实验 4 构建 cache 模拟器

实验时间 2023.6.6

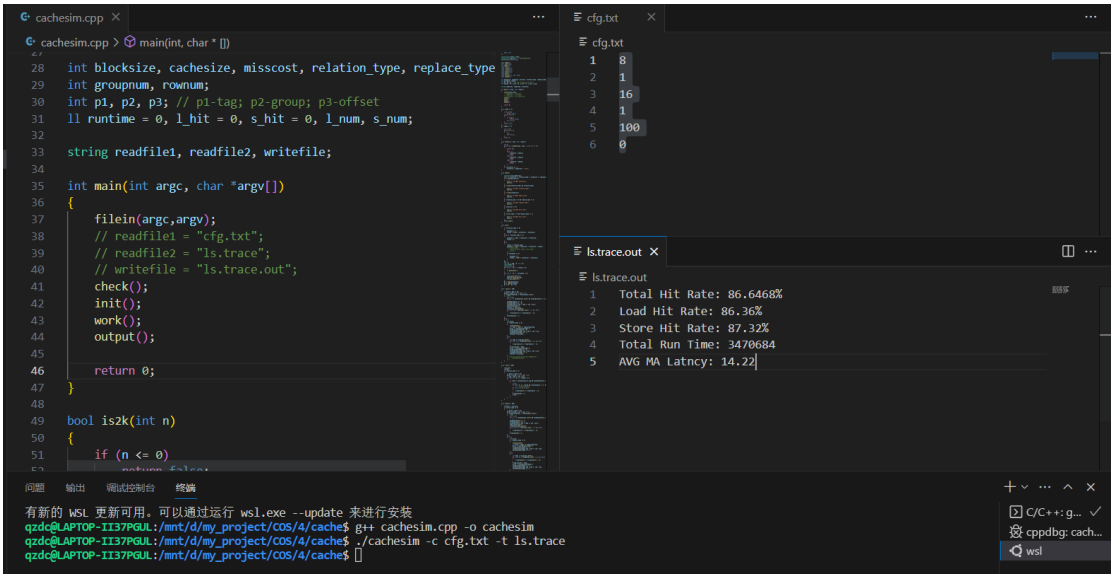
1 实验目的

完成 cache 模拟器
理解 cache 块大小对 cache 性能的影响
理解 cache 关联性对 cache 性能的影响;
理解 cache 总大小对 cache 性能的影响
理解 cache 替换策略对 cache 性能的影响
理解 cache 写回策略对 cache 性能的影响

2 实验过程

2.1 实验结果

按照老师所给的配置文件和跟踪文件，运行结果如下：



总命中率为 84.4360%，总时钟周期为 3470684，平均取数延迟为 14.22

2.2 设计目标

需要实现以下功能:

读取配置文件，按配置文件实现 cache 模拟器

对给出的跟踪文件进行回放，计算并输出 cache 模拟器的性能参数。

使用命令行，编译生成 cachesim 程序，典型用法包括

cachesim -c cfg.txt -t ls.trace [-o ls.trace.out]

其中:

-c 后面的参数指定配置文件:

-t 后面的参数指定跟踪文件;

