华中科技大学

计算机系统结构实验报告

姓　　名：　　李椿

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2002

学　　号：　　U202015332

分数	
教师签名	

2023 年　4 月 28 日

目 录

1. Cache 模拟器实验

1.1. 实验目的

理解 Cache 的基本工作原理，使用 C 语言编写一个高效的事件驱动的 Cache 模拟器。实现的 Cache 可以处理来自 Valgrind 的跟踪和输出统计信息，如命中、未命中和逐出的次数，其中淘汰策略采用 LRU 算法。

1.2. 实验环境

Linux 64-bit ，C 语言

1.3. 实验思路

整体的设计思路如下：通过设计一系列的数据结构和函数，处理命令行输入的 cache 参数。初始化 cache 后，读出文件中的每一条指令，并根据指令类型和存储器地址对 cache 进行不同的访问。在访问 cache 时，根据不同的查找结果对 cache 进行更新，并记录本次访问的结果。

(1) 数据结构与全局变量设计

1. 定义 opt 变量 s、b、E、B、S，用来存储命令行输入的 cache 的大小，并定义辅助变量 verbose，用来标记输入的命令行参数中是否带有-v。

2. 定义全局时刻表 T，初始化为 0，用于 LRU 算法淘汰 cache 行。

3. 定义 cache 的行结构体 lineNode，考虑到未初始化的 cache 行其有效位与时刻均为 0，同时考虑到本实验中不需要存储每一个 cache 行实际的块，因此结构体中的数据成员为时刻 t 和标记 tag，并定义行结构体指针为 cache 组 groupNode。

4. 定义 cache，为 cache 组 groupNode 的数组。

5. 定义枚举类型 Category，依次存放 HIT、MISS、EVICTION，同时定义对应的字符串数组 categoryString，用以输出相应的信息。

6. 定义最终结果数组 result，用来保存命中、未命中和逐出的次数。

(2) 关键函数设计

1. 定义处理命令行参数函数 opt

函数的参数是命令行输入的字符数量 argc 和字符串指针 argv，返回已经打开的文件指针。

在函数中，使用 for 循环和 getopt 函数读入命令行的对应参数，每读到一个字符，使用 switch 语句判断是 hvsEbt 中的哪一个。若读到的字符为 h，则输出帮助文档；若为 v，则使用变量 verbose 进行标记，以输出详细的运行过程信息；若为 s、E 或 b，则使用 atoi 函数将 optind 参数指向的字符串转为数字，并进行非法性判断，再更新 cache 的组数 S 和块的大小 B；若为 t，则以只读方式打开 optind 指向的文件。

2. 定义初始化 cache 函数 init_Cache

在函数中，根据读取到的 cache 组数 S，使用 malloc 函数为 cache 分配 S 个 cache 组的空间；根据读取到 cache 每一组的行数 E，使用 malloc 对每个 cache 组分配 E 个 cache 行的空间，同时初始化每一行的时间 t。

3. 定义更新 cache 函数 update_Cache

函数的参数是 cache 组 group、cache 行索引 line_idx、查找结果 category、标记 tag 和保存运行信息的字符串数组 resultV。

在函数中，首先判断行索引是否合法，然后使用全局时刻 T 和传入的标记 tag 更新 group 中行索引 line_idx 对应的行，同时在结果数组 result 中记录查找结果。若需要输出运行信息，则在 resultV 中使用 strcat 函数追加信息。

4. 定义查找 cache 函数 search_Cache

函数的参数是标记 addr_tag、组索引 group_idx 和保存运行信息的字符串数组 resultV。

在函数中，首先根据组索引在 cache 中取得查找的 cache 组 group，同时定义记录最久未访问的行索引变量和空行的行索引变量。然后遍历 group 中的每一个 cache 行，若行非空且命中，则调用 update_Cache 函数对当前行进行 HIT 更新，并直接返回。在遍历过程中，不断更新最久未访问的行索引和空行的行索引，在退出遍历后，调用 update_Cache 函数进行 MISS 更新，若此时的空行仍为初始值，则再次调用 update_Cache 函数对最久未访问的行进行 EVICTION 更新。

(3) 主函数 main 调用流程

在主函数中，首先调用 opt 函数，处理命令行参数，得到文件指针；然后调用 init_Cache 函数初始化 cache。读取文件中的每一条指令，得到指令的类型存储器地址和访问的字节数，如果读到的指令是指令加载 I，则继续读取下一条指令，否则取出存储器地址中的组索引和标志，调用 search_Cache 函数进行访问，如果读到的指令是数据修改 M，则需要再次调用 search_Cache 函数。如果 verbose 被标记，则需要输出每一条指令的运行结果。最后调用 printSummary 函数提交最终结果，并释放内存。

