中的数据被修改时，需要决定何时将修改后的数据写回到主存储器中。常见的写策略包括写回 (Write Back) 和写直达 (Write Through)。

6. 一致性和可靠性：缓存系统必须确保数据的一致性和可靠性，即缓存中的数据必须与主存储器中的数据保持同步，而且在系统发生故障时不会丢失数据。

3.3 cache的映射策略

除上文介绍的内容外，映射策略对于cache也十分重要，它指定了如何将主存地址空间中的数据映射到缓存中的存储位置。下面介绍常见的几种映射策略。

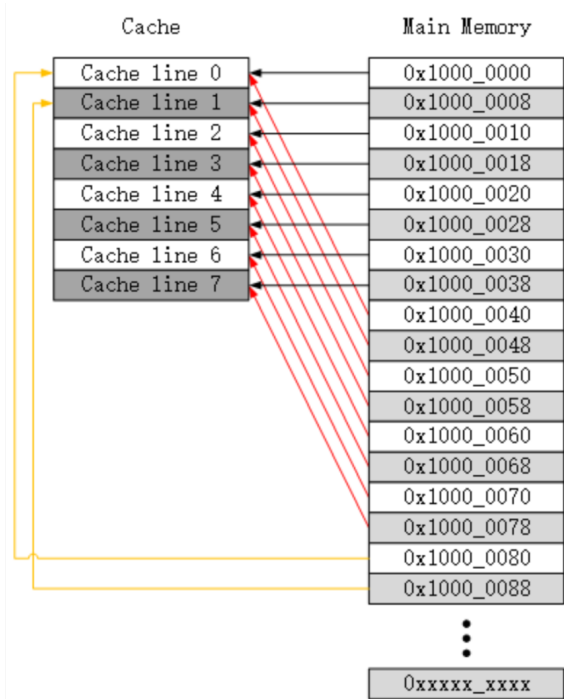
3.3.1 直接映射 (Direct Mapping)

直接映射缓存是一种简单而有效的缓存组织方式。其基本原理是将主存地址空间划分为若干个等大小的块，并且将每个块映射到缓存中的唯一一个位置，也就是说，每个主存块只能映射到缓存中的一个特定位置。

具体来说，主存地址被分为四个部分：有效位(Valid)、标签(Tag)、索引(Index)和块偏移(Block Offset)。有效位用于指示缓存行是否包含有效的数据，标签部分用于唯一标识主存中的块，索引部分用于确定主存块在缓存中的位置，而块偏移部分用于确定要访问块内的具体字节。

直接映射缓存通过将主存地址的索引部分直接作为缓存中的索引来实现映射。例如，如果缓存有 N 个位置，那么索引的范围通常是 0 到 N-1，主存地址的索引部分就会被映射到这个范围内的某个位置上。

具体如下图所示：



当要访问一个主存地址时，缓存控制器会首先将该地址的索引部分提取出来，然后在缓存中查找对应的位置。如果该位置的有效位为1（即代表有效）且与要访问的主存块的标签匹配，那么就发生了缓存命中，可以直接从缓存中读取或写入数据。否则，就会发生缓存未命中，需要将主存中的块加载到缓存中，并且替换掉当前位置的块（如果有的话）。

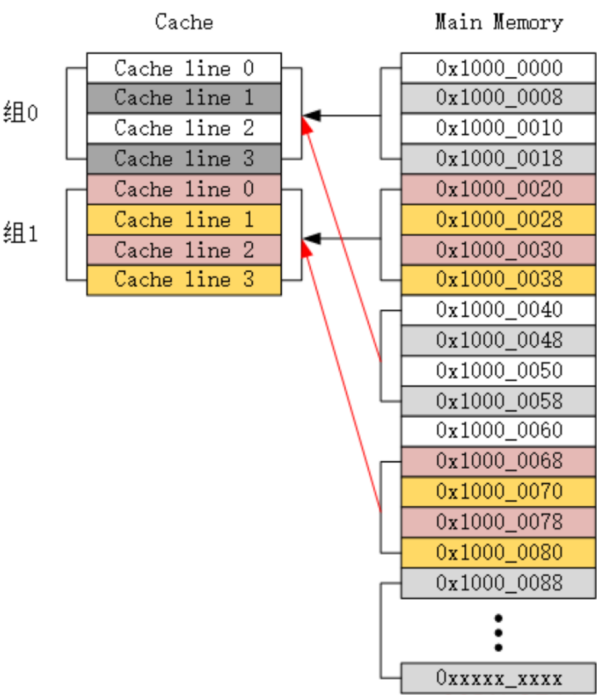
直接映射缓存的优点是实现简单、硬件开销小，并且能够有效地利用缓存的容量。缺点是可能会出现较高的缓存未命中率，特别是在访问具有相同索引但不同标签的主存块时，会发生冲突，导致频繁地替换缓存中的块。

3.3.2 组相联 (Set Associative Mapping)

组相联缓存是一种介于直接映射和全相联之间的缓存结构。它将缓存划分为多个组，每个组内有多个缓存行，而每个缓存行可以存储多个块。

具体来说，组相联缓存将主存地址划分为有效位(Valid)、标签(Tag)、索引(Index)和块偏移(Block Offset)四部分。与直接映射缓存不同的是，组索引用于选择组，块偏移用于指示数据块内的具体字节，标签部分对应到了内存块在一个组内的位置。每个组内有固定数量的缓存行，而索引部分用于确定主存块应该存储在哪个组内。

具体如下图所示：



当要访问一个主存地址时，缓存控制器首先提取出地址的索引部分，然后在对应的组内搜索。如果在组内找到了与要访问的主存块相匹配的缓存行，就发生了缓存命中，可以直接从缓存中读取或写入数据。如果组内没有匹配的缓存行，就会发生缓存未命中。

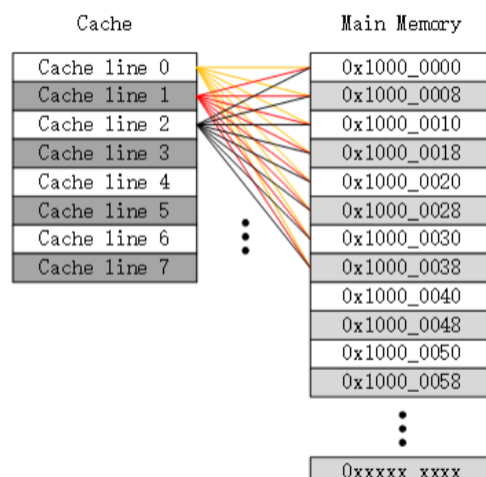
在组相联缓存中，未命中时有几种处理策略可供选择。常见的包括：随机替换、最近最少使用替换、先进先出替换，随后我们具体讲解这几种替换的原理。

组相联缓存的优点是相比直接映射，能够减少缓存冲突，从而降低缓存未命中率。相比全相联缓存，组相联缓存可以更好地利用硬件资源，并且通常能够提供较高的性能。然而，与直接映射缓存相比，组相联缓存的实现和硬件开销更高一些。

3.3.3 全相联 (Fully Associative Mapping)

全相联缓存允许任何主存块存储在缓存的任意位置，不需要事先分配特定的位置。这意味着每个主存块可以存储在任何一个缓存行中，因此没有固定的索引或组的概念。

具体如下图所示：



全相连缓存的基本原理是通过比较主存地址的标签(Tag)来确定缓存中是否存在对应的块。当要访问一个主存地址时，缓存控制器首先提取出地址的标签部分，然后与缓存中所有的标签进行比较。如果在缓存中找到了与要访问的主存块相匹配的标签，就发生了缓存命中，可以直接从缓存中读取或写入数据。如果没有找到匹配的标签，就会发生缓存未命中。

全相连缓存的优点是能够最大程度地减少缓存未命中率，因为每个主存块都有机会存储在缓存中的任意位置。然而，与直接映射和组相联缓存相比，全相连缓存的实现更复杂，需要更多的硬件资源来进行标签比较和替换策略的实现

3.4 cache的替换策略

缓存替换策略是用于决定在缓存中哪些数据应该被替换的算法。当缓存已满且需要为新数据腾出空间时，替换策略决定哪个现有的缓存块应该被淘汰，以便为新数据腾出位置。

我们学习到的缓存方法有：最近最少使用（LRU）策略、先进先出（FIFO）策略、随机替换（Random）策略、最不常用（LFU）策略，下面我们详细介绍一下这四个替换策略

3.4.1 最近最少使用（LRU）策略

LRU（Least Recently Used）缓存是一种常见的缓存淘汰策略，其原理基于“最近最少使用”的原则。当缓存空间不足时，LRU缓存会淘汰最近最久未被使用的数据，以确保缓存中始终存储着最新和最频繁使用的数据。

实现LRU缓存的基本原理可以通过维护一个有序的访问队列来实现。LRU策略会访问顺序维护，LRU缓存维护一个数据访问的顺序，通常使用双向链表或其他数据结构来记录数据的访问顺序。当数据被访问时，将其移动到队列的头部，表示它是最近被使用的数据。LRU的淘汰策略为当缓存空间不足时，需要淘汰队列尾部的数据，因为它们是最久未被使用的数据。淘汰的数据通常是队列尾部的数据，即最近最少使用的数据。当新数据被插入缓存时，将其放在队列的头部，表示它是最新被使用的数据。

3.4.2 先进先出（FIFO）策略

FIFO（First In First Out）是一种常见的缓存替换策略，也被用于队列（Queue）等数据结构中。让我为你详细解释一下FIFO策略的原理：FIFO策略的核心原则是，如果一个数据最先进入缓存中，则应该最早被淘汰掉。这意味着最早进入缓存的数据将最早被替换出去，以便为新数据腾出空间。