-o 后面的参数指定输出文件，该参数是可选的(意味着该参数可以不写)，此时直接输出到标准输出。

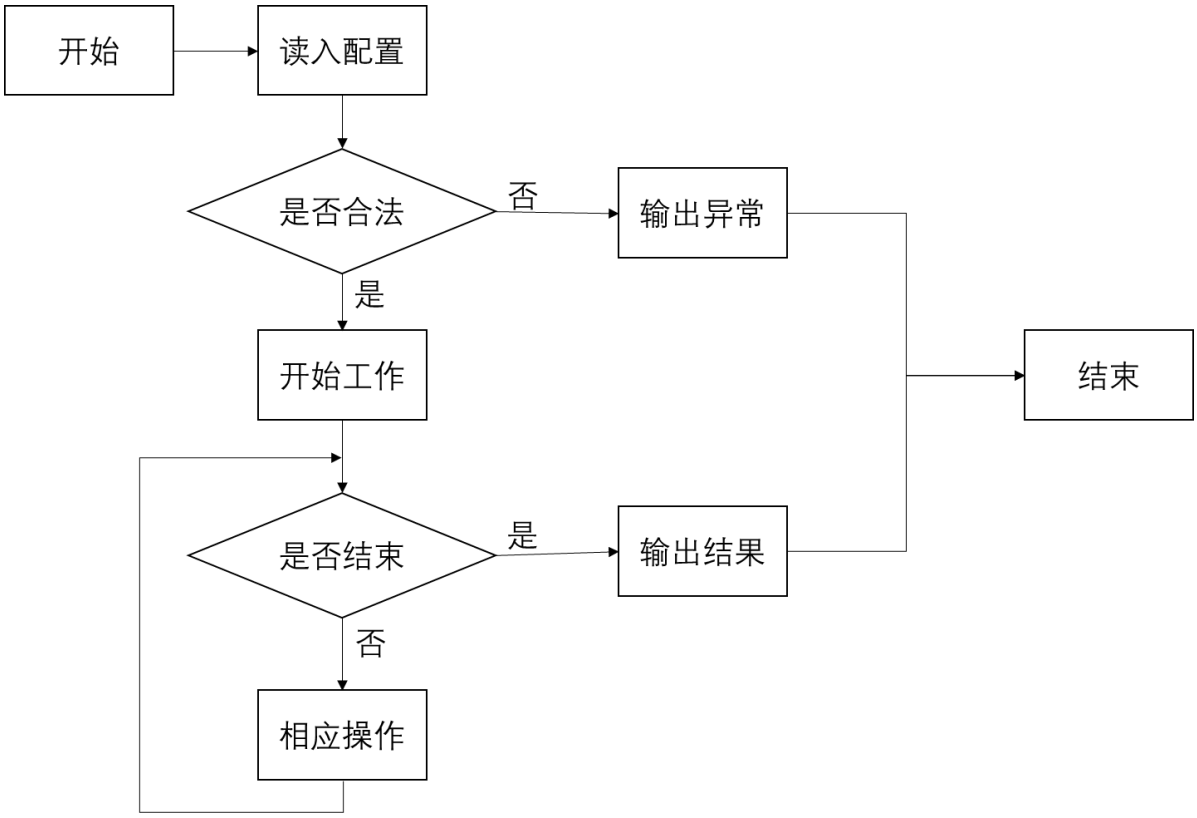
2.3 设计思路

使用 C++语言实现。

对于标准输入输出，由于在命令行中实现参数的传递，因此需要使用 getopt() 函数，通过读入 main 函数的参数，实现文件的选择。选择的文件必须和程序在同一文件夹下。输出的文件可以自己指定，如果不指定，就默认是跟踪文件文件名后加上.out。

对于每一步的执行，每次循环读入一行跟踪文件，根据读还是写、命中还是不命中，结合配置文件的信息，模拟 cache 在各种操作下的行为，并更新计数器的信息。直到读到文件结束标志，即 cin 重载的输入流符号返回值为 false 时结束循环。

下面是主函数的流程图：



下面是主函数的代码：

```
int main(int argc, char* argv[]){
    filein(argc,argv);
```

```
    // readfile1="cfg.txt";
    // readfile2="ls.trace";
    // writefile="ls.trace.out";
    check();
    init();
    work();
    output();

    return 0;
}
```

下面将逐个介绍函数的功能。其中注释部分是为了方便调试，在非命令行下 Debug，选择的默认文件。

2.4 变量与函数

变量

通过使用结构 ROW 模拟每一行的 cache，嵌套的 vector 模拟 cache，第一层索引是 Group Number，第二层索引是组内的行号。由于 cache 是提前分配好的空间，所以不能用 size()函数得到每一组的已有数据的个数，需要使用了 int 型的 vector，记录 cache 每一组已有数据个数，用于判断是否满了。

还使用了嵌套的 vector 用于实现 LRU 替换方式的列表。每一个组有一个对应的 LRU 表，所以需要两层嵌套。不使用 deque 或者 stack 结构是因为不方便查找，所以用数组模拟。由于不是实现分配好的空间，所以已有数据可以通过 size()得到，所以不需要额外的辅助数组。

```
struct ROW {
    ll tag;
    bool v,d;
};

vector<vector<ROW>> cache;
vector<int> capacity;//每组已有数据个数
vector<vector<int>> lru;
```

