



华中科技大学

计算机系统结构实验报告

姓 名： 刘本嵩
学 院： 计算机科学与技术
专 业： 计算机科学与技术
班 级： CS1601
学 号： U201614531
指导教师： 童薇、施展

分数	
教师签名	

2020 年 4 月 28 日

目 录

1. Cache 模拟器实验.....	3
1.1. 实验目的.....	3
1.2. 实验环境.....	3
1.3. 实验思路.....	3
1.4. 实验结果和分析.....	7
2. 总结和体会.....	7
3. 对实验课程的建议.....	8

1. Cache 模拟器实验

在源代码注释中注意到，我们需要完成一个为 valgrind 的 lackey 工具的内存访问情况输出结果，来计算缓存命中情况的程序。

1.1. 实验目的

1. 理解 CPU cache 工作原理；
2. 实现一个高效的 cache 模拟器。

1.2. 实验环境

操作系统环境：

```
recolic@RECOLICMPC
OS: Manjaro 19.0.2 Kyria
Kernel: x86_64 Linux 4.19.107-1-MANJARO
Uptime: 317d7h55m
Packages: 1823
Shell: fish 3.1.0
Resolution: 1920x1080
DE: GNOME 3.34.4
WM: Mutter
Disk: 93G / 112G (88%)
CPU: Intel Core i3-7Y30 @ 4x 2.6GHz [45.0°C]
GPU: Intel Corporation HD Graphics 615 (rev 02)
RAM: 2149MiB / 3827MiB
```

编译环境：

gcc 9.2.1-archlinux, GNU Make 4.3

1.3. 实验思路

显然，我们只需要依次读入文件的每一行，忽略其中的 I 指令，将 M 指令解

释为一个 L 加一个 S 指令。输入的文件就变成了一连串的 Load 和 Store 指令。

然后，当发生一次内存访问时，我们检查其对应的 cache line 是否命中，如果命中，就更新 timestamp 并记录结果。如果不命中，就用 LRU 策略，将最老的一个 cache line 清除出去(如果必要)，并相应的记录结果。我们可以注意到，在这个过程中，Load 操作和 Store 操作并没有什么区别。因此在这一部分代码甚至无需区分它们。

在具体实现中，我抛弃了原有 csim.c，修改了 Makefile，使用 C++17 完成实验。这使得老师需要一个比较新(至少 2017 年后)的 C++ 编译器来编译我的代码。同时，我有一个完全自己实现的 C++ 库 rlib，它提供了一些很便捷的函数，因此我也将它包含在了我提交的代码包中，老师直接使用 make 命令即可进行编译。

具体实现细节如下：

```
#include <rlib/stdio.hpp>
#include <rlib/opt.hpp>
#include <fstream>
using namespace rlib::literals;

using addr_t = uint64_t;
constexpr addr_t INVALID_ADDR = (addr_t)-1;
struct cache_line_t {
    addr_t addr = INVALID_ADDR; uint64_t time;
}; // cache line contains ANY ONE addr, which lies in this cacheline.
using cache_set_t = std::vector<cache_line_t>; // E cache lines in every
// cache set.
using cache_t = std::vector<cache_set_t>; // 2^s cache sets.

cache_t cache;
uint64_t wall_time = 0;
auto hit_counter = 0, miss_counter = 0, evict_counter = 0;

int main(int argc, char **argv) {
```

```

rlib::opt_parser args(argc, argv);
// Prepare parameters.
///////////
if(args.getBoolArg("-h")) {
    rlib::println("Sorry, no help message provided.");
    return 1;
}
auto fname = args.getValueArg("-t");
auto s = args.getValueArg("-s").as<size_t>();
auto E = args.getValueArg("-E").as<size_t>();
auto b = args.getValueArg("-b").as<size_t>();
auto verbose = args.getBoolArg("-v");

cache.resize(1 << s); // 2^s cache sets.
for(auto &cache_set : cache) cache_set.resize(E); // E cache lines
in every cache set.

// Actual implementation.
///////////
auto is_addr_match = [&](addr_t l, addr_t r) {
    return (l xor r) >> b == 0;
};
auto process_query = [&](addr_t addr) {
    // Return 'H' for Hit, 'M' for miss, 'E' for miss+eviction.
    ++wall_time;
    auto &my_cache_set = cache[(addr >> b) % cache.size()];
    auto *cache_line_to_evict = &my_cache_set[0];
    for(auto &cache_line : my_cache_set) {
        if(is_addr_match(cache_line.addr, addr)) {
            cache_line.time = wall_time;
            return 'H'; // Cache hit.
        }
        if(cache_line.time < cache_line_to_evict->time)
            cache_line_to_evict = &cache_line;
    }

    // Cache miss.
    auto returnVal = cache_line_to_evict->addr == INVALID_ADDR ? 'M' :
'E';
    cache_line_to_evict->addr = addr;
    cache_line_to_evict->time = wall_time;
    return returnVal;
};
auto update_counters = [&](char res) {