1. FIFO通常使用一个双向链表或其他数据结构来维护数据的访问顺序。当数据被访问时，将其插入到队列的尾部，表示它是最近被使用的数据。如果缓存已满，需要淘汰队列头部的数据，因为它们是最久未被使用的数据。数据插入需要新访问的数据插入FIFO队列的尾部。数据淘汰需要如果缓存已满，将队列头部的数据删除。

3.4.3 随机替换 (Random) 策略

随机替换 (Random Replacement) 策略是一种缓存替换算法，其核心思想是随机选择要被替换的缓存块，从而保证所有缓存块被替换的概率相等。在缓存空间有限的情况下，当需要替换缓存中的某个数据块时，随机替换算法会从当前缓存中随机选择一个数据块进行替换。随机算法实现简单，容易理解和实现。在缓存大小较大时表现良好，能够减少缓存替换的次数，提高缓存命中率。但随机算法性能不稳定，在缓存大小较小时，表现较差。因为随机替换可能导致频繁的缓存替换，降低了缓存的命中率。无法适应不同数据访问模式的需求，不能利用数据局部性进行缓存优化。

3.4.4 最不常用 (LFU) 策略

LFU (Least Frequently Used) 是一种缓存替换算法，其核心思想是根据数据的历史访问频率来淘汰数据。LFU 算法的基本思想和所有的缓存算法一样，都是基于局部性原理：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。LFU 是基于这种思想进行设计：一定时期内被访问次数最少的页，在将来被访问到的几率也是最小的。LFU 需要记录每个数据被访问的次数。当需要替换缓存中的某个数据块时，LFU 会选择访问次数最少的数据进行淘汰。LFU 执行过程先在缓存中查找客户端需要访问的数据。如果缓存命中，则将访问的数据从队列中取出，并将数据对应的频率计数加 1，然后将其放到频率相同的数据队列的头部。如果没有命中，表示缓存穿透，将需要访问的数据从磁盘中取出，加入到缓存队列的尾部，记频率为 1。如果此时缓存满了，则需要先置换出去一个数据，淘汰队列尾部频率最小的数据，然后再在队列尾部加入新数据。

4 实验内容

4.1 获得正确的输出

4.1.1 构建 gem5 时遇到的问题

首先构建 gem5，构建过程中有数次链接操作，链接器需要使用大量内存，观测到的峰值为 7GiB，当虚拟机物理内存与交换空间之和小于 8GiB 时容易发生 Out Of Memory 错误使链接器被 Linux Kernel 的 OOM Killer 杀掉从而导致编译错误。

4.1.2 编译 ARM 架构下的 ntt 可执行文件

要想在 x86_64 宿主 (host) 下编译目标 (target) 为 ARM 的 ntt 可执行文件，需要交叉编译工具链。先观察 gem5 附带的 ARM 架构 hello 可执行文件：

```
1 $ file tests/test-progs/hello/bin/arm/linux/hello
2 tests/test-progs/hello/bin/arm/linux/hello: ELF 32-bit LSB executable, ARM,
  EABI4 version 1 (SYSV), statically linked, for GNU/Linux 2.6.16, with
  debug_info, not stripped
```

发现该可执行文件为 32 位 ARM 架构，静态链接，EABI 版本 4，最低需要 GNU/Linux 2.6.16 版本内核，猜测其编译器版本较旧。

首先使用 ARM 官方提供的 aarch64 工具链进行尝试：

```
1 $ aarch64-linux-gnu-gcc -v
2 Using built-in specs.
3 COLLECT_GCC=aarch64-linux-gnu-gcc
4 COLLECT_LTO_WRAPPER=/usr/lib/gcc/aarch64-linux-gnu/13.2.0/lto-wrapper
5 Target: aarch64-linux-gnu
6 .....
7 Thread model: posix
8 Supported LTO compression algorithms: zlib zstd
9 gcc version 13.2.0 (GCC)
10
11 $ aarch64-linux-gnu-gcc --static -o ntt ./ntt.c
12 $ file ntt
13 ntt: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux),
  statically linked, BuildID[sha1]=b090d11b1030401f870aa947d2399b1e46968a29,
  for GNU/Linux 3.7.0, with debug_info, not stripped
14
15 $ ./build/ARM/gem5.opt --outdir=memaccess --debug-flag=MemoryAccess --debug-
  file=MemoryAccess.out ./configs/deprecated/example/se.py -c ntt
16 gem5 Simulator System. https://www.gem5.org
17 .....
18 **** REAL SIMULATION ****
19 src/sim/simulate.cc:199: info: Entering event queue @ 0. Starting
  simulation...
20 src/sim/syscall_emul.cc:74: warn: ignoring syscall set_robust_list(...)
21 src/sim/syscall_emul.cc:85: warn: ignoring syscall rseq(...)
22 (further warnings will be suppressed)
23 src/sim/mem_state.cc:448: info: Increasing stack size by one page.
```

```
24 src/arch/arm/insts/pseudo.cc:172: warn:      instruction 'bti'
    unimplemented
25 src/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
26 Exiting @ tick 334650000 because exiting with last active thread context
```

发现运行错误，更换为 32 位 ARM 架构编译器进行尝试：

```
1  $ arm-none-linux-gnueabi-gcc -v
2  Using built-in specs.
3  COLLECT_GCC=./arm-gnu-toolchain-13.2.Rel1-x86_64-arm-none-linux-
    gnueabi/bin/arm-none-linux-gnueabi-gcc
4  COLLECT_LTO_WRAPPER=/home/lwzheng/foss/gem5/arm-gnu-toolchain-13.2.Rel1-
    x86_64-arm-none-linux-gnueabi/bin/./libexec/gcc/arm-none-linux-
    gnueabi/13.2.1/lto-wrapper
5  Target: arm-none-linux-gnueabi
6  .....
7  Thread model: posix
8  Supported LTO compression algorithms: zlib
9  gcc version 13.2.1 20231009 (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7))
10
11 $ arm-none-linux-gnueabi-gcc --static -o ntt ./ntt.c
12 $ file ntt
13 ntt: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
    linked, for GNU/Linux 3.2.0, with debug_info, not stripped
14
15 $ ./build/ARM/gem5.opt --outdir=memaccess --debug-flag=MemoryAccess --debug-
    file=MemoryAccess.out ./configs/deprecated/example/se.py -c ntt
16 .....
17 **** REAL SIMULATION ****
18 src/sim/simulate.cc:199: info: Entering event queue @ 0. Starting
    simulation...
19 src/sim/syscall_emul.cc:74: warn: ignoring syscall set_robust_list(...)
20 src/sim/syscall_desc.hh:209: fatal: Syscall 398 out of range
21 Memory Usage: 688836 KBytes
```

仍然出现了错误，错误为 398 号系统调用超出范围，猜测是较新的编译工具链对 ABI 做了修改，导致 gem5 模拟器无法正确模拟系统调用，遂换用较旧的编译器：

```
1  $ arm-linux-gnueabi-gcc -v
2  Using built-in specs.
3  COLLECT_GCC=./gcc-arm-8.2-2018.08-x86_64-arm-linux-gnueabi/bin/arm-linux-
    gnueabi-gcc
4  COLLECT_LTO_WRAPPER=/home/lwzheng/foss/gem5/gcc-arm-8.2-2018.08-x86_64-arm-
    linux-gnueabi/bin/./libexec/gcc/arm-linux-gnueabi/8.2.1/lto-wrapper
5  Target: arm-linux-gnueabi
6  .....
7  Thread model: posix
8  gcc version 8.2.1 20180802 (GNU Toolchain for the A-profile Architecture
    8.2-2018-08 (arm-rel-8.23))
9
10 $ arm-linux-gnueabi-gcc --static -o ntt ./ntt.c
11 $ file ntt
```

```

12 ntt: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
    linked, for GNU/Linux 3.2.0, with debug_info, not stripped
13
14 $ ./build/ARM/gem5.opt --outdir=memaccess --debug-flag=MemoryAccess --debug-
    file=MemoryAccess.out ./configs/deprecated/example/se.py -c ntt
15 .....
16 **** REAL SIMULATION ****
17 src/sim/simulate.cc:199: info: Entering event queue @ 0. Starting
    simulation...
18 src/sim/mem_state.cc:448: info: Increasing stack size by one page.
19 Exiting @ tick 1815544000 because exiting with last active thread context

```

本次即可正确运行，获得正确的输出。

运行实验任务中的三条命令，获得三个输出文件 dram.out, mmu.out 与 MemoryAccess.out。

4.1.3 处理输出

观察 MemoryAccess.out，最开始的部分是由于缺页中断使 gem5 向内存中写入可执行文件，类型为 Write from functional，不属于本次实验需要考虑的部分。从该部分结束后开始分析。

缺页中断导致的写入结束后，出现了格式如下的内存访问 trace 信息：

```

1 0: global: IFetch from cpu.inst of size 4 on address 0x35c data 0xe3a0b000
  C
2 1000: global: Read from cpu.data of size 4 on address 0x6af30 data 0x1 C
3 2500: global: Write from cpu.data of size 4 on address 0x6af30 data 0xbffffff34
  C

```

其中 IFetch 为取指，Read/Write 为内存读写，这里主要关注内存读写，写出以下脚本对 MemoryAccess.out 做处理：

```

1 if __name__ == "__main__":
2     with open("MemoryAccess.out", "r") as f:
3         lines = f.readlines()
4         for line in lines:
5             data = ""
6             sline = line.strip().split(" ")
7             if len(line) > 1:
8                 if sline[1] == 'global:' and sline[0] != '0:':
9                     if sline[2] == 'Read' or sline[2] == 'write':
10                        print( f"{sline[10]} {"r" if sline[2]=="Read" else
                            "w"}" )

```

即可获得如下格式的内存访问序列：

```

1 0x6af30 r
2 0x6af30 w
3 0x6af2c w
4 .....