接下来是全局变量，blocksize 为块大小，cachesize 为 cache 数据区大小，misscost 为不命中的开销，relation_type 为相连方式，replace_type 是替换策略，alloc_type 代表是否是写分配。

groupnum 是组的总数，rownum 是每一组内行的总数，p1 是地址中 tag 的位数，p2 是组号的位数，p3 是块内地址的位数，这些是根据配置文件计算得到的。

runtime 是运行时间， l_hit=是读命中次数，s_hit=是写命中次数，l_num 是读的总次数，s_num 是写的总次数。

readfile1 是配置文件的文件名，readfile2 是跟踪文件的文件名，writefile 是输出文件的文件名。

```
int blocksize,cachesize,misscost,relation_type,replace_type,alloc_type;
int groupnum,rownum;
int p1,p2,p3;//p1-tag; p2-group; p3-offset
ll runtime=0, l_hit=0, s_hit=0, l_num, s_num;

string readfile1,readfile2,writefile;
```

函数

is2k 函数判断输入的函数是否是 2 的 k 次方，log2 函数返回输入值 log2 的值。init 用于初始化，work 用于循环执行 cache，filein 用于读入命令行的参数，output 函数用于输出结果。r_miss，r_hit，w_miss，w_hit 分别是读不命中，读命中，写不命中，写命中，用于模拟 cache 的操作。

```
void check();
bool is2k(int);
int log2(int);
void init();
void r_miss(ll);
void r_hit(ll);
void w_miss(ll);
void w_hit(ll);
void work();
void filein(int, char*[]);
void output();
```

2.5 各个模块详细介绍

读入命令行参数模块 filein()

使用 getopt 函数和 swithc-case 结构，得到相关文件名，并且如果没有指定输出文件，就使用默认输出文件。

```
    int o;
    while ((o = getopt(argc, argv, "c:t:o:")) != -1) {
        switch (o) {
            case 'c':
                readfile1=optarg;
                break;
            case 't':
                readfile2=optarg;
                break;
            case 'o':
```

```
                writefile=optarg;
                break;
        }
    }
    if(writefile=="")
        writefile=readfile2+".out";
```

读入配置文件并检查模块 check()

从文件输入流读入配置数据，然后判断是否符合格式，比如块大小、cache 总大小是否是 2 的幂次、写回策略、替换策略不为 0 或 1 等。不符合就使用 exit 函数终止程序并输出。如果都符合就关闭配置文件的输入流。本模块及其之后的代码较为简单，就不展示了，完整代码见云盘链接。

初始化模块 init()

根据输入的配置文件，计算组数和行数，划分 48 位的地址。

如果是全相联法，就相当于有 1 组，cachesize/blocksize 行；

如果是直接映射，就相当于 cachesize/blocksize 组，每组 1 行。

接下来根据刚才计算出的行数以及组数，对 cache、LRU、capacity 分配空间。

读不命中模块 r_miss()

由于初始化的处理，就不需要对相连方式做分类讨论。取出地址位之后，如果是组相联或者直接映射，得到的就是组号或行号，如果是全相连就得到 0，就是“组号”。按下面方式取出组号：

```
    ll group = add >> p3;
    group = group & ((1 << p2) - 1);
```

然后判断所在组是否已满，满了就不换，直接加入 cache 即可，不满就需要根据替换策略，是 LRU 还是随机替换，进行替换的操作。

最后需要将运行时间增加额外开销的时间。

本来还考虑了是否需要处理 dirty 位，即被替换的 cache 组如果是 dirty 的，就要额外的开销来写回内存。但是跟老师沟通后，老师说这个是自动完成的，不需要额外的开销。因此就不考虑了。

```
    runtime +=misscost;
    if (capacity[group] < cache[group].size())
    { // 不换
        直接放入 cache
    }
    else
    { // 换
        if (replace_type == 0)
```

```
        {
            随机寻找行数替换
        }
        else
        {
            最久未使用的替换，并更新 LRU
        }
    }
```

读命中模块 r_hit()

