哈爾濱工業大學

实验报告

实 验（六）

题　　目 Cachelab

高速缓冲器模拟

专　　业 计算机系

学　　号 1190201421

班　　级 1936603

学　　生 张瑞

指 导 教 师 刘宏伟

实 验 地 点 G709

实 验 日 期 2021 年 5 月 27 日

计算机科学与技术学院

目 录

第 1 章 实验基本信息 ..... - 3 -

1.1 实验目的 ..... - 3 -
1.2 实验环境与工具 ..... - 3 -
1.2.1 硬件环境 ..... - 3 -
1.2.2 软件环境 ..... - 3 -
1.2.3 开发工具 ..... - 3 -
1.3 实验预习 ..... - 3 -

第 2 章 实验预习 ..... - 5 -

2.1 画出存储器层级结构，标识容量价格速度等指标变化 ..... - 5 -
2.2 计算机 CACHE 的参数查看与分析 ..... - 5 -
2.3 写出各类 CACHE 的读策略与写策略 ..... - 6 -
2.4 写出用 GPROF 进行性能分析的方法 ..... - 6 -
2.5 写出用 VALGRIND 进行性能分析的方法 ..... - 7 -

第 3 章 CACHE 模拟与测试 ..... - 10 -

3.1 CACHE 模拟器设计 ..... - 10 -
3.2 矩阵转置设计 ..... - 12 -

第 4 章 总结 ..... - 15 -

4.1 请总结本次实验的收获 ..... - 15 -
4.2 请给出对本次实验内容的建议 ..... - 15 -

参考文献 ..... - 16 -

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统存储器层级结构

掌握 Cache 的功能结构与访问控制策略

培养 Linux 下的性能测试方法与技巧

深入理解 Cache 组成结构对 C 程序性能的影响

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识

画出存储器的层级结构，标识其容量价格速度等指标变化

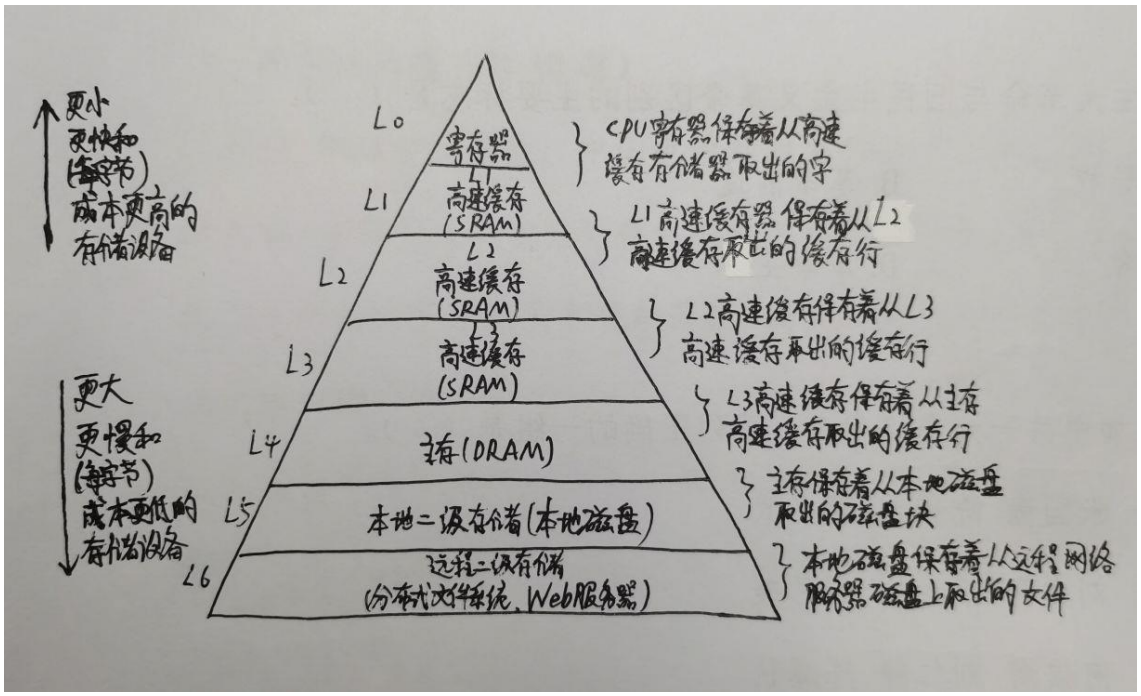
用 CPUZ 等查看你的计算机 Cache 各参数，写出 Cache 的基本结构与参数：缓存大小 C、分组数量 S、关联度/组内行数 E、块大小 B，及对应的编码位数：组索引位数 s、e、块内偏移位数 b