```

其中每行一个内存地址，末尾为 r 表示读，为 w 表示写，每次操作数据大小均为 4 字节。

4.2 编写cache模拟器

根据实验要求，编写的cache模拟器要具有以下功能：

- 命中检查机制，即检查miss/hit的情况。
- 内存地址和cache块的映射（全相联、组相联、直接映射）的实现，即映射方式。
- 一些替换算法（Random,LRU），即不同的替换策略。
- 以获得的内存地址访问序列文件为输入，即读取内存地址序列。
- 统计miss/hit情况，即统计cache命中率。
- 调整相关参数（cache块大小，cache大小），即cache及其块大小可以改变。

根据以上要求，我们决定先设计cache模拟器的处理地址文件的功能，因为要求cache的大小可变，且根据cache模拟器的特点，我们决定用类class去实现，初始化时可以调整其大小，之后设计不同的映射方式，再根据不同的映射方式实现不同的替换策略，同时设计命中检查机制，最后统计其miss/hit情况。

实现cache模拟器的基本思路如下图所示：

暂时无法在飞书文档外展示此内容

下面将根据此图详细介绍每部分的实现思路以及过程。我们组用Python与C++两种语言分别实现了cache模拟器，但实现思路基本相同，故下面介绍时两种实现的思路不再重复描述，在介绍时采用的示例为C++版本，Python版本在附录中。

4.2.1 处理地址序列文件

我们设计函数读取内存序列地址，文件中每一行是一个内存地址，函数将逐行读取地址，由于内存地址在文件中是以0x123abc的十六进制的字符串形式存储的，所以函数要将其处理为整数并传到cache中进行检查是否命中。由于函数是逐行读取检查，因此我们可以统计每次的命中情况来实现统计命中率。

以下是C++的具体实现示例：

```
1 // 处理地址文件的函数
2 void processAddressFile(const string& filename, DirectMappedCacheSimulator&
   simulator) {
3     ifstream file(filename); // 打开地址文件
4     string line;
5     int hitCount = 0; // 命中的地址数量
6     int totalCount = 0; // 总的地址数量
7
8     if (file.is_open()) {
9         while (getline(file, line)) { // 逐行读取地址
10             istringstream iss(line);
11             int address;
12             iss >> hex >> address; // 将十六进制的字符串转换为整数
13             if (simulator.isHit(address)) { // 如果地址在cache中
14                 hitCount++;
15             }
16             totalCount++;
17         }
18         file.close(); // 关闭文件
19
20         // 输出命中率
```

```

21         cout << "Hit rate: " << static_cast<double>(hitCount) / totalCount
    << endl;
22     }
23
24     else {
25         cerr << "Failed to open file: " << filename << endl; // 输出错误信息
26     }
27
28 }

```

4.2.2 通过类class实现cache

因为要求cache的大小以及cache块的大小可变，且根据其特点，我们决定用类实现。

由于cache是由许多个cache line组成的，一个cache line有若干个cache block（也就是实验中的cache块），所以我们决定将cache和cache block分别用类实现。

4.2.2.1 Cache line

cache block的大小会影响cache line的内存块偏移量，由于在本实验中并不真正涉及cache block的取地址等操作，因此为了方便且与实验的要求相同，我们将省略掉cache block的操作，直接实现cache line。cache line中包括有效位valid，标签位tag以及储存块偏移量的blockoffset，但因为在检查命中的时候只需要index以及tag相同即表示命中，并不需要计算出偏移地址等操作，故此处我们为了简化操作改为存储每个cache block的块大小表示cache block的大小可改变，而索引位index则在实现cache的时候存储。

以下是C++的cache line具体实现示例（不同替换策略会有改动，此处只是基础功能实现）：

```

1  class CacheLine {
2  public:
3      bool valid; // 有效位
4      int tag; // 标签位
5      vector<int> data; // 存储对应数据块，由于该模拟器只模拟读写操作，故不需要存储数据，
        所以后续无用处，在此只存储数据块大小
6      CacheLine(int blockSize) : valid(false), tag(0), data(blockSize) {} // 构
        造函数，初始化有效位、标签位和数据块
7  };

```

4.2.2.2 Cache

我们通过建立cache line型的数组来实现cache，数组中每个元素对应的是一个cache line，索引对应cache line的索引index（或是cache line set的组索引index），由于不同映射方式具体的cache与cache line之间的关系不同，所以我们在此只介绍相同部分，不同部分后续说明。在检查命中的时候会提取地址的不同位数进行匹配操作，以及存储的时候要存储标签tag，所以我们还需要计算出indexBits索引位数blockOffsetBits块偏移位数来方便后续操作，同时存储blockSize块大小方便计算cache line的索引index。在每次初始化cache及cache块的大小后，cache计算出cache line的数量（或是cache line set的数量），建立相应大小的cache line数组，数组索引即为index，同时计算出indexBits索引位数blockOffsetBits块偏移位数。

以下是C++的cache具体实现示例（此处只是基础功能实现）：

```

1  class Cache {
2  private:
3      vector<CacheLine> cache; // 创建一个cacheline数组模拟cache

```



```

4     int blockOffsetBits; // 块偏移位数
5     int indexBits; // 索引位数
6     int blockSize; // 内存块大小，用于后续操作
7
8     public:
9         // 构造函数，初始化cache大小和块大小
10        Cache(int cacheSize, int blockSize) : blockSize(blockSize) {
11            int blockCount = cacheSize / blockSize; // 计算cacheline的数量
12            cache.resize(blockCount, CacheLine(blockSize)); // 初始化cache数组
13            blockOffsetBits = 0;
14            // 计算内存地址中块偏移位数
15            while (blockSize > 1) {
16                blockSize >>= 1;
17                blockOffsetBits++;
18            }
19            indexBits = 0;
20            // 计算内存地址中索引位数
21            while (blockCount > 1) {
22                blockCount >>= 1;
23                indexBits++;
24            }
25        }

```

同时cache中还需要有地址序列命中检查机制以及cache line的替换等功能，为此我们设计类函数来实现cache的基本功能。

- 命中检查机制

由于命中检查机制与cache的读写操作是一体的，而不同替换策略的读写操作不同，故在此只介绍基本的命中检查机制，对替换机制不做介绍。

我们此前已经计算出indexBits索引位数blockOffsetBits块偏移位数，对于传入的地址，我们提取出该地址对应的index以及tag方便后续的匹配。我们首先根据地址的index位定位到对应cache的cache line，检查该行有效位是否有效，若有效标签位tag是否匹配，若匹配则说明cache命中，返回true代表hit；否则就是未命中，此时需要更新cache，将该cache line的对应tag位进行修改并将有效位valid设为true有效，返回false表示miss。

以下是C++的cache命中检查机制具体实现示例（此处只是基础功能实现）：

```

1 // 检查一个地址是否在cache中
2 bool isHit(int address) {
3     // 计算索引和标签
4     int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
5     int tag = address >> (blockOffsetBits + indexBits);
6
7     // 如果有效位valid为1并且标签tag位匹配，那么就是命中
8     if (cache[index].valid && cache[index].tag == tag) {
9         cout << hex << address << " hit" << endl;
10        return true;
11    }
12    // 否则就是未命中，并且更新cacheline的有效位和标签tag
13    else {
14        cout << hex << address << " miss" << endl;
15        cache[index].valid = true;
16        cache[index].tag = tag;
17        return false;