按刚才的方法读出 group，取 add 的高 p1 位作为 tag。

命中次数和运行时间都加一，并且如果替换策略是 LRU 就要把相应的项放在最前面，代表最近使用。

```
    runtime++;
    l_hit++;
    if (replace_type == 1)
    {
        更新 LRU
    }
```

写不命中模块 w_miss()

取出 group 和 tag。运行时间增加额外开销。

接下来，如果写策略是“写回”，就意味着写不命中是采用“写分配”的，需要修改 cache 内容，判断所在组是否已满，满了就不换，直接加入 cache 即可，不满就需要根据替换策略，是 LRU 还是随机替换，进行替换的操作。

如果写策略是“写直达”就意味着不对 cache 进行操作，直接写内存。

```
    runtime += misscost;
    if (alloc_type == 0)
    {

        { // 不换
            直接加入 cache
        }
        else
        { // 换
            if (replace_type == 0)
        {
            随机寻找行数替换
        }
        else
```

```
        {
            最久未使用的替换，并更新 LRU
        }
    }
```

写命中模块 w_hit()

取出 group 和 tag。命中次数加一。

如果写策略是“写回”，运行时间就加 1，然后将其 dirty 位置为 1。如果写策略是“写直达”，运行时间增加额外开销，因为要写内存和 cache。

接下来，如果是 LRU 的替换策略，就需要将命中项放在数组最前，代表最近使用。

```
    s_hit++;

    if (alloc_type == 0)
        runtime++;
    else
        runtime +=misscost;
    if (replace_type == 1)
    {
        更新 LRU
    }
```

工作模块 work()

本模块先打开追踪文件，然后循环读入追踪文件，直到文件结尾或遇到#eof 标签，每次判断是读还是写，是否命中，然后转相应的处理程序。

使用 stringstream 的方式将十六进制字符串转为十进制数。由于有 48bit，所以得用 long long 型数据：

```
while (fin2 >> t >> rw >> temps)
    {
        flag = false;
        sin.clear();
        sin << hex << temps;
        sin >> add;
       if(rw == 'R'){
            ……
        }
        else{
            ……
        }
       //主体部分省略，完整版见实验结论的网盘链接;
    }
```

```
fin2.close();
```

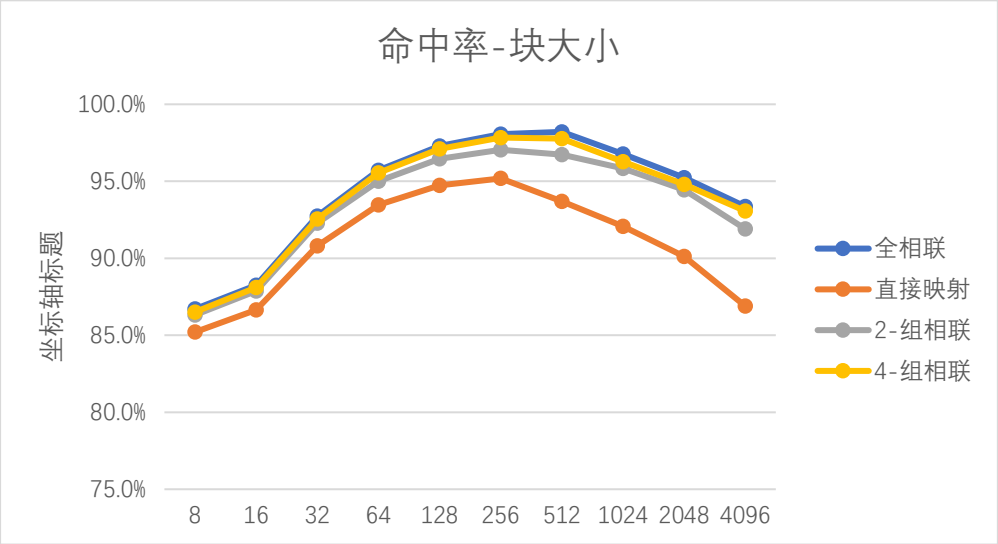
work 函数最后关闭读入的文件。

辅助模块

- is2k(int t) 判断输入的数是否为 2 的幂次
- log2(int t) 返回 $\log_2 t$
- output() 打开输出文件，根据得到的运行结果，按要求输出