写出 Cache 的各种读策略与写策略

掌握 Valgrind、gprof 的使用方法

第 2 章 实验预习

2.1 画出存储器层级结构，标识容量价格速度等指标变化



2.2 计算机 Cache 的参数查看与分析

用 CPUZ 等查看你的计算机 Cache 的参数，写出各级 Cache 的 C(大小)、 S(组数)、 E(路数)、 B(块大小)，并分析相应的 s（组编码位数）、b（块内偏移地址位数）数值。

Cache	C	S	E	B	s	b
L1 D-Cache：	32KBytes	64	8	64Bytes	6	6
L1 I-Cache：	32KBytes	64	8	64Bytes	6	6
L2 Cache：	256KBytes	1024	4	64Bytes	10	6
L3 Cache：	3MBytes	4096	12	64Bytes	12	6

2.3 写出各类 Cache 的读策略与写策略

关于读的策略：

首先在高速缓存中寻找所需字 w 的副本，如果命中，立即返回字 w。如果不命中，从存储器层次结构中较低层中取出包含字 w 的块，将这个块存储到高速缓存行中（可能会采用一定的替换策略，如 LRU 策略，驱逐一个有效的行），然后返回字 w。

关于写的策略：

假设 CPU 写一个已经缓存了的字 w，如果写命中，在高速缓存更新了它的 w 的副本之后，更新 w 在层次结构中紧接着低一层中的副本时，有两种方法。第一种叫做直写，是立即将 w 的高速缓存块写回紧接着的低一层中的，虽然简单但是每次写都会引起总线流量。另一种方法叫做写回，尽可能地推迟更新，只有当替换算法要驱逐已经更新的块时，才将其写到紧接着的低一层中，这种方法显著减少总线流量，但增加了复杂性，必须为每个高速缓存行维护一个额外的修改位。

在处理写不命中问题上，也有两种策略。第一种称为写分配，加载相应的低一层中的块到高速缓存中，然后更新这个高速缓存块，缺点是每次不命中都会导致一个块从低一层传送到高速缓存。另一种方法称为非写分配，避开高速缓存，直接把这个字写到低一层中。

2.4 写出用 gprof 进行性能分析的方法

gprof 是 GNU profile 工具，可以运行于 linux、AIX、Sun 等操作系统进行 C、C++、Pascal、Fortran 程序的性能分析，用于程序的性能优化以及程序瓶颈问题的查找和解决。通过分析应用程序运行时产生的“flat profile”，可以得到每个函数的

调用次数，每个函数消耗的处理器时间，也可以得到函数的“调用关系图”，包括函数调用的层次关系，每个函数调用花费了多少时间。使用步骤如下：

（1）用 gcc、g++、xlC 编译程序时，使用-pg 参数，如：g++ -pg -o test.exe test.cpp 编译器会自动在目标代码中插入用于性能测试的代码片断，这些代码在程序运行时采集并记录函数的调用关系和调用次数，并记录函数自身执行时间和被调用函数的执行时间。

（2）执行编译后的可执行程序，如：./test.exe。该步骤运行程序的时间会稍慢于正常编译的可执行程序的运行时间。程序运行结束后，会在程序所在路径下生成一个缺省文件名为 gmon.out 的文件，这个文件就是记录程序运行的性能、调用关系、调用次数等信息的数据文件。

（3）使用 gprof 命令来分析记录程序运行信息的 gmon.out 文件，如：gprof test.exe gmon.out 则可以在显示器上看到函数调用相关的统计、分析信息。上述信息也可以采用 gprof test.exe gmon.out> gprofresult.txt 重定向到文本文件以便于后续分析。

2.5 写出用 Valgrind 进行性能分析的方法

Valgrind 是运行在 Linux 上一套基于仿真技术的程序调试和分析工具，它包含一个内核——一个软件合成的 CPU，和一系列的小工具，每个工具都可以完成一项任务——调试，分析，或测试等。

Valgrind 的参数

用法：valgrind [options] prog-and-args [options]: 常用选项，适用于所有 Valgrind 工具

--tool=

最常用的选项。运行 valgrind 中名为 toolname 的工具。默认 memcheck。

-h --help

显示所有选项的帮助，包括内核和选定的工具两者。

--version

显示 valgrind 内核的版本，每个工具都有各自的版本。

-q --quiet

安静地运行，只打印错误信息。

--verbose

更详细的信息。

--trace-children=

跟踪子线程? [default: no]

--track-fds=

跟踪打开的文件描述？[default: no]

--time-stamp=

增加时间戳到 LOG 信息? [default: no]