```



```
18     }
19 }
```

由此我们实现了cache模拟器最基本的操作。

4.2.3 映射方式

4.2.3.1 直接映射

直接映射即是cache模拟器最基本操作的实现，没有复杂的替换策略，只需将前面的基础实现进行整合。

以下是C++的直接映射具体实现示例：

```
1  #include <vector>
2  #include <fstream>
3  #include <iostream>
4  #include <string>
5  #include <sstream>
6  using namespace std;
7
8  class CacheLine {
9  public:
10     bool valid; // 有效位
11     int tag; // 标签位
12     vector<int> data; // 存储对应数据块，由于该模拟器只模拟读写操作，故不需要存储数据，
    所以后续无用处，在此只存储数据块大小
13     CacheLine(int blockSize) : valid(false), tag(0), data(blockSize) {} // 构造函数，初始化有效位、标签位和数据块
14 };
15
16 class DirectMappedCacheSimulator {
17 private:
18     vector<CacheLine> cache; // 创建一个cacheline数组模拟cache
19     int blockOffsetBits; // 块偏移位数
20     int indexBits; // 索引位数
21     int blockSize; // 内存块大小，用于后续操作
22
23 public:
24     // 构造函数，初始化cache大小和块大小
25     DirectMappedCacheSimulator(int cacheSize, int blockSize) :
    blockSize(blockSize) {
26         int blockCount = cacheSize / blockSize; // 计算cacheline的数量
27         cache.resize(blockCount, CacheLine(blockSize)); // 初始化cache数组
28         blockOffsetBits = 0;
29         // 计算内存地址中块偏移位数
30         while (blockSize > 1) {
31             blockSize >>= 1;
32             blockOffsetBits++;
33         }
34         indexBits = 0;
35         // 计算内存地址中索引位数
36         while (blockCount > 1) {
37             blockCount >>= 1;
38             indexBits++;
```

```

39     }
40 }
41
42 // 检查一个地址是否在cache中
43 bool isHit(int address) {
44     // 计算索引和标签
45     int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
46     int tag = address >> (blockOffsetBits + indexBits);
47
48     // 如果有效位valid为1并且标签tag位匹配，那么就是命中
49     if (cache[index].valid && cache[index].tag == tag) {
50         cout << hex << address << " hit" << endl;
51         return true;
52     }
53     else {
54         // 否则就是未命中，并且更新cacheline的有效位和标签tag
55         cout << hex << address << " miss" << endl;
56         cache[index].valid = true;
57         cache[index].tag = tag;
58         return false;
59     }
60 }
61 };
62
63 // 处理地址文件的函数
64 void processAddressFile(const string& filename, DirectMappedCacheSimulator&
simulator) {
65     ifstream file(filename); // 打开地址文件
66     string line;
67     int hitCount = 0; // 命中的地址数量
68     int totalCount = 0; // 总的地址数量
69
70     if (file.is_open()) {
71         while (getline(file, line)) { // 逐行读取地址
72             istringstream iss(line);
73             int address;
74             iss >> hex >> address; // 将十六进制的字符串转换为整数
75             if (simulator.isHit(address)) { // 如果地址在cache中
76                 hitCount++;
77             }
78             totalCount++;
79         }
80         file.close(); // 关闭文件
81
82         // 输出命中率
83         cout << "Hit rate: " << static_cast<double>(hitCount) / totalCount
<< endl;
84     }
85
86     else {
87         cerr << "Failed to open file: " << filename << endl; // 输出错误信息
88     }
89
90 }
91
92 int main() {

```

```

93     DirectMappedCacheSimulator simulator(1024, 4); // 创建一个容量为1024，块大小
    为4的缓存模拟器
94     processAddressFile("data_.txt", simulator); // 处理地址文件
95     return 0;
96 }

```

4.2.3.2 组相联

组相联将缓存划分为多个组，每个组内有多个缓存行。为实现该功能，我们决定利用二维数组实现，一级索引表示组索引，二级索引表示每组中cache line的个数，同时我们此时需要计算组的个数而不是cache line的个数。

组相联的命中检查机制也有所更改，我们首先根据地址的组索引index找到对应的组，再在该组内检查cache line是否有效，若有效与不同的cache line进行tag匹配，若匹配则命中，否则为未命中进行替换操作，组相联有更多复杂的替换策略，我们在此暂不说明。

以下是C++的组相联具体实现示例（以LRU策略为例）：

```

1  class SetAssociativeCacheSimulator {
2  private:
3      vector<vector<CacheLine>> cache; // 创建一个二维向量模拟cache
4      int blockOffsetBits; // 块偏移位数
5      int indexBits; // 索引位数
6      int blockSize; // 内存块大小，用于后续操作
7      int sets; // 每个组中的块数
8      int currentTime; // 当前时间，用于更新lastAccessed
9
10 public:
11     // 构造函数，初始化cache大小和块大小
12     SetAssociativeCacheSimulator(int cacheSize, int blockSize, int sets) :
    blockSize(blockSize), sets(sets), currentTime(0) {
13         int groupCount = cacheSize / (blockSize * sets); // 计算组的数量
14         cache.resize(groupCount, vector<CacheLine>(sets,
    CacheLine(blockSize))); // 初始化cache数组
15         blockOffsetBits = 0;
16         // 计算内存地址中块偏移位数
17         while (blockSize > 1) {
18             blockSize >>= 1;
19             blockOffsetBits++;
20         }
21         indexBits = 0;
22         // 计算内存地址中索引位数
23         while (groupCount > 1) {
24             groupCount >>= 1;
25             indexBits++;
26         }
27     }
28
29     // 检查一个地址是否在cache中
30     bool isHit(int address) {
31         // 计算索引和标签
32         int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
33         int tag = address >> (blockOffsetBits + indexBits);
34
35         // 在给定索引的所有块中查找标签匹配的块

```

```

36         for (int i = 0; i < sets; i++) {
37             if (cache[index][i].valid && cache[index][i].tag == tag) {
38                 //cout << hex << address << " hit" << endl;
39                 cache[index][i].lastAccessed = ++currentTime; // 更新最后访问时
间
40                 return true;
41             }
42         }
43
44         // 如果未找到匹配的块, 选择一个块来替换
45         // 使用LRU策略选择最长时间未被访问的块
46         //cout << hex << address << " miss" << endl;
47         int lruIndex = 0;
48         for (int i = 1; i < sets; i++) {
49             if (cache[index][i].lastAccessed < cache[index]
[1ruIndex].lastAccessed) {
50                 lruIndex = i;
51             }
52         }
53         cache[index][lruIndex].valid = true;
54         cache[index][lruIndex].tag = tag;
55         cache[index][lruIndex].lastAccessed = ++currentTime; // 更新最后访问时
间
56         return false;
57     }
58 };

```

4.2.3.3 全相联

全相联没有固定的索引或组的概念, 因此我们取消索引位index (或是改为组相联中路的个数=cache line的数量, 即将全部的cache line存储在一组中), 因此我们不再需要计算index位以及indexOffset位, 检查命中时依次检查有效位valid是否有效, 然后比较tag位进行匹配, 若匹配则命中, 否则为miss进行替换操作 (同组相联在此暂不说明)。

以下是C++的全相联具体实现示例 (以LRU策略为例) :

```

1  class FullyAssociativeCacheSimulator {
2  private:
3      vector<CacheBlock> cache; // 创建一个一维向量模拟cache
4      int blockOffsetBits; // 块偏移位数
5      int blockSize; // 内存块大小, 用于后续操作
6      int sets; // 缓存中的块数
7      int currentTime; // 当前时间, 用于更新lastAccessed
8
9  public:
10     // 构造函数, 初始化cache大小和块大小
11     FullyAssociativeCacheSimulator(int cacheSize, int blockSize) :
        blockSize(blockSize), currentTime(0) {
12         sets = cacheSize / blockSize; // 计算总的块数
13         cache.resize(sets, CacheBlock(blockSize)); // 初始化cache数组
14         blockOffsetBits = 0;
15         // 计算内存地址中块偏移位数
16         while (blockSize > 1) {
17             blockSize >>= 1;
18             blockOffsetBits++;

```

```

19     }
20 }
21
22 // 检查一个地址是否在cache中
23 bool isHit(int address) {
24     // 计算标签
25     int tag = address >> blockOffsetBits;
26
27     // 在所有块中查找标签匹配的块
28     for (int i = 0; i < sets; i++) {
29         if (cache[i].valid && cache[i].tag == tag) {
30             //cout << hex << address << " hit" << endl;
31             cache[i].lastAccessed = ++currentTime; // 更新最后访问时间
32             return true;
33         }
34     }
35
36     // 如果未找到匹配的块，选择一个块来替换
37     // 使用LRU策略选择最长时间未被访问的块
38     //cout << hex << address << " miss" << endl;
39     int lruIndex = 0;
40     for (int i = 1; i < sets; i++) {
41         if (cache[i].lastAccessed < cache[lruIndex].lastAccessed) {
42             lruIndex = i;
43         }
44     }
45     cache[lruIndex].valid = true;
46     cache[lruIndex].tag = tag;
47     cache[lruIndex].lastAccessed = ++currentTime; // 更新最后访问时间
48     return false;
49 }
50 };

```

4.2.4 替换策略

因为只有组相联与全相联有不同的替换策略，且全相联相当于一组的组相联，因此以下只用组相联进行具体实现示例展示。

4.2.4.1 最近最少使用 (LRU) 策略

由于我们需要根据每个cache line的最近访问时间来进行替换判断，因此我们需要在cache中添加时间机制来记录他们的最近访问时间。

首先在CacheLine中添加lastAccessed变量表示该行的最后访问时间，在Cache中添加currentTime表示当前时间，用于更新lastAccessed。将所有cache line的lastAccessed初始化为0，cache的currentTime初始设置为0，在每次未命中进行替换后或者命中取地址的时候，将lastAccessed赋值为++currentTime表示更新最后访问时间。在替换前，遍历该组找出其中lastAccessed最小的cache line，即最少使用的cache line，之后进行替换操作。

以下是C++的LRU策略具体实现示例（以组相联为例）：

```

1 class CacheBlock {
2 public:
3     bool valid; // 有效位
4     int tag; // 标签位