2.6 cache 性能与其参数

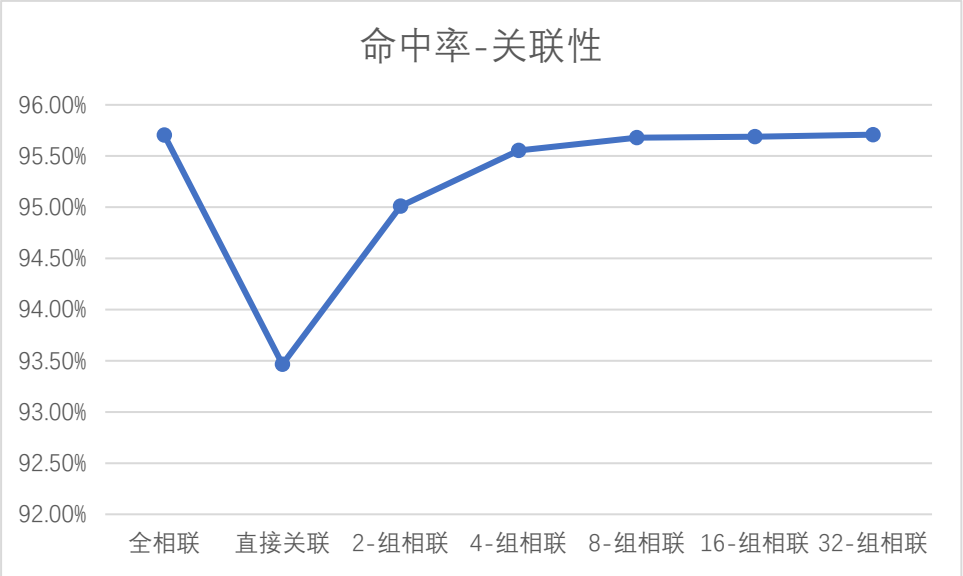
cache 块大小对 cache 性能的影响



可以看到，在其它条件不变的情况下，增大块大小会提高命中率，并且随着块大小的增大，提升块大小的提高作用变得不显著。但是超过 256B 之后，命中率就会开始下降。直接映射方式的下降幅度最大。

因为提升块大小可以增加块内可以放的数据容量，每一次 miss 带来的数据量也会增加，所以命中率会提高。但是如果块大小过大，那么 cache 的行号就会大幅减小，因此带来命中率的极度下降。由于直接映射最依赖行号，所以下降幅度最大。

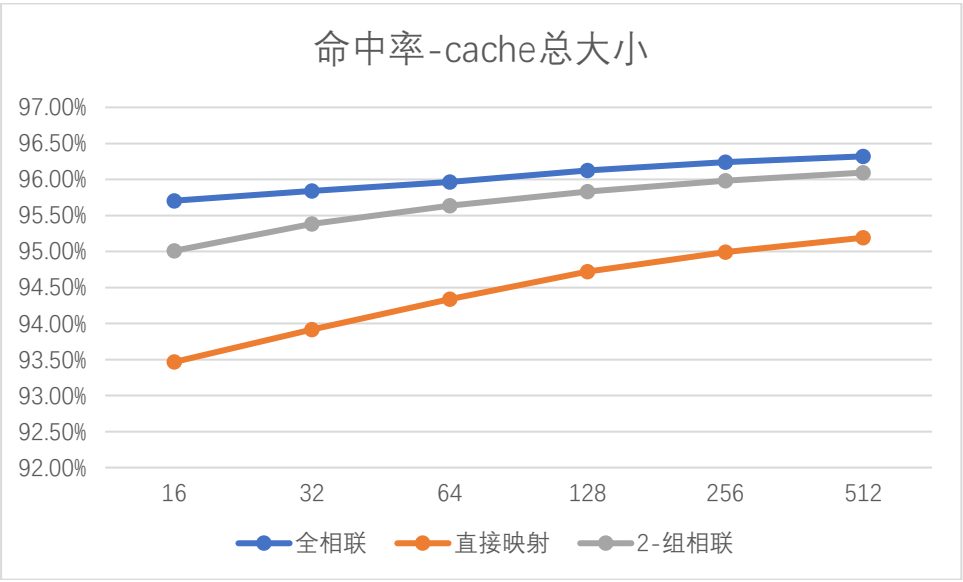
cache 关联性对 cache 性能的影响



可以发现，在全相联的情况下，cache 的命中率最高，直接映射的命中率最低，随着组相联的数量增加，cache 的命中率也增加，并且不断靠近全相联的情况。

因为全相联对 cache 的利用率较高，可以尽可能减少替换的次数，所以 cache 命中率较高。直接相连的替换率较高，所以可能有“颠簸”现象，但随着组相连个数增加，命中率会提高