--log-fd=

输出 LOG 到描述符文件 [2=stderr]

--log-file=

将输出的信息写入到 filename.PID 的文件里，PID 是运行程序的进行 ID

--log-file-exactly=

输出 LOG 信息到 file LOG 信息输出

--xml=yes 将信息以 xml 格式输出，只有 memcheck 可用

--num-callers=

show callers in stack traces [12]

--error-exitcode=

如果发现错误则返回错误代码 [0=disable]

--db-attach= 当出现错误，valgrind 会自动启动调试器 gdb。[default: no]

--db-command= 启动调试器的命令行选项[gdb -nw %f %p] 适用于 Memcheck 工具的相关选项：

--leak-check= 要 求对 leak 给出详细信息? Leak 是指，存在一块没有被引用的内存空间，或 没有被释放的内存空间，如 summary，只反馈一些总结信息，告诉你有多少个 malloc，多少个 free 等；如果是 full 将输出所有的 leaks，也就是定位到某一个 malloc/free。 [default: summary]

--show-reachable= 如果为 no，只输出没有引用的内存 leaks，或指向 malloc 返回的内存块中部某 处的 leaks [default: no]

更详细的参数指令见附录 A。

Valgrind 的使用

首先，在编译程序的时候打开调试模式（gcc 编译器的-g 选项）。如果没有调试信息，即使最好的 valgrind 工具也将中能够猜测特定的代码是属于哪一个 函数。打开调试选项进行编译后再用 valgrind 检查，valgrind 将会给你的个详细的报告，比如哪一行代码出现了内存泄漏。

当检查的是 C++程序的时候，还应该考虑另一个选项 -fno-inline。它使得函数调用链很清晰，这样可以减少你在浏览大型 C++程序时的混乱。比如在使用这个选项的时候，用 memcheck 检查 openoffice 就很容易。当然，你可能不会做这项 工作，但是使用这一选项使得 valgrind 生成更精确的错误报告和减少混乱。

一些编译优化选项(比如-O2 或者更高的优化选项)，可能会使得 memcheck 提 交错误的未初始化报告，因此，为了使得 valgrind 的报告更精确，在编译的时候最 好不要使用优化选项。

如果程序是通过脚本启动的，可以修改脚本里启动程序的代码，或者使用 --trace-children=yes 选项来运行脚本。

下面是用 memcheck 检查 sample.c 的例子

这里用到的示例程序文件名为：sample.c（如下所示）,选用的编译器为 gcc。

生成可执行程序

gcc －g sample.c －o sample

运行 Valgrind valgrind --tool=memcheck ./sample

以下是运行上述命令后的输出

左边显示类似行号的数字（10297）表示的是 Process ID。

最上面的红色方框表示的是 valgrind 的版本信息。

中间的红色方框表示 valgrind 通过运行被测试程序，发现的内存问题。通过阅读这些信息，可以发现：

l 这是一个对内存的非法写操作，非法写操作的内存是 4 bytes。

l 发生错误时的函数堆栈，以及具体的源代码行号。

l 非法写操作的具体地址空间。 最下面的红色方框是对发现的内存问题和内存泄漏问题的总结。内存泄漏的大小

（40 bytes）也能够被检测出来。

第 3 章 Cache 模拟与测试

3.1 Cache 模拟器设计

提交 csim.c（详见作业压缩包）

程序设计思想：

本实验要求设计一个 cache 模拟器，在输入参数 s、E、b 为任意值时都能正确工作。阅读给出的模板之后，发现程序已经搭建好了基本的框架，完成了部分函数，只需要做四件事——补充 void initCache()，void freeCache()，void accessData(mem_addr_t addr)的代码并计算 S、E、B 的值。

对于 void initCache()函数，先检查参数 s 和 E 是否合法，然后用 cache = malloc(S * sizeof(cache_set_t))申请整个 cache 的空间，再用 cache[i] = malloc(E * sizeof(cache_line_t)申请每个 cache 组的空间，最后将申请得到的每个 cache 行初始化，将 valid、lru 和 tag 全部设为 0。

对于 void freeCache()函数，先释放每个 cache 组的空间，再释放整个 cache 的空间。

对于 void accessData(mem_addr_t addr)函数，先处理传入的参数 addr 得到组索引和标记。后续操作需分类讨论：

（1）若命中（标记位匹配且有效位被设置），将组内除命中行以外，所有有效位被设置的行的 lru 加 1，hit_count 也加 1，返回；

（2）若不命中，miss_count 加 1，接下来需考虑行替换：

①先查找是否有空行，若有，则将该行 valid 标为 1，同时将标记和 lru 进行设置，最后将组内除填入的行以外，所有有效位被设置的行的 lru 加 1，返回；