```

```

5     vector<int> data; // 存储对应数据块
6     int lastAccessed; // 最后访问时间
7     CacheBlock(int blockSize) : valid(false), tag(0), data(blockSize),
lastAccessed(0) {} // 构造函数，初始化有效位、标签位、数据块和最后访问时间
8 };
9
10 class SetAssociativeCacheSimulator {
11 private:
12     vector<vector<CacheBlock>> cache; // 创建一个二维向量模拟cache
13     int blockOffsetBits; // 块偏移位数
14     int indexBits; // 索引位数
15     int blockSize; // 内存块大小，用于后续操作
16     int sets; // 每个组中的块数
17     int currentTime; // 当前时间，用于更新lastAccessed
18
19 public:
20     // 构造函数，初始化cache大小和块大小
21     SetAssociativeCacheSimulator(int cacheSize, int blockSize, int sets) :
blockSize(blockSize), sets(sets), currentTime(0) {
22         int groupCount = cacheSize / (blockSize * sets); // 计算组的数量
23         cache.resize(groupCount, vector<CacheBlock>(sets,
CacheBlock(blockSize))); // 初始化cache数组
24         blockOffsetBits = 0;
25         // 计算内存地址中块偏移位数
26         while (blockSize > 1) {
27             blockSize >>= 1;
28             blockOffsetBits++;
29         }
30         indexBits = 0;
31         // 计算内存地址中索引位数
32         while (groupCount > 1) {
33             groupCount >>= 1;
34             indexBits++;
35         }
36     }
37
38     // 检查一个地址是否在cache中
39     bool isHit(int address) {
40         // 计算索引和标签
41         int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
42         int tag = address >> (blockOffsetBits + indexBits);
43
44         // 在给定索引的所有块中查找标签匹配的块
45         for (int i = 0; i < sets; i++) {
46             if (cache[index][i].valid && cache[index][i].tag == tag) {
47                 //cout << hex << address << " hit" << endl;
48                 cache[index][i].lastAccessed = ++currentTime; // 更新最后访问时
间
49                 return true;
50             }
51         }
52
53         // 如果未找到匹配的块，选择一个块来替换
54         // 使用LRU策略选择最长时间未被访问的块
55         //cout << hex << address << " miss" << endl;
56         int lruIndex = 0;

```

```

57         for (int i = 1; i < sets; i++) {
58             if (cache[index][i].lastAccessed < cache[index]
[1ruIndex].lastAccessed) {
59                 1ruIndex = i;
60             }
61         }
62         cache[index][1ruIndex].valid = true;
63         cache[index][1ruIndex].tag = tag;
64         cache[index][1ruIndex].lastAccessed = ++currentTime; // 更新最后访问时
间
65         return false;
66     }
67 };

```

4.2.4.2 先进先出 (FIFO) 策略

与LRU策略类似，由于我们需要根据每个cache line的最近进入cache的时间来进行替换判断，因此我们需要在cache中添加时间机制来记录他们的最后进入时间。

首先在CacheLine中添加enTime变量表示该行的最后进入的时间，在Cache中添加currentTime表示当前时间，用于更新enTime。将所有cache line的enTime初始化为0，cache的currentTime初始设置为0，不同于LRU策略，FIFO策略只需要在每次未命中进行替换后，将enTime赋值为++currentTime表示更新最后进入时间。在替换前，遍历该组找出其中enTime最小的cache line，即最先进入的cache line，之后进行替换操作。

以下是C++的FIFO策略具体实现示例（以组相联为例）：

```

1  class CacheBlock {
2  public:
3      bool valid; // 有效位
4      int tag; // 标签位
5      vector<int> data; // 存储对应数据块
6      int enTime; // 最后进入时间
7      CacheBlock(int blockSize) : valid(false), tag(0), data(blockSize),
enTime(0) {} // 构造函数，初始化有效位、标签位、数据块和最后访问时间
8  };
9
10 class SetAssociativeCacheSimulator {
11 private:
12     vector<vector<CacheBlock>> cache; // 创建一个二维向量模拟cache
13     int blockOffsetBits; // 块偏移位数
14     int indexBits; // 索引位数
15     int blockSize; // 内存块大小，用于后续操作
16     int sets; // 每个组中的块数
17     int currentTime; // 当前时间，用于更新enTime
18
19 public:
20     // 构造函数，初始化cache大小和块大小
21     SetAssociativeCacheSimulator(int cacheSize, int blockSize, int sets) :
blockSize(blockSize), sets(sets), currentTime(0) {
22         int groupCount = cacheSize / (blockSize * sets); // 计算组的数量
23         cache.resize(groupCount, vector<CacheBlock>(sets,
CacheBlock(blockSize))); // 初始化cache数组
24         blockOffsetBits = 0;
25         // 计算内存地址中块偏移位数

```

```

26     while (blockSize > 1) {
27         blockSize >>= 1;
28         blockOffsetBits++;
29     }
30     indexBits = 0;
31     // 计算内存地址中索引位数
32     while (groupCount > 1) {
33         groupCount >>= 1;
34         indexBits++;
35     }
36 }
37
38 // 检查一个地址是否在cache中
39 bool isHit(int address) {
40     // 计算索引和标签
41     int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
42     int tag = address >> (blockOffsetBits + indexBits);
43
44     // 在给定索引的所有块中查找标签匹配的块
45     for (int i = 0; i < sets; i++) {
46         if (cache[index][i].valid && cache[index][i].tag == tag) {
47             //cout << hex << address << " hit" << endl;
48             return true;
49         }
50     }
51
52     // 如果未找到匹配的块，选择一个块来替换
53     // 使用FIFO策略选择最长时间未被访问的块
54     //cout << hex << address << " miss" << endl;
55     int fifoIndex = 0;
56     for (int i = 1; i < sets; i++) {
57         if (cache[index][i].enTime < cache[index][fifoIndex].enTime) {
58             fifoIndex = i;
59         }
60     }
61     cache[index][fifoIndex].valid = true;
62     cache[index][fifoIndex].tag = tag;
63     cache[index][fifoIndex].enTime = ++currentTime;
64     return false;
65 }
66 };

```

4.2.4.3 最不常用 (LFU) 策略

由于我们需要根据每个cache line的最近访问次数来进行替换判断，因此我们需要在cache中添加访问记录机制来记录他们的最近访问次数。

首先在CacheLine中添加accessCount变量表示该行的访问次数，将所有cache line的accessCount初始化为0，在每次命中取地址的时候，将accessCount++表示更新该行的访问次数，未命中进行替换后，将accessCount赋值为1表示重新初始化该行的访问次数。在替换前，遍历该组找出其中accessCount最小的cache line，即最不常使用的cache line，之后进行替换操作。

以下是C++的LFU策略具体实现示例（以组相联为例）：

```

1 class CacheBlock {

```



```

2 public:
3     bool valid; // 有效位
4     int tag; // 标签位
5     vector<int> data; // 存储对应数据块
6     int accessCount; // 访问次数
7     CacheBlock(int blockSize) : valid(false), tag(0), data(blockSize),
    accessCount(0) {} // 构造函数，初始化有效位、标签位、数据块和最后访问时间
8 };
9
10 class SetAssociativeCacheSimulator {
11 private:
12     vector<vector<CacheBlock>> cache; // 创建一个二维向量模拟cache
13     int blockOffsetBits; // 块偏移位数
14     int indexBits; // 索引位数
15     int blockSize; // 内存块大小，用于后续操作
16     int sets; // 每个组中的块数
17
18 public:
19     // 构造函数，初始化cache大小和块大小
20     SetAssociativeCacheSimulator(int cacheSize, int blockSize, int sets) :
    blockSize(blockSize), sets(sets) {
21         int groupCount = cacheSize / (blockSize * sets); // 计算组的数量
22         cache.resize(groupCount, vector<CacheBlock>(sets,
    CacheBlock(blockSize))); // 初始化cache数组
23         blockOffsetBits = 0;
24         // 计算内存地址中块偏移位数
25         while (blockSize > 1) {
26             blockSize >>= 1;
27             blockOffsetBits++;
28         }
29         indexBits = 0;
30         // 计算内存地址中索引位数
31         while (groupCount > 1) {
32             groupCount >>= 1;
33             indexBits++;
34         }
35     }
36
37     // 检查一个地址是否在cache中
38     bool isHit(int address) {
39         // 计算索引和标签
40         int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
41         int tag = address >> (blockOffsetBits + indexBits);
42
43         // 在给定索引的所有块中查找标签匹配的块
44         for (int i = 0; i < sets; i++) {
45             if (cache[index][i].valid && cache[index][i].tag == tag) {
46                 //cout << hex << address << " hit" << endl;
47                 cache[index][i].accessCount++;
48                 return true;
49             }
50         }
51
52         // 如果未找到匹配的块，选择一个块来替换
53         // 使用LFU策略选择最长时间未被访问的块
54         //cout << hex << address << " miss" << endl;

```

```

55     int lfuIndex = 0;
56     for (int i = 1; i < sets; i++) {
57         if (cache[index][i].accessCount < cache[index]
[lfuIndex].accessCount) {
58             lfuIndex = i;
59         }
60     }
61     cache[index][lfuIndex].valid = true;
62     cache[index][lfuIndex].tag = tag;
63     cache[index][lfuIndex].accessCount = 1;
64     return false;
65 }
66 };

```

4.2.4.4 随机替换 (Random) 策略

不同于以上的替换策略，random策略要求我们生成一个随机数，随机选择被替换的cache line，因此不需要添加额外的监视机制，只需要在替换时生成随机数，根据随机数进行决策。

以下是C++的Random策略具体实现示例（以组相联为例）：

```

1  class CacheBlock {
2  public:
3      bool valid; // 有效位
4      int tag; // 标签位
5      vector<int> data; // 存储对应数据块
6      CacheBlock(int blockSize) : valid(false), tag(0), data(blockSize) {} //
构造函数，初始化有效位、标签位、数据块和最后访问时间
7  };
8
9  class SetAssociativeCacheSimulator {
10 private:
11     vector<vector<CacheBlock>> cache; // 创建一个二维向量模拟cache
12     int blockOffsetBits; // 块偏移位数
13     int indexBits; // 索引位数
14     int blockSize; // 内存块大小，用于后续操作
15     int sets; // 每个组中的块数
16
17 public:
18     // 构造函数，初始化cache大小和块大小
19     SetAssociativeCacheSimulator(int cacheSize, int blockSize, int sets) :
blockSize(blockSize), sets(sets) {
20         int groupCount = cacheSize / (blockSize * sets); // 计算组的数量
21         cache.resize(groupCount, vector<CacheBlock>(sets,
CacheBlock(blockSize))); // 初始化cache数组
22         blockOffsetBits = 0;
23         // 计算内存地址中块偏移位数
24         while (blockSize > 1) {
25             blockSize >>= 1;
26             blockOffsetBits++;
27         }
28         indexBits = 0;
29         // 计算内存地址中索引位数
30         while (groupCount > 1) {
31             groupCount >>= 1;