cache 总大小对 cache 性能的影响

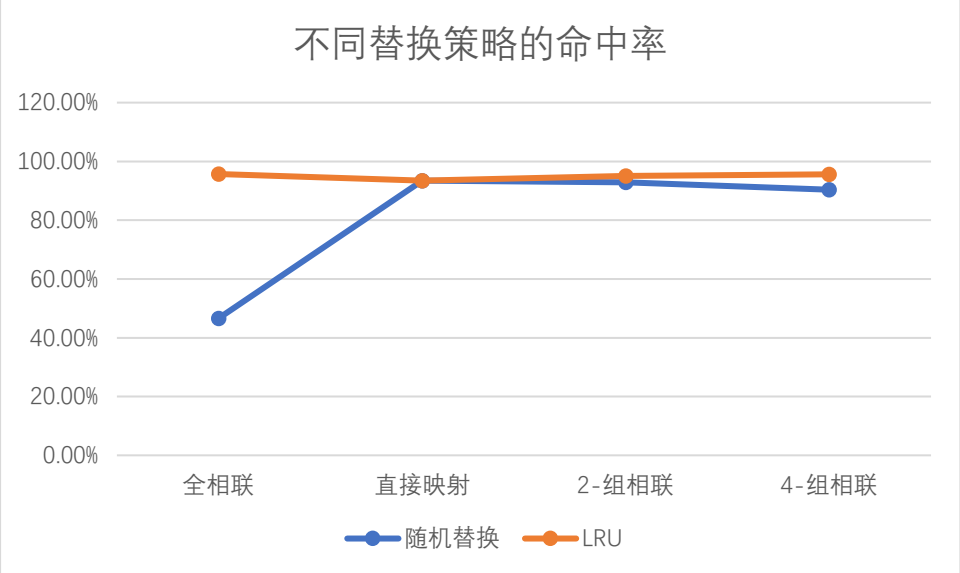


可以看到，在其它条件不变的情况下，增大 cache 大小会提高命中率，并且随着 cache 的增大，提升 cache 大小的提高作用变得不显著。并且这种

提高作用随着组相联的个数的增加而效果逐渐减弱。对直接映射的提高作用最大。

因为 cache 容量增加，导致可以放入更多条 cache，所以会提高命中率；由于直接映射需要尽可能多的行数，提高总大小明显可以提高行数。并且由于全相联对 cache 利用率已经很高，所以提升 cache 大小的效果不会很明显。

