实验四：高速缓冲存储器 Cache 的建模

1. 实验目的

1) 熟悉并掌握 Cache 的基本结构及常见替换策略
2) 设计一个可灵活配置的单级 Cache 存储体系仿真器
3) 基于该仿真器和 SPEC 标准测试程序对 Cache 存储体系的性能进行分析

2. 实验环境

1) 系统要求：Linux
2) 编程工具：GCC
3) 编程语言：C/C++/JAVA

3. 实验内容

1) 设计一个通用的 Cache 仿真模型及单级 Cache 仿真器，可对主要的 Cache 体结构参数、替换以及写策略进行灵活配置。
2) 该仿真器使用具有标准格式的访存地址流文件作为输入（该地址流已由 SPEC 标准测试程序产生），并将最终 Cache 中的 tag 部分的存储内容和性能分析结果以标准格式输出到结果文件中。

4. 实验步骤

(1) Cache 的配置

该通用 Cache 仿真模型在仿真开始前，**可由用户在终端对各种体系结构参数、替换及写策略进行灵活配置**，所需进行配置的参数及要求如下：

Cache 体系结构参数

SIZE：Cache 的总容量（单位：Byte，取值任意）注：数据部分

ASSOC：Cache 相联度（ASSOC = 1 为直接相联 Cache，取值任意）

BLOCKSIZE：Cache 的块容量（单位：Byte，取 2 的幂次）

配置上述参数后，Cache 中的组数可由如下公式决定：

$$\text{Number of Sets} = \frac{\text{SIZE}}{\text{ASSOC}\times\text{BLOCKSIZE}}$$ （组数也取 2 的幂次）

(2) Cache 替换和写策略

本实验中将实现两种常见 Cache 替换策略：LRU（Least Recently Used）和 LFU

（Least Frequently Used）。替换策略也作为参数，在仿真器开始运行前进行配置。

a. LRU 替换策略

请见压缩包中的文件“**LRU.pptx**”。

b. LFU 替换策略

LFU 替换策略的主要思想是选择近期引用频率最低的数据块进行替换。每个块设置一个计数器用于跟踪该块的被引用的次数（每次读或写均为引用 1 次）。但该替换策略存在一个比较严重的问题：某一个块即使很长时间不被引用，但由于其具有较高的引用次数，而继续被保留在 Cache 中，从而浪费了有效的 Cache 存储空间。

因此，本实验将实现一个改进的 LFU 替换策略以解决上述问题，称为“动态生命期 **LFU（LFU with Dynamic Aging）**”策略。除了为每个块分配一个用于记录其引用次数的计数器（COUNT_BLOCK）之外，每一组设置一个 age（生命期）计数器（COUNT_SET）用于初始化块的引用次数，当一个块被调入 Cache 后。其工作机制如下所示：所有计数器初始化为“0”。

- 当一个块被调入某个 Cache 组后，其引用次数 COUNT_BLOCK 被初始化为 COUNT_SET + 1。
- 当一个块被引用时，其计数器 COUNT_BLOCK 自增“1”。
- 当发生替换的时候，选择该组中 COUNT_BLOCK 最小的块进行替换。当有多个块的 COUNT_BLOCK 相同时，按顺序选择第一个块进行输出。
- 当某组中的一个块被替换后，该组的 age 计数器 COUNT_SET 被设置为被替换块的引用次数。

有上述改进型 LFU 工作机制可以看出：

- COUNT_SET 的值等于或小于该组中所有 COUNT_BLOCK 的最小值。因此，COUNT_SET 可近似认为是该组中所有块的生命期“age”。
- 该策略解决了一个最近不常使用但具有较高引用次数块长期占用有效 Cache 空间的问题。即使一个块具有很高的引用次数，但如果它很长时间不被引用，该组中其它块的 COUNT_BLOCK 值将缓慢增加，以最终该长时间不被引用的块将成为引用次数最低的块。

参 考 文 献 ： Dilley et. al., "Enhancement and Validation of Squid CacheReplacement Policy", HP Laboratories, Palo Alto, CA, 1999.

(3) Cache 写策略

本实验将实现 Cache 的两种写策略，即，写回写分配 WBWA（write-back + write-allocate）和写直通写不分配 WTNA（write-through + write-not-allocate）。写策略也作为参数，在仿真器开始运行前进行配置。

a. Write-back Write-Allocate (WBWA)

如果写操作在 Cache 中命中，则更新 Cache 中的对应块，并将该块设置为“脏”。但是，该写操作并不同时更新下一级存储（下一级 Cache 或主存）。如果一个“脏”块被从 Cache 中替换，则该“脏”块将被“写回”下一级存储器。如果写操作在 Cache 中不命中，则 Cache 会分配一个数据块给该写操作。因此，在该策略中读缺失和写缺失都会在 Cache 中分配数据块。

b. Write-through Write-Not-Allocate (WTNA)

如果写命中，则 Cache 和下一级存储（下一级 Cache 或主存）中对应的块都被更新。该级 Cache 中的块不存在“脏”位。当发生替换的时候，被替换的块直接丢弃即可。如果写操作在 Cache 中不命中，则该级 Cache 不会给该写操作分配数据块。该写操作将直接被发送到下一级存储。