```

```

32         indexBits++;
33     }
34 }
35
36 // 检查一个地址是否在cache中
37 bool isHit(int address) {
38     // 计算索引和标签
39     int index = (address >> blockOffsetBits) & ((1 << indexBits) - 1);
40     int tag = address >> (blockOffsetBits + indexBits);
41
42     // 在给定索引的所有块中查找标签匹配的块
43     for (int i = 0; i < sets; i++) {
44         if (cache[index][i].valid && cache[index][i].tag == tag) {
45             //cout << hex << address << " hit" << endl;
46             return true;
47         }
48     }
49
50     // 如果未找到匹配的块，选择一个块来替换
51     // 使用LRU策略选择最长时间未被访问的块
52     //cout << hex << address << " miss" << endl;
53     random_device rd;
54     mt19937 gen(rd());
55     std::uniform_int_distribution<int> dis(0, sets - 1);
56     int randomIndex = dis(gen);
57     cache[index][randomIndex].valid = true;
58     cache[index][randomIndex].tag = tag;
59     return false;
60 }
61 };

```

5 数据处理与分析

自行编写Cache模拟器，以获得的内存地址访问序列文件为输入，统计miss/hit情况。调整相关参数（cache块大小，cache大小）和算法（映射方式，替换算法），查看miss/hit的变化情况，并做出相应解释。

5.1 模拟器封装

我们根据前文所提到的各种策略，将其封装为一个完整的Cache模拟器。主代码如下，分代码原理同上，会放在附录：

```
1  #! /usr/bin/env python3
2  import os
3
4  def init_visual():
5  # 打印初始化提示信息
6      print(r"""
7  /$$$$$          /$$          /$$   /$$ /$$
8
9  | $$_  $$          | $$          | $$   | $$_  /
10
11 | $$ \__ / $$$$$$ /$$$$$$| $$$$$$ /$$$$$ | $$   | $$ /$$
12 /$$$$$$ /$$   /$$ /$$$$$ | $$
13 | $$   |___ $$ /$$_ /$$_  $$ /$$_  $$ | $$   / $$/ | $$
14 /$$_ /$$_  | $$ | $$ |___ $$ | $$
15 | $$   /$$$$$$| $$   | $$ \ $$| $$$$$$$ \ $$ $$/ | $$|
16 $$$$$$ | $$   | $$ /$$$$$$| $$
17 | $$   $$ /$$_  $$/ | $$   | $$ | $$| $$_ /
18 \___  $$/ | $$ /$$_  $$/ | $$
19 | $$$$$$| $$$$$$| $$$$$$| $$ | $$| $$$$$$ \ $/ | $$
20 /$$$$$$| $$$$$$| $$$$$$| $$
21 \___ / \___ / \___ /|_ / |_ / \___ /
22 |_ /|_ / \___ / \___ /|_ /
23
24                                     SDU-2024-CWL-GZC-ZAY-BHR
25 """)
26 # 可视化缓存状态
27
28 def main():
29     init_visual()
30     # □ direct.py □ group+FIFO.py □ group+LFU.py □ group+LRU.py □
31     group+random.py
32     # 根据不同的命令行参数，调用不同的算法，可视化的话加-v 如果不是direct要提供num
33     import argparse
34     parser = argparse.ArgumentParser()
35     parser.add_argument("-v", "--visualize", help="Visualize the cache",
36 action="store_true")
37     # num不是必须
38     parser.add_argument("-n", "--num", help="The number of cache lines")
39     # 加提示有哪些算法，包括direct, FIFO, LFU, LRU, random
40     parser.add_argument("-a", "--algorithm", help="The algorithm to use",
41 choices=["direct", "FIFO", "LFU", "LRU", "random"])
42     # 加上块大小
43     parser.add_argument("-b", "--block-size", help="The size of a block")
```

```

32     # 加上cache大小
33     parser.add_argument("-c", "--cache-size", help="The size of the cache")
34     args = parser.parse_args()
35     if args.algorithm == "direct":
36         import direct
37         hit_r =
38     direct.main(args.visualize,int(args.block_size),int(args.cache_size))
39     elif args.algorithm == "FIFO":
40         import group_FIFO
41         hit_r =
42     group_FIFO.main(args.visualize,int(args.num),int(args.block_size),int(args.c
43     ache_size))
44     elif args.algorithm == "LFU":
45         import group_LFU
46         hit_r =
47     group_LFU.main(args.visualize,int(args.num),int(args.block_size),int(args.ca
48     che_size))
49     elif args.algorithm == "LRU":
50         import group_LRU
51         hit_r =
52     group_LRU.main(args.visualize,int(args.num),int(args.block_size),int(args.ca
53     che_size))
54     elif args.algorithm == "random":
55         import group_random
56         hit_r =
57     group_random.main(args.visualize,int(args.num),int(args.block_size),int(args
58     .cache_size))
59     else:
60         # 打印帮助信息
61         parser.print_help()
62         exit(1)
63         # 记录每次的命中率和算法、块大小、组数，写入cache.log文件
64         with open("cache.log", "a") as f:
65             f.write(f"Algorithm: {args.algorithm}, Cache Size:
66             {args.cache_size}, Block Size: {args.block_size}, Group Number: {args.num},
67             Hit Rate: {hit_r}\n")
68         print("[+] the result has been written to cache.log")
69
70
71
72 if __name__ == "__main__":
73     main()

```

目录如下:

```

1 β ~/Progress/csapp/3/gzc/ ls
2 □ __pycache__ □ cache.log □ cache.tar.gz □ cache_visual □ data.txt □
  direct.py □ group_FIFO.py □ group_FRU.py □ group_LFU.py □ group_random.py
  □ test

```

执行效果如下:

```

1 β ~/Progress/csapp/3/gzc/ ./cache_visual

```



```
25         echo "Algorithm: $algorithm"
26         echo "Cache Size: $cache_size, Block Size:
$block_size"
27         echo COMMAND: ./cache_visual -c $cache_size -b
$block_size -a $algorithm -n $block_size
28         ./cache_visual -c $cache_size -b $block_size -a
$algorithm -n $block_size
29         echo
30     done
31 done
32 echo
33 done
```

5.2 实验数据

以 Cache 大小 1024 Bytes，块大小分别为 4 和 8 为例，三种映射方式和四种随机替换策略的命中率数据如下（完整数据见附录）：

Cache 大小	映射方式	块大小	最近最少使用 (LRU)	先进先出 (FIFO)	最不常用 (LFU)	随机替换 (Random)
1024 Bytes	直接映射	4	0.9642034971660323			
		8	0.9737754101357977			
	4路组相连	4	0.9704469524458554	0.9684773033259645	0.9704469524458554	0.9690756777421339
		8	0.9840204943237538	0.9828050462909097	0.9840204943237538	0.9826305204195269
	全相连	4	0.9704469524458554	0.9684773033259645	0.9704469524458554	0.9691774845004405
		8	0.9840204943237538	0.9827053172215481	0.9840204943237538	0.9834034207070791

6 实验结论

6.1 什么是仿真器，仿真器有什么作用/价值？

- 在本次实验中，我们使用了 gem5 这一仿真器，它主要用于模拟真实的 CPU 运行和内存读写，通过读取仿真器模拟过程中追踪的数据，我们可以获取程序的内存地址访问序列。
- 跳出本次实验的范围，仿真器可以以更低廉的成本（无需购买硬件，经济成本低；无需烧写固件等，事件成本低）实现对程序性能、正确性等的初步检测。但也要注意，仿真器的仿真性是有限的，仿真结果是一种理想情况，其总与真实情况存在一定的差异。

6.2 正确性分析

为验证cache模拟器的正确性，我们采用micro-benchmarking的方法，在此以直接映射实现cache模拟器作为示例，分别展示C++实现以及Python实现（可视化实现）的正确性，其他映射方式也满足但因篇幅原因不做介绍。

我们编写了一段内存序列地址，具体如下：

```
1 0x742ec = 1110100001011101100
2 0x6ed8c = 1101110110110001100
3 0x6ed8c = 1101110110110001100
4 0x6ed9c = 1101110110110011100
5 0x6ed98 = 1101110110110011000
6 0x6ed98 = 1101110110110011000
7 0x6ed98 = 1101110110110011000
8 0x6ed98 = 1101110110110011000
9 0x6ed9c = 1101110110110011100
10 0x6edb0 = 1101110110110110000
11 0x6ed90 = 1101110110110010000
12 0x6ed98 = 1101110110110011000
13 0x6ed24 = 1101110110100100100
14 0x6edbc = 1101110110110111100
15 0x6ed98 = 1101110110110011000
```