cache 替换策略对 cache 性能的影响

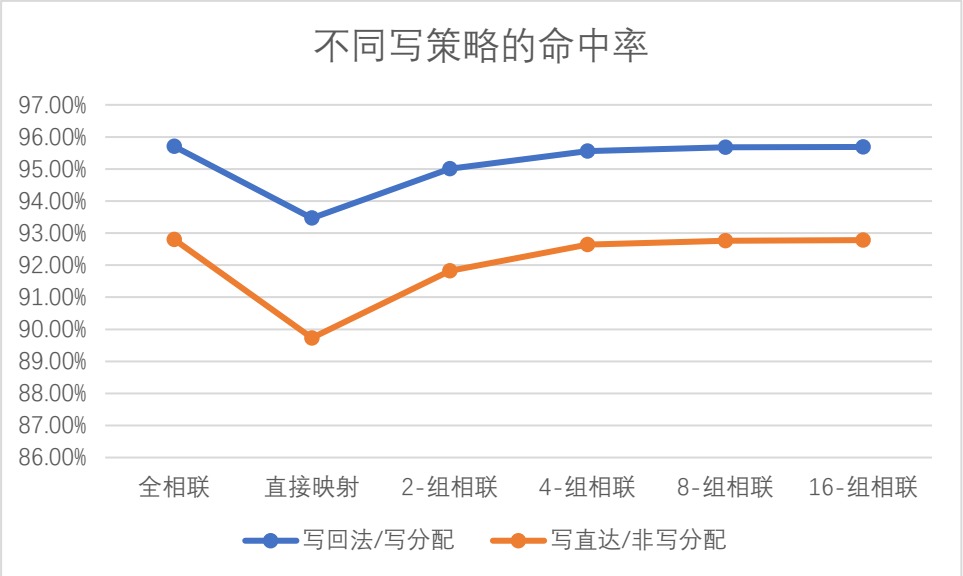


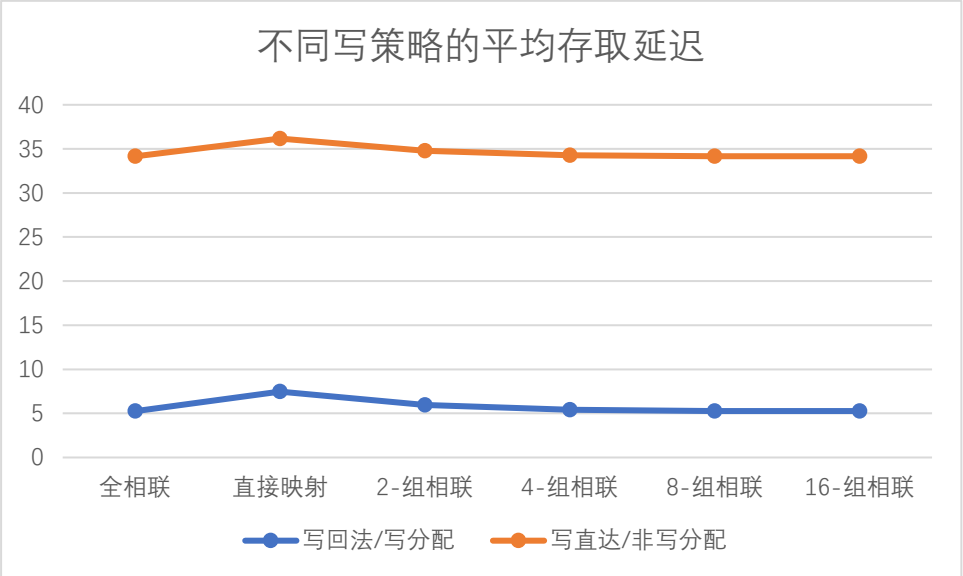
其中，随机替换的命中率是运行 5 次之后求出来的平均值。

观察发现，在直接映射时，替换策略不会影响命中率，随着组相联数目的增大，随机替换影响命中率也会增大，降低命中率。并且在全相联时，由于可以看成是一个组，是 cache 大小/块大小的相联方式，因此影响最大，甚至直接折半了。

这是因为直接映射相当于 1-组相联，每组只有 1 行，所以替换政策不会生效；随着组相联增大，每一组行数也增加，直至增加到上限，成为全相联，在这个过程中，采用随机替换，替换掉最近正在使用的 cache 行的概率就越大，因此对命中率的影响逐渐增大，甚至最后直接折半。

cache 写回策略对 cache 性能的影响





根据实验图像，发现对于各种映射方式，写回法的命中率都会高于写直达法。并且写直达的写命中策略的时间开销也是要远大于写回法的。

因为使用了写直达/非写分配就相当于关于写的操作跳过了 cache，没有发挥 cache 的作用，于是导致命中率的大幅降低、平均存取延迟的大幅增加，甚至接近额外开销的一半。

3 实验结论

本次实验通过使用 C++语言，编写了一个 cache 模拟器，可以通过命令行传递参数，选择配置文件、跟踪文件、输出文件，来模拟 cache 的操作。

实验采用老师所给的配置文件和跟踪文件，得到总命中率为 84.4360%，总时钟周期为 3470684，平均取数延迟为 14.22。

最后分析 cache 的性能与各个参数的关系，并且在 2.6 部分给出了原因分析。由于不考虑比较 tag 带来的时间损失，所以全相联方式可以做到高命中率、低平均存取延迟。