(4) Cache 的建模

本实验所设计的仿真器可对 Cache 存储体系中的任意一级 Cache 进行建模。对于实验的第一部分，仿真器先建模单级 Cache 存储体系。

单级 Cache 从 CPU 接受读/写请求。仅当发生读缺失或写缺失时，才会与下一级存储器（主存）发生交互。当发生读缺失的时候，Cache 总会为之分配一个数据块。但发生写缺失的时候，Cache 是否为之分配数据块取决于写策略。

我们仅说明需要分配数据块的工作流程。假设 Cache 为某个发生读/写缺失的数据块 X 分配空间。分配的过程由两步组成，需按如下顺序进行：

①. 为块 X 分配空间：如果在该组中有一个尚未分配的块（invalid），则直接将给块分配给块 X，掉转到步骤②。为了结果一致，总是将稍描到的第 1 个尚未分配的块分配给块 X。另一方面，如果该组中所有的块都已经分配，则需要根据替换策略挑选出某块 V 丢弃。对于 WBWA 策略，如果 V 是“脏”块，则需要将块 V 写回到主存储器。

②. 将块 X 存入 Cache：发射读 X 块请求到下一级存储器，将块 X 读出并

存入 Cache 组中相应的位置（步骤①决定的位置）。

(5) 仿真器的输入

仿真器读入的读/写地址流文件格式如下所示：

r|w <hex address>

r|w <hex address>

...

其中，“r”表示 CPU 发出读操作，“w”表示 CPU 发出写操作。仿真器需要解析地址流文件，并将其发送到 Cache 模型中。地址流文件保存在./trace 中，

(6) 原始数据统计

设计仿真器的目的是在仿真结束后收集各种原始统计数据，并对 Cache 的性能进行计算分析。在第一部分是实验中，整个 Cache 存储体系只有 1 级 L1 Cache，所需收集的原始统计数据如下：

a. L1 读次数

b. L1 读缺失次数

c. L1 写次数

d. L1 写缺失次数

e. L1 缺失率 = (b + d)/(a + c)

f. L1 的写回次数

g. 主存与 Cache 间的通讯量 = 从主存读出和写入主存的总块数

注：当使用 WBWA 策略时，g = (b + d + f)。

仿真器在仿真结束后以规定的格式打印存储体系的配置参数、原始统计数据以及 L1 Cache 中 tag 部分的内容。

(7) 性能分析

Cache 平均访问时间（Average Access Time, AAT）

Cache 平均访问时间 AAT 是指 Cache 服务一次读/写请求的平均时间。对于只有 L1 的存储体系，AAT 可通过如下公式计算。

$$\text{Total Accesstime} = (\text{Reads}_{L1} + \text{Writes}_{L1}) \times \text{HT}_{L1} + (\text{ReadMisses}_{L1} + \text{WriteMisses}_{L1}) \times \text{MissPenalty}_{L1}$$

$$\text{L1 Miss Rate } (\text{MR}_{L1}) = \frac{\text{ReadMisses}_{L1} + \text{WriteMisses}_{L1}}{\text{Reads\_L1} + \text{Writes\_L1}}$$

$$\textbf{Average Access Time} = \text{HT}_{L1} + (\text{MR}_{L1} \times \textbf{MissPenalty}_{L1})$$

其中，HT_{L1}（L1 命中时间）和 L1 缺失代价如下所示：

L1 命中时间 HT_{L1}（以 ns 为单位）= 0.25ns + 2.5ns × (L1_Cache Size / 512KB) + 0.025ns × (L1_BLOCKSIZE / 16B) + 0.025ns × L1_SET_ASSOCIATIVITY

L1 缺失代价 MissPenaltyL1（以 ns 为单位）= 20ns + 0.5 × (BLOCKSIZE / 16B/ns)

(8) 仿真结果的验证

利用所提供的验证文件与仿真器的输出文件进行自动化（非手工）比对，对所设计仿真器功能的正确性进行验证。所需验证的内容包括：

- Cache 的配置参数
- Cache 最终的存储内容
- 各种性能指标

a. 仿真器编译与运行的要求

学生需要将仿真器的源代码提交到智慧树平台，授课老师对仿真器进行编译和运行，并给出成绩。因此，所提交的源代码必须满足如下的严格要求，否则仿真器可能无法正常运行，导致评分时被扣除相应的分数。

- 仿真器的编译和运行环境必须是 Ubuntu，以保证授课教师能够正确编译和运行所提交的仿真器。
- 除了提交源代码文件，还必须提交 Makefile 文件以完成仿真器的自动编译。通过此 Makefile 文件编译生成的仿真器可执行文件名为“sim_cache”，同时 Makefile 文件还需要提供“make clean”命令，以便于删除目标（.o 文件）文件和可执行文件。由于学生缺乏 Linux 下的

编程经验，为了减轻设计难度，本实验工程中提供一个 Makefile 文件的示例，可通过适当修改以满足需求。

- 仿真器运行时，能够在终端命令行输入如下 6 个参数（顺序不能改变）：

sim_cache <L1_BLOCKSIZE> <L1_SIZE> <L1_ASSOC> <L1_REPLACEMENT_POLICY> <L1_WRITE_POLICY> <trace_file>