主函数 main 的执行流程图如图 1.1 主函数 main 的执行流程图所示。

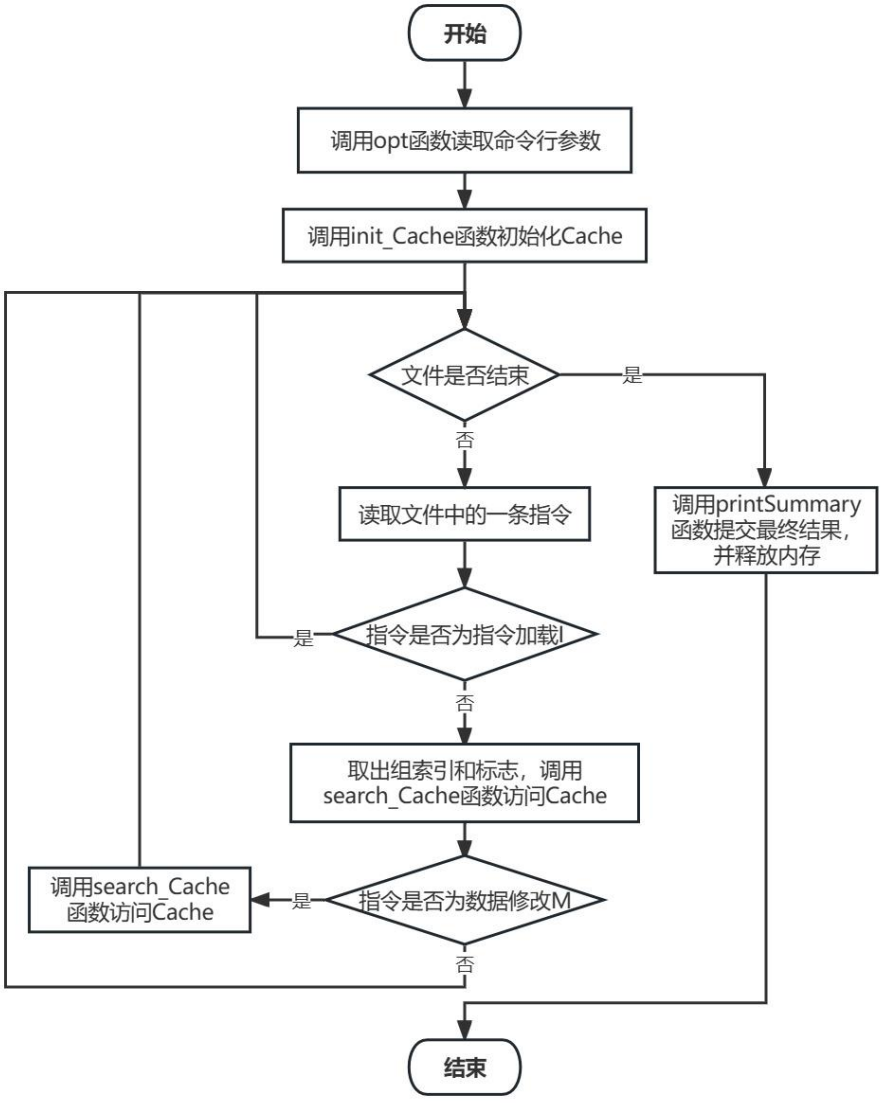

图 1.1 主函数 main 的执行流程图

1.4. 实验结果和分析

实验结果如图 1.2 Cache 模拟器实验结果所示，并通过了 educoder 平台的测试，说明所设计的 Cache 模拟器能够正确模拟缓存相对内存访问轨迹的命中、缺失和逐出行为，并对结果做出正确统计。

```
测试模拟器
运行 ./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

yi2.trace测试通过,
yi1.trace测试通过,
dave.trace测试通过,
trace.trace测试通过,
long.trace测试通过!
```

图 1.2 Cache 模拟器实验结果

2. 优化矩阵转置实验

2.1. 实验目的

编写一个实现矩阵转置优化的函数。所实现的函数能够处理不同规模的矩阵转置，在调用过程中对 Cache 的不命中次数尽可能少。

2.2. 实验环境

Linux 64-bit ，C 语言

2.3. 实验思路

整体的设计思路如下：对 32x32、64x64 和 67x61 大小的矩阵分别实现转置优化函数，在函数中采取不同的分块策略，考虑到访问寄存器的速度远大于访问主存的速度，借助临时变量，使得矩阵转置的不命中率尽可能低。

(1) 32x32 矩阵转置优化思路

1. 分块依据：一个 cache 的块大小为 32 字节，即 8 个 int。考虑 8x8 分块，则所需要的最大 cache 行数为 8+8=16，考虑 16x16 分块，则需要的最大 cache 行数为 2x16x2=64 行，超过了 cache 的容量，会导致矩阵内的重叠。因此将矩阵分割成若干个 8x8 的矩阵块。

2. 实现函数：在实现函数中，分块读取整个矩阵。一次读取 8x8 分块中的一行元素，存储在临时变量中，也即是存储在寄存器中，再将其拷贝到列的对应位置。

3. 不命中次数：在每个 8x8 分块中，按行读取的矩阵每行只会 miss 一次，按列读取的矩阵第一次读取时 miss 八次，此后读取列时只有包含对角线元素的那一列会 miss 一次。而分块共有 4x4=16 个，因此理论上的总 miss 数为 16x(8+8)+31=287 次。