1. 在一定范围内，cache 命中率随块大小的增大而增大，超出 256B 后 cache 命中率随块大小增大反而减小，直接映射方式的下降幅度最大。
2. 在全相联的情况下，cache 命中率最高，直接映射的命中率最低，随着组相联的数量增加，cache 命中率也增加，并且不断靠近全相联的情况。
3. cache 命中率随 cache 大小的增大而提高，并且随着 cache 的增大，提升 cache 大小的提高作用变得不显著。并且这种提高作用随着组相联的个数的增加而效果逐渐减弱。对直接映射的提高作用最大。
4. 在直接映射时，替换策略不会影响 cache 命中率，随着组相联数目的增大，随机替换影响 cache 命中率也会增大，降低命中率。并且在全相联时，随机替换的影响最大，接近 LRU 的一半。
5. 对于各种映射方式，写回法的命中率都会高于写直达法。并且写直达的写命中策略的时间开销也是要远大于写回法的

本次实验的相关文件位于下面的链接中，其中 cachesim 是实验的主体的部分，cache_loop 是用于分析阶段的批量产生运行结果的部分，包含统计表格。

链接:https://pan.baidu.com/s/17su-MIgcJdEMQfLp44S2ig
提取码:lab4

---

4 实验感想

本实验是在老师讲授完 cache 部分，并且完成了相应的作业之后开始入手完成的，因此对于原理部分的理解有一定的基础，在实验过程中没有遇到原理上的不理解与混淆。唯一理解不准确的地方是关于 Dirty 位的，我以为只有在某一条 cache 被替换掉之后才检查 dirty 位，如果被修改过就要进行一次 Memory 的写操作，只会发生在命中的策略时。但是与老师沟通后明白这个过程是 CPU 不参与的，由其它硬件完成的。

本次实验在编写程序时，并没有遇到明显的 bug，但是有三次缩减程序体量、简化程序逻辑的操作。

在最开始是把所有东西都放在 main 函数里，并没有很好的模块化设计，导致项目极为庞大，大概 900 行，充斥着许多重复的代码。并且当时对于 cache 关联性的认识不够深刻，就根据全相联、直接映射、2^k 组相联由一

个 if-else 块，分别实现各自的初始化、读、写相关操作、判断命中相关操作。这就使得代码重复性极大，冗余多，修改起来不方便，逻辑复杂，导致混乱。

第一次优化是把 main 函数拆开，每个功能用一个模块实现。但是虽然逻辑清晰了一点，但是对于代码的复用率也不高，并且由于拆成多个函数实现，传递参数、局部变量相关的部分还是有好几次出 bug。在优化完成后程序并没有体量减小，达到 1000 多行。

第二次优化是在调试时发现，其实三种关联性可以当作一种来看，只要分配好各组、组内行数即可。于是将三种关联性的初始化、读写、判断等模块相合并，删除一部分冗余代码，将直接映射和 2^k 组相联合并成一种，不区分直接映射的组号和 2^k 组相联的行号。将全相联单独划出，与这一部分并列。优化完的代码只有 600 行。

第三次优化时在测试关联性对 cache 性能的影响时发现，其实全相联和（直接映射与 2^k 组相联）也可以合并，因为从地址里取出来的 group 组号在全相联的情况下就是 0，就是在全相联时的形式组号。所以将这部分也合并，最终代码只有 420 行。并且由于减少分类次数，代码的逻辑性更加明显了，更加易懂。

最后在 cache 参数与其性能的分析中，我也试着分析了现象的原因，并且实验的实际情况也都符合理论。

实验过程中，我学到了 getopt 函数的用法，以及第一次在 linux 下用带参数的命令行运行自己所写的程序，有着新的收获。还在读入的时候了解到了 stringstream 这一标准库，可以用于输入的十六进制字符转为十进制数字的操作。

这次有关 cache 的实验对于我理解 cache 原理、了解其性能与各参数的关系有着极大的帮助，不但加深我对 cache 的理解，而且锻炼了自己模块化设计的能力。实验中还通过各种方式尽可能地优化代码体积，简化代码逻辑，解决了各种小 Bug。总之，这次试验对我的个人能力有很大的提升。