命令行参数	解释
<L1_BLOCKSIZE>	块大小（单位：B），2 的幂次，正数
<L1_SIZE>	Cache 的容量（单位：B），正数
<L1_ASSOC>	Cache 的相联度，至少为“1”
<L1_REPLACEMENT_POLICY>	替换策略，0：LRU，1：LFU
<L1_WRITE_POLICY>	写策略，0：WBWA，1：WTNA
<trace_file>	地址流文件名

例：8KB 4 路组相联 Cache，块大小为 32B，采用 LRU 替换策略和 WTNA 写策略，仿真器输入的地址流文件名为“gcc_trace”，命令行参数为：

$ ./sim_cache 32 8192 4 0 1 gcc_trace

仿真器的输出必须打印在终端之上，以便授课老师重定向到文件中与验证文件进行结果的比较。

b. 验证

仿真器的输入结果必须在内容和格式上都与验证文件一致。授课教师将通过“diff”命令评判仿真器功能的正确性。因此，在提交仿真器之前，学生需要通过如下两个步骤来对仿真器的功能进行验证。

- 首先，将仿真器的输出结果重定向到一个临时文件，可通过“>”操作符实现。例如：

$ ./sim_cache 32 8192 4 0 1 gcc_trace>my_output

- 然后，通过运行“diff”命令，将保存输出结果的临时文件与验证文件进行比较，若输出“nothing”则表示输出结果正确。例如：

$ diff –iw my_output validation_run

其中，-iw 表示 diff 命令将忽略大小写和空格。

5. 实验提交

学生需要将设计完成的仿真器提交到智慧树平台之上，以 ZIP 压缩包的形式提交，文件名为（第 X 组_ project4.zip）。该压缩包中应该包含如下文件：

1) 源代码
2) Makefile 文件

在 Linux 下创建 ZIP 压缩包的命令如下，假设源文件为.c 和.h 文件：

$ zip 第 X 组_ project4 *.c *.h Makefile

6. 评分标准

评分要求：

- 所有性能指标正确
- Cache 最终的存储内容正确

实验满分 100 分。其中，实验工程 70 分，实验报告 30 分，实验工程分值如下：

仿真器能够编译通过并运行		20
LRU+WBWA	Validation Run #1	10
LRU+WTNA	Validation Run #2	10
	Validation Run #3	10
LFU+WBWA	Validation Run #4	10
LFU+WTNA	Validation Run #5	10
Total		70

注：通过实验包中的 checklist.pdf 文件完成自查，确保符合所有评分要求。

TIPS: 可能会用到的一些 C 函数

strtok, strtol, strcmp, atoi, 位运算（&, |, ~），移位（>>, <<）等

附 录

——设计建议

该附录对 Cache 仿真器设计给出一些建议。综合考虑仿真效率和仿真功能，在 Cache 仿真器只需要仿真每个块的 tag 部分，并不需要仿真任何的数据移动。学生可以使用 C、C++或 JAVA 语言进行仿真器设计。下面的叙述中假设使用 C/C++。

该附录只供参考之用。学生在进行 Cache 仿真器设计时不一定要遵循该建议，只要仿真输出结果正确即可。

设计仿真器程序时，如果使用 C 语言，Cache 可以表示为 struct 结构体，若使用 C++，则 Cache 可表示为 class 类。建议使用 C++语言，因为第一部分设计的 Cache 为一个通用模型，若使用 C++进行描述，在后续设计中可以方便的继承该类，并实例化出各级 Cache。然后，每个 Cache 可以看成是链表中的一个节点。每个 Cache 实例可以通过其内部定义的一个指针去访问下一级存储。对于直接与主存交互的 Cache，该指针为 NULL。这样，仿真器只需要将读/写请求发送给 L1 Cache（与 CPU 交互）。L1 Cache 在需要的时候将该请求通过指针 nextlevel 发送到下一级存储器。Cache 可通过递增的计数器变量记录读次数、写次数、写回次数等指标。Cache 的类示例如下所示：

```
class Cache {
    private:
        /* Cache 类的数据成员：
                Cache 的配置参数，如 cache 容量、相联读、块大小等；
                tag 部分；
                各种计数器变量；
        */
        // 指向下一级 Cache 的指针
        CACHE *nextLevel;

    public:
        // Cache 类的成员函数
        bool readFromAddress(unsigned int add);
        bool writeToAddress(unsigned int add);
        // Cache 其他功能函数
}
```

在仿真过程中，首先，利用通过命令行传递的配置参数初始化 Cache 实例。

然后，解析地址流 trace 文件，将读/写请求发送到 L1 Cache，L1 Cache 相应的计数器开始计数，更新 tag 部分，如果需要，还需向下一级 L2 Cache 发送读/写请求。同样，L2 Cache 也执行递增其相关计数器变量，更新 tag 等操作。仿真结束时，即 trace 文件中所有的访存请求都已经被处理完，则仿真器读取计算机变量，并显示出每级 Cache 的性能指标。