```

```

if(verbose)
    rlib::println(res);
switch(res) {
case 'H':
    ++hit_counter;
    return;
case 'E':
    ++evict_counter;
case 'M':
    ++miss_counter;
    return;
}
};

auto parse_line = [] (const auto &line) -> std::pair<char, addr_t> {
    auto op = line[1];
    if(op == ' ') return std::make_pair('I', 0); // 'I' operation.
    auto addr_and_size = line.substr(3);
    auto pos = addr_and_size.find(',');
    if(pos == std::string::npos)
        throw std::invalid_argument(line);
    return std::make_pair(op, std::stoull(addr_and_size.substr(0, pos),
0, 16));
};

// Main loop here. /////////////////////////////////
auto finput = std::ifstream(fname);
while(true) {
    auto line = rlib::scanln(finput);
    if(finput.eof()) break;
    auto [op, addr] = parse_line(line);
    switch(op) {
    case 'I':
        break;
    case 'M':
        update_counters(process_query(addr));
        [[fallthrough]];
    case 'L':
    case 'S':
        update_counters(process_query(addr));
        break;
    default:
        throw std::invalid_argument("Unknown op.");
    }
}

```

```

// Results.
rlib::println("hits:{} misses:{} evictions:{}"_format(hit_counter,
miss_counter, evict_counter));
std::ofstream(".csim_results", std::ios::trunc) << "{} {}"
{}\_n"_format(hit_counter, miss_counter, evict_counter);
}

```

1.4. 实验结果和分析

实验结果显示，我的实现完全正确。所有命令行选项均能正常工作。实验结果截图如下：

```

→ cachelab-handout git:(master) ✘ ./test-csim
          Your simulator      Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
  3 (1,1,1)        9     8       6        9     8       6  traces/yi2.trace
  3 (4,2,4)        4     5       2        4     5       2  traces/yi.trace
  3 (2,1,4)        2     3       1        2     3       1  traces/dave.trace
  3 (2,1,3)      167    71      67      167    71      67  traces/trans.trace
  3 (2,2,3)      201    37      29      201    37      29  traces/trans.trace
  3 (2,4,3)      212    26      10      212    26      10  traces/trans.trace
  3 (5,1,5)      231     7       0      231     7       0  traces/trans.trace
  6 (5,1,5)  265189  21775   21743  265189  21775   21743  traces/long.trace
27
TEST_CSIM_RESULTS=27
→ cachelab-handout git:(master) ✘

```

2. 总结和体会

通过这次实验,我对于 cache 的结构,运作方式以及如何更高效的写出 cache friendly 的代码有了更为深入的了解。通过对于 cache 的模拟,我对于 cache 的组成方式,与内存地址的对应关系以及 cache 组相连的工作方式有了更为深入的认识。

在对于错误代码的调试过程中,我尝试使用了各种方法来结构性的显示 cache 的内容以及替换的过程,这也给了我有关 cache 工作方式更为直观的印象。

3. 对实验课程的建议

建议在提交作业时，允许学生自己定义 Makefile. 自动评分脚本按如下方法工作：

1. 学生提交完整的作业文件
2. 脚本拿到一个学生的压缩包，解压，cd 进去
3. 运行 make，这里可以有一个预先定义好的 target/goal 名字.
4. 从老师的测试机拷贝一个好的测试文件 test-csim 和 csim-ref 进去，使用老师的测试机中的测试样例完成测试.
5. 将关键代码 csim.c 拷贝出来，人工进行检查，防止作弊.

这样，学生可以自己修改 Makefile 中的 gcc 编译选项，可以自己选用 c++ 等测试机器上已安装的其他语言，更加方便.