我们将使用此数据进行正确性验证。

6.2.1 C++实现Cache模拟器正确性分析

以C++实现的直接映射cache模拟器为例，验证cache大小为1024，块大小为4时的正确性。

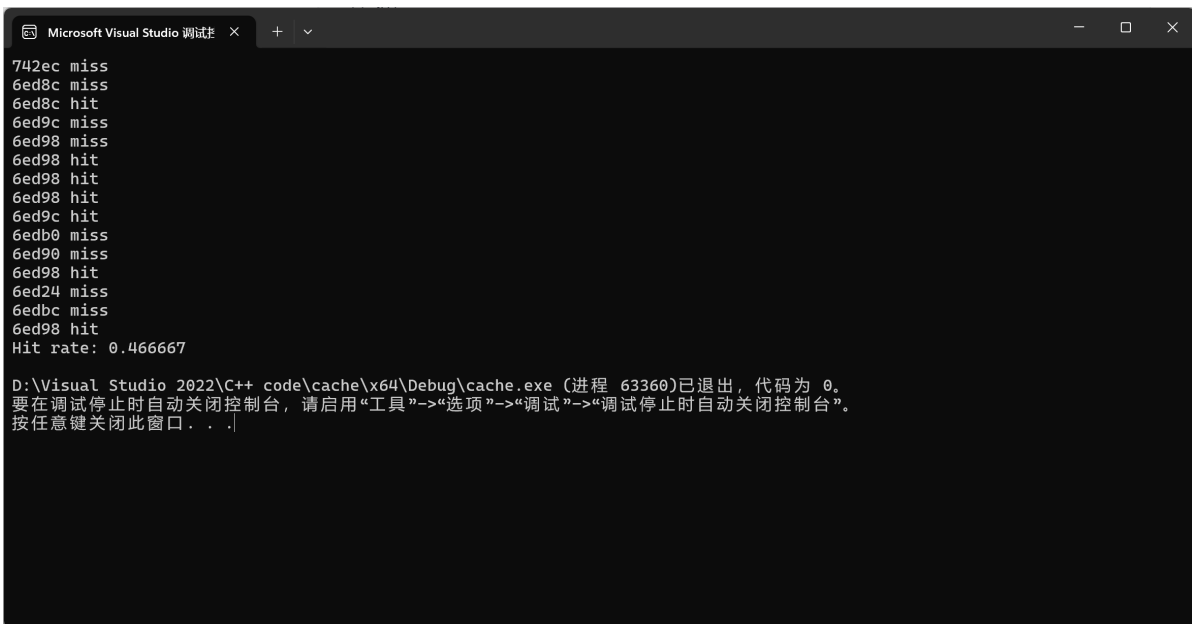
首先我们先推导出这段内存序列的miss/hit情况，因为块大小为4，所以地址最后2bits为内存块偏移量，cache大小为1024，块大小为4，则一共有256个cache line，对应index位大小应该是8位，其余位应该为tag位。

首先读取0x742ec，由于最初cache为空，故为miss，此时0x742ec存入index=10111011=187的cache line中，改变有效位，之后读取0x6ed8c，获得其index位对应数字为01100011，找到对应cache line为空，未命中miss，存入对应数据，改变有效位，之后再次访问0x6ed8c，获得对应index=01100011，找到对应cache line，有效位为有效，比较tag位相同，故此时hit，以此类推得到地址的对应命中情况应如下：

```
1 miss
2 miss
```

```
3 hit
4 miss
5 miss
6 hit
7 hit
8 hit
9 hit
10 miss
11 miss
12 hit
13 miss
14 miss
15 hit
```

运行代码可得：



```
Microsoft Visual Studio 调试
742ec miss
6ed8c miss
6ed8c hit
6ed9c miss
6ed98 miss
6ed98 hit
6ed98 hit
6ed98 hit
6ed98 hit
6ed9c hit
6edb0 miss
6ed90 miss
6ed98 hit
6ed24 miss
6edbc miss
6ed98 hit
Hit rate: 0.466667

D:\Visual Studio 2022\C++ code\cache\x64\Debug\cache.exe (进程 63360)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

发现与推导结果相同，故验证C++实现cache模拟器满足正确性。

组相联与全相联及不同替换策略实现同样满足正确性，因篇幅原因不再介绍。

6.2.2 可视化模拟器正确性分析（Python）

使用 5.1 模拟器封装中的可视化工具进行 python 版本的可视化正确性分析，首先读取 0x742EC，变化如下

```
1 Address: 0x742EC Miss
2 |Block 187: | Tag: 0x1D0 | Status: Valid |
```

之后依次执行内存读取，变化如下：

```
1 Address: 0x6ED8C Miss
2 |Block 99: | Tag: 0x1BB | Status: Valid |
3 Address: 0x6ED8C Hit
4 |Block 99: | Tag: 0x1BB | Status: Valid |
5 Address: 0x6ED9C Miss
6 |Block 103: | Tag: 0x1BB | Status: Valid |
7 Address: 0x6ED98 Miss
8 |Block 102: | Tag: 0x1BB | Status: Valid |
```