(2) 64x64 矩阵转置优化思路

1. 分块依据：若直接采用 8x8 分块，由于每 4 行组索引就会重复，因此不命中率大幅度提升；若采用 4x4 分块，会出现资源浪费导致的 miss 次数上升。因此将矩阵划分成 8x8 的分块，在每个分块内再分成 4x4 的小分块。

2. 实现函数：不妨假设原 8x8 分块为 A，转置后的 8x8 分块为 B，划分成的 4x4 分块从上到下从左到右依次编号为 A11、A12、A21、A22 和 B11、B12、B21、B22。在实现函数中，转置分为以下三步：

第一步，将 A11 和 A12 翻转到 B11 和 B12 中。一次读出 A11 和 A12 的同一行元素，存储在临时变量中，然后将其存放到 B11 和 B12 的同一列中。此时 A11 已完成向 B11 的转置，B12 中存放着理应放在 B21 的 A12 的转置。

第二步，将 A21 翻转到 B12，同时将 B12 转移到 B21。读出 A21 的同一列元素和 B12 的同一行元素，存储在临时变量中，然后分别存放到 B12 的同一行和 B21 的同一行中。此时，A21 已完成向 B12 的转置，原 B12 中存放的 A12 的

转置也转移到 B21 中。

第三步，将 A22 翻转到 B22。每次读出 A22 中的一行，存储在临时变量中，再存放到 B22 中的同一列中。此时，A22 已完成向 B22 的转置。

3. 不命中次数：在上述的三个步骤中，按行读取的分块每行只会 miss 一次，按列读取的分块第一次读取时 miss 四次，此后读取列时只有包含对角线元素的那一列会 miss 一次，再次按列读取的同行元素会全部命中。而 8x8 分块共有 8x8=64 个，因此理论上的总 miss 数为 64x(8+2+8)=1152 次。

(3) 67x61 矩阵转置优化思路

1. 分块依据：对于不规则的矩阵，转置前后的相同组索引的联系减弱，此时可以不用考虑对角线的情况。不断猜测不同分块大小的效果，分块大小为 16、17、18 等等都能通过测试，因此因此将矩阵分割成若干个 17x17 的矩阵块。

2. 实现函数：由于分块大小为 17，无法使用寄存器存储数据，因此采用直接赋值的方法进行转置。在每个分块中，还需要考虑分块不能超过原有矩阵的边界。

2.4. 实验结果和分析

实验结果如所图 2.1 优化转置矩阵实验结果示，并通过了 educoder 平台的测试，说明所完成的不同规模的矩阵优化转置函数的功能正确，矩阵转置的缺失总数符合要求，即矩阵优化转置函数的性能合格。

测试矩阵转置

实验汇总:

转置矩阵类型	得分	该项分值	是否有效
32x32	8.0	8	287
64x64	8.0	8	1179
61x67	10.0	10	1950
合 计	26.0	26	

矩阵32x32转置优化完成,
矩阵64x64转置优化完成,
矩阵61x67转置优化完成!

图 2.1 优化转置矩阵实验结果

3. 总结和体会

在本次实验中，我完成了 Cache 模拟器的设计与实现，也完成了矩阵转置的优化。通过本次实验，我不仅理解了 Cache 的基本工作原理，组相联 cache 的，熟练掌握了 LRU 算法的实现和使用，而且掌握了优化矩阵转置的原理和方法，理解了矩阵分块能够有效提高矩阵转置的命中率，同时也认识到矩阵分块过大会导致矩阵在 cache 中重叠，矩阵分块过小则会增加 cache 的不命中率。

在完成 64x64 矩阵的转置优化时，一开始我直接将原矩阵分成 8x8 的分块后，发现转置后的矩阵在写入时，每次访问都是 eviction，输出运行信息后，发现这里每 4 行的组索引正好重复，矩阵正好重叠了。而在我换成 4x4 分块后，虽然解决了冲突，但是缓存利用率不高，命中率也没有明显提升。经过一段漫长的思考，以及在网络上查询相关处理的时候，才了解到在 8x8 的分块内分成 4 个 4x4 个分块就能解决组索引冲突的问题，这也才看懂老师在 educoder 平台上给的分析思路与提示中的 A11、A12、B12、B21 所表达的意思。理解了在分块中再次分块的想法，我顿时豁然开朗，既然只能处理 4x4 的分块，为什么不在 8x8 的分块中处理呢。

通过本次实验，我不仅学习和掌握很多知识，而且将这些知识在代码中运用起来，也通过自己的思考将知识融会贯通了，我收获颇丰。

4. 对实验课程的建议

对于本次实验，在看到实验标题和实验内容时，我就意识到这个实验是深入理解计算机系统的实验，即 CSAPP Lab。虽然在网络上能找到很多本实验的实验讲解，但是在童老师的班级里没有提供实验指导资料，也许是老师认为 educoder 提供的信息已经足够吧。我还是建议能给一个实验指导资料，或者 CMU 的实验说明，毕竟网络上的资料良莠不齐，有老师给的资料能少走点弯路。

此外，我还建议能够提供本地实验环境，在 linux 上调试会方便许多，而且 educoder 平台的编译器上总有一些无法解释的 bug。