②若没有空行，eviction_count 加 1，用 lru 策略对最后一次访问时间最久远的一行进行替换——遍历该 cache 组，找出 lru 值最大的一行进行驱逐，最

后将组内除填入的行以外所有行的 lru 加 1，返回。

对于计算 S、E、B，按定义即可：S=1<<s；E=E；B=1<<b。

测试用例 1 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
hits:9 misses:8 evictions:6
```

测试用例 2 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:2
```

测试用例 3 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
hits:2 misses:3 evictions:1
```

测试用例 4 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
hits:167 misses:71 evictions:67
```

测试用例 5 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
hits:201 misses:37 evictions:29
```

测试用例 6 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
hits:212 misses:26 evictions:10
```

测试用例 7 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
hits:231 misses:7 evictions:0
```

测试用例 8 的输出截图：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
hits:265189 misses:21775 evictions:21743
```

用 test-csim 验证上述结果均正确：

```
zr@ubuntu:~/shared/cache/cachelab-handout$ ./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
```

3.2 矩阵转置设计

提交 trans.c（详见作业压缩包）

程序设计思想：

本实验要求在参考 cache 模拟器 csim-ref 上运行时，对不同大小的矩阵进行转置操作，缓存缺失的数量分别能减少至一定范围内。查看示例，发现矩阵有三种规格：32*32、64*64 和 61*67，且 cache 的参数为-s 5 -E 1 -b 5，即对这个 cache，有 32 组，每组 1 行，每行 32 个字节（能放下 8 个 int 型数据），整个 cache 能放下 32*8=256 个 int 型数据，可见待转置矩阵的大小大于 cache 大小。要想使缓存缺失的数量尽量小，就要减少 A、B 两个矩阵访问元素时产生的冲突，尽量将矩阵划分成一个个的小块，每次将各个小块存入 cache 进行转置操作。

对于 32*32 的矩阵，因为 cache 的一行能放下 8 个 int 数据，考虑将矩阵分为 8*8 的小块分别转置，这样一来每次取数据的时候，都能将 cache 里一行的数据全部用上，当其被覆盖时，不会有再次调入而导致的缓存缺失数增加，且每次调用 A 时，仅在矩阵对角线上与 B 冲突，缓存缺失数较少。

对于 64*64 的矩阵，上述划分已经不足以达到性能要求，需进一步改进。分析发现，若矩阵变为 64*64 时，矩阵每 4 行就能占满整个 cache，此时若仍按上述方案转置，会在调用 B 的过程中产生抖动，即每个 8*8 矩阵前 4 行和后 4 行之间的反复冲突与驱逐。所有需要考虑将最大跨度更改为 4 行。则考虑将每个 8*8 的矩阵内部再细分成 4 个 4*4 的小矩阵进行处理：先一次性将 A 中左上和右上的数据全部取出，转置后存入 B 中左上和右上（非左下，避免抖动）；再将 A 中左下的数据按列取出，且将 B 中即将被替换的数据按行取出

保存，将刚从 A 取出的数据转置后存入 B 中右上，再将保存的 B 中数据按行存入 B 中左下；最后将 A 中右下数据转置后存入 B 中右下。

对于 61*67 的矩阵，因为 61 和 67 不是 2 的倍数也不是 2 的幂，故不方便将整体分块为整数个从而进行优化，考虑对分块大小进行尝试。实验表明 16*16、17*17、18*18 和 19*19 都能达到性能要求，其中 17*17 的划分最优。

32×32：运行结果截图

```
zr@ubuntu:~/桌面/cache/cachelab-handout$ ./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

64×64：运行结果截图

```
zr@ubuntu:~/桌面/cache/cachelab-handout$ ./test-trans -M 64 -N 64

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9074, misses:1171, evictions:1139

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1171

TEST_TRANS_RESULTS=1:1171
```

61×67：运行结果截图

```
zr@ubuntu:~/桌面/cache/cachelab-handout$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6229, misses:1950, evictions:1918

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1950

TEST_TRANS_RESULTS=1:1950
```

第 4 章 总结

4.1 请总结本次实验的收获

本次实验很有趣味，我在实践中加深了对 cache 实现的理解，尝试实现了不命中时的 LRU 策略。还在对一些矩阵转置操作的实现过程中，进一步掌握了编写高速缓存友好代码的方法。

4.2 请给出对本次实验内容的建议

建议多设置这样的实验，能将学习到的较为抽象的理论知识很好地可视化，在实践中加以呈现，内容清楚明了，结果也方便检查验收。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.