```
9 Address: 0x6ED98 Hit
10 |Block 102: | Tag: 0x1BB | Status: valid |
11 Address: 0x6ED98 Hit
12 |Block 102: | Tag: 0x1BB | Status: valid |
13 Address: 0x6ED98 Hit
14 |Block 102: | Tag: 0x1BB | Status: valid |
15 Address: 0x6ED9C Hit
16 |Block 103: | Tag: 0x1BB | Status: valid |
17 Address: 0x6EDB0 Miss
18 |Block 108: | Tag: 0x1BB | Status: valid |
19 Address: 0x6ED90 Miss
20 |Block 100: | Tag: 0x1BB | Status: valid |
21 Address: 0x6ED98 Hit
22 |Block 102: | Tag: 0x1BB | Status: valid |
23 Address: 0x6ED24 Miss
24 |Block 73: | Tag: 0x1BB | Status: valid |
25 Address: 0x6EDBC Miss
26 |Block 111: | Tag: 0x1BB | Status: valid |
27 Address: 0x6ED98 Hit
28 |Block 102: | Tag: 0x1BB | Status: valid |
29
30 Hit Rate = 0.4666666666666667
```

缓存变化情况与 C++ Cache 模拟器以及人工推理一致，即可判断 Python 实现的 Cache 模拟器是正确的。使用同样的方法可验证其他算法的正确性。

6.3 Cache 大小对命中率的影响

6.3.1 原理分析

当我们以四路组相联并采用LRU（最近最少使用）策略的Cache为例，来分析Cache大小对命中率的影响时，我们需要考虑以下几个因素。

1. Cache容量对命中率的影响：

1. Cache的容量越大，命中率通常越高。这是因为较大的Cache可以容纳更多数据块，从而减少了缺失率。
2. 然而，随着Cache容量的增加，命中率的提高速度会逐渐减缓。这是因为边际效应递减原理表明，增加Cache容量后，每增加一个单位的容量对命中率的提升效果会减弱。

2. 块大小对命中率的影响：

1. 在四路组相联的Cache中，块大小与组数成反比。较小的块大小会导致更多的组，从而提高了空间局部性，进而提高了命中率。
2. 然而，如果块大小过大，会减少Cache的总行数，同时离所访问的位置较远的块被再次使用的概率也变小。因此，存在一个“最佳块大小”，在这一点处Cache命中率达到最大值。

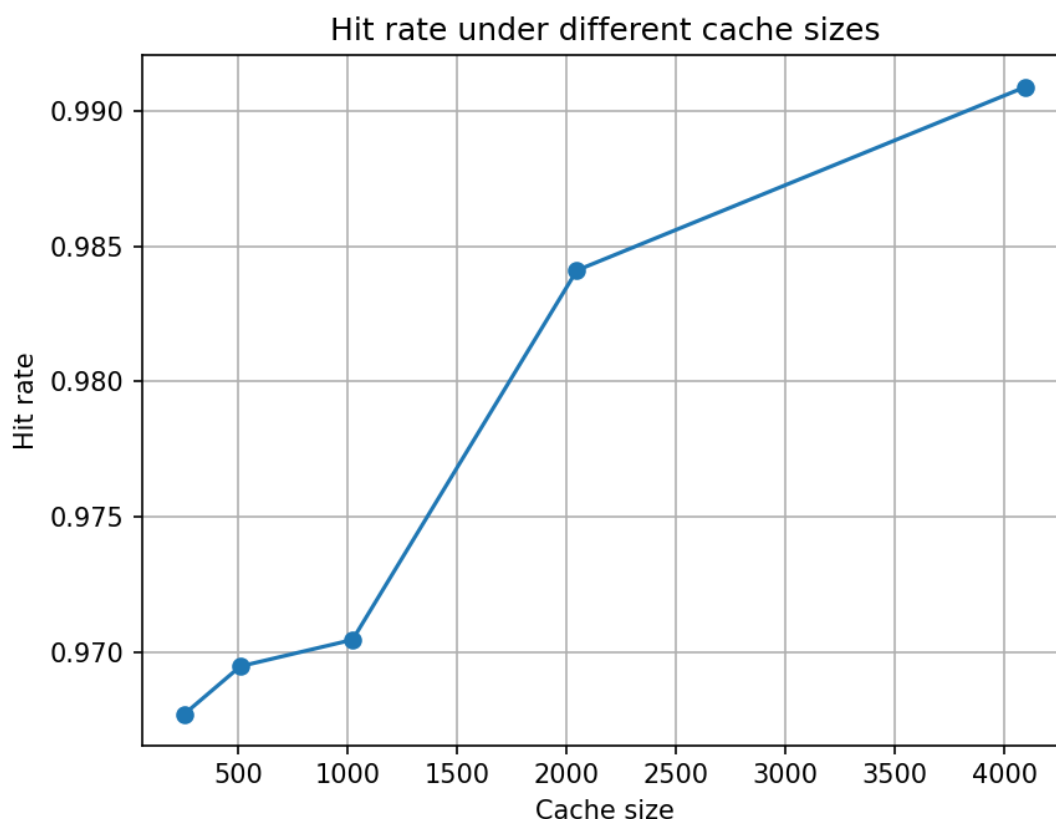
3. LRU替换策略对命中率的影响：

1. LRU替换策略选择最近最少使用的块进行替换。这种策略较好地反映了程序的局部性规律。
2. 当Cache容量较小时，随机替换策略可能相对较好；随着Cache容量的增加，LRU策略效果更好。

综上所述，Cache的大小、块大小以及替换策略都对命中率产生影响

6.3.2 结果分析

我们采取cache大小分别为256,512,1024,2048,4096进行测试，命中率结果如下：



不难发现，cache越大，命中率越高

6.4 不同组相连配置对命中率的影响

6.4.1 原理分析

当我们以组相联并采用LRU（最近最少使用）策略的Cache为例，来分析不同组相连路数对命中率的影响时，我们需要考虑以下几个因素。

1. 组相联映射方式：

1. 组相联映射是一种折中方案，结合了全相联映射和直接映射的特点。
2. 在组相联映射中，Cache的行被分成多个组，每组包含多行。每行内的数据块是全相联映射的，而不同组之间是直接映射的。
3. 规律是：主存块号 mod Cache总组数，决定了主存块映射到哪个组。

2. 影响因素：

1. 组相联映射的每组的行数通常取较小的值，如2、4、8、16。这里的“路数”表示每组的行数。
2. 组内有一定的灵活性，而且因为组内行数较少，比较的硬件电路相对简单。同时，空间利用率也比直接映射方式要高。

3. 命中率的影响：

1. 组相联映射兼具全相联和直接映射的优点。它在一定程度上提高了命中率。

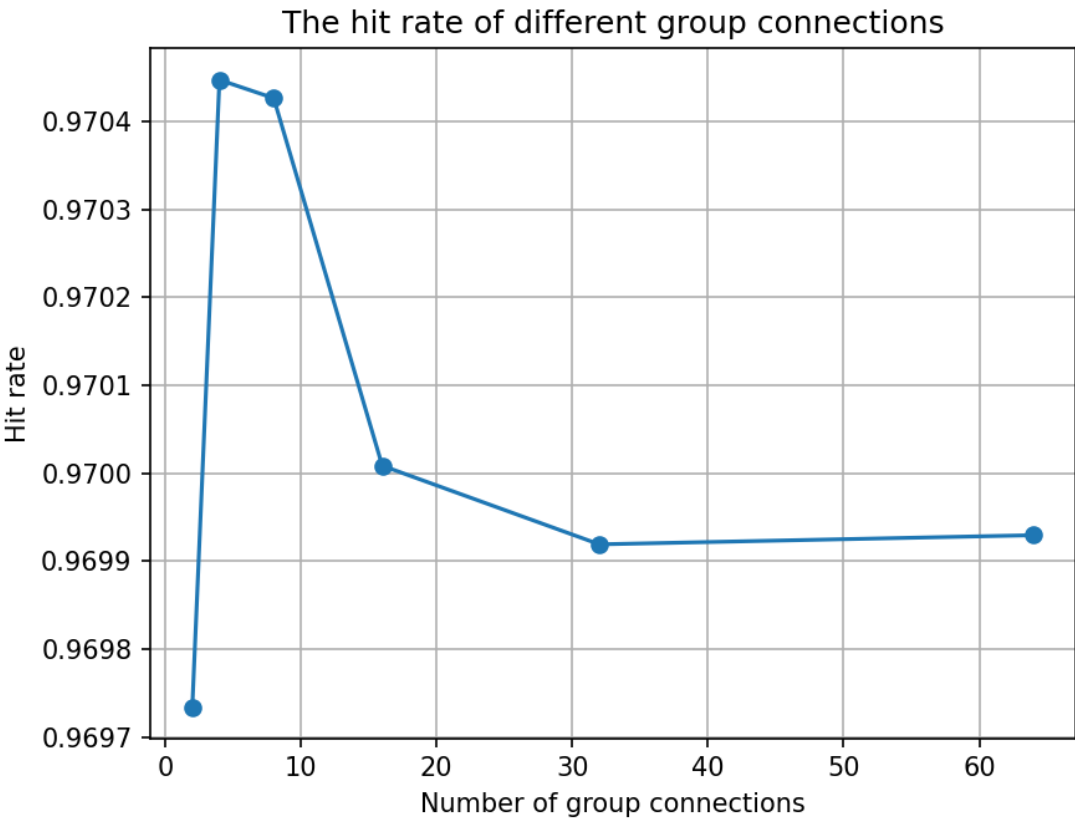
2. 当Cache的容量一定时，不同组相连路数对命中率的影响如下：

- 较小的路数（如2或4）：由于组内行数较少，容易发生冲突，频繁替换，效率可能降低。
- 较大的路数（如16）：虽然硬件复杂度较高，但命中率相对较高。

综上所述，组相联映射方式在一定程度上平衡了灵活性和命中率。选择适当的组相连路数取决于Cache的容量和性能需求

6.3.2 结果分析

我们采取组相连数分别为2,4,8,16,32进行测试，命中率结果如下：



不难发现，组相连数在为4和8时候命中率较高命中率，随着组数增长而变小。

6.5 对 LRU 算法的分析

Cache 大小	映射方式	块大小	最近最少使用 (LRU)	先进先出 (FIFO)	最不常用 (LFU)	随机替换 (Random)
1024 Bytes	直接映射	4	0.9642034971660323			
	4路组相连	4	0.9704469524458554	0.9684773033259645	0.9704469524458554	0.9690756777421339

以 Cache 大小 1024Bytes，四路组相连，块大小为 4 的实验数据来看，LRU 在本实验的场景下是一个很好的选择，它的命中率较高，同时它的代码实现较为容易。

从原理来看，LRU 基于局部性原理，即最近使用的数据很可能会在不久的将来再次被使用。因此，LRU 保留了最近使用的数据，以最大程度地提高缓存命中率。

但同时 LRU 算法仍然有一定缺陷：

1. 在链表，堆等按照访问顺序排序的数据结构的内存访问序列中，使用 LRU 算法会带来额外的性能开销。
2. 当缓存大小不足以容纳所有需要缓存的数据时，如果应用程序的访问模式不符合LRU假设，也会导致较大的开销。

LRU 的优势主要在于目前大部分真实应用场景都是符合其假设的，即最近使用的数据很可能在不久的将来再次被使用，仅在特定情况下会产生一定的开销。

6.6 不同的代码实现与优化方法对 cache 命中率的影响

综合本次实验与历次实验来看，不同的代码实现与优化方法会对内存访问序列造成极大影响，从而进一步影响 cache 命中率。

1. 从硬件角度，目前大部分消费级 CPU 使用了 LRU 作为缓存策略，L1 缓存至 L3 缓存大小不相同，映射方式也不同。要想获得更少的性能开销，就必须根据 CPU 的缓存特性对数据访问的局部性做针对性的优化。
2. 同时，不同的内存访问模式也会影响缓存命中率。顺序访问相邻的内存位置更能会利用缓存行的局部性，从而提高缓存命中率，而随机访问或跳跃式访问会导致缓存行的浪费，降低缓存命中率。
3. 对于 Memory Bound 的场景，应择适当的数据结构可以减少缓存行的失效，最大程度利用局部性，提高缓存命中率。
4. 对于多线程场景，应该采取合适的同步机制和访问合并策略，以减少对 L3 缓存的竞争和缓存失效，从而提高缓存命中率。

6.7 实验心得

在本次实验中，我们通过 gem5 仿真器获得了 ntt 程序的内存访问序列，针对这一访存序列实现了 Cache 模拟器。在这一过程中，我们了解到了仿真器与仿真在程序设计中，特别是异构程序设计中的意义。手动验证 Cache 正确性，实现各映射方式与替换算法，让我们对缓存有了更进一步的认识，也让我们对缓存的硬件设计有了初步的了解。

7 附录

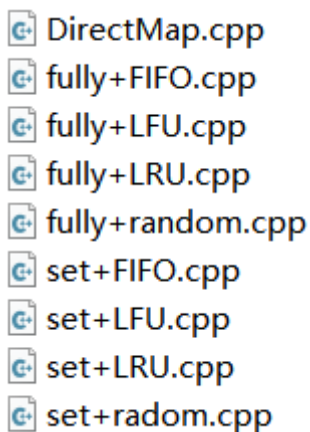
7.1 内存序列地址

memaccess.zip: 内存访问序列原始数据, 即三个输出文件 dram.out, mmu.out 与 MemoryAccess.out

MemoryAccess_p.out: 处理后的内存访问序列

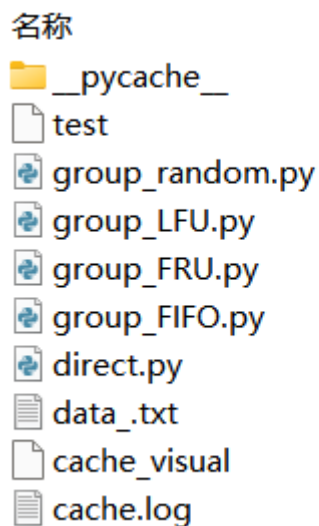
7.2 Cache模拟器C++实现

见 cache_cpp.zip:



7.3 Cache模拟器Python实现

见 cache.tar.gz:



7.4 完整实验数据

见cache.log