

**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](#_Toc520262182)

[1.1. 实验目的 3](#_Toc952435694)

[1.2. 实验环境 3](#_Toc600717453)

[1.3. 实验思路 3](#_Toc1108702144)

[1.4. 实验结果和分析 7](#_Toc1778088119)

[2. 总结和体会 7](#_Toc1359776868)

[3. 对实验课程的建议 8](#_Toc1473327863)

# 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.

## 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。

## 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 m3-7Y30 @ 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

## 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.

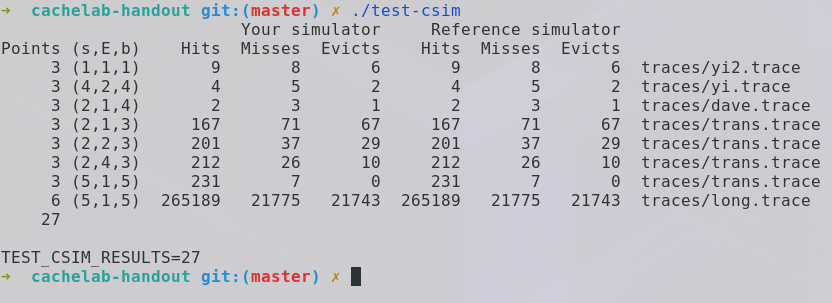
rlib::println("hits:{} misses:{} evictions:{}"\_format(hit\_counter, miss\_counter, evict\_counter));

std::ofstream(".csim\_results", std::ios::trunc) << "{} {} {}\n"\_format(hit\_counter, miss\_counter, evict\_counter);

}

## 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:



# 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。

# 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.