南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

体系结构设计期末实验报告

Cache 的预取和替换策略研究

马琦

年级：20 级

专业：计算机科学与技术

指导教师：李雨森

2023 年 1 月 7 日

摘要

本次实验主要是探索 Cache 不同的预取和替换策略对 CPU 性能的影响。主要使用 ChampSim 模拟器完成了基于 GHB 的步长预取算法以及 LLC 上的 LFU 算法，同时对两种算法的基础实现进行改进，更好地降低了算法的复杂度，提升了算法的性能。另外针对论文中所提到的基于 Markov Chain 的预取算法也进行了复现，并取得了优于 GHB 的分数。

关键字：GHB、LFU、Pangloss

目录

一、 引入

当下由于半导体技术以及微体系结构的快速发展，处理器的时钟周期大幅度降低，而存储器现在也在往高密度方向发展，这使得以处理器时钟周期测量时，主存储器的延迟显著增加。那么为了解决这种问题，我们采用多级 Cahe 来加速 CPU 取数据的速度，然而 Cache 本身的容量是相当有限的，关联度也非常小，所以进一步的，我们需要设计一种能够更高效利用 Cache 的算法，本次实验我们主要是通过预取和替换算法，提前将需要的内存取出放入 Cache，如果 Cache 满了则淘汰掉其中特定的一些数据块，具体算法策略的好坏将直接影响到 CPU 运行的效率。

那么最朴素的想法，也是最初的思路，就是我们按照顺序，预取一段连续的内存，更高级一点的可能是先观察访问模式，如果出现了顺序访问的模式，那么就开始大规模顺序预取。这个在一定程度上是有效的，因为 CPU 执行时有相当一部分指令会进行连续地址的访问。不过针对其他的访问模式，这样的预取手段可以说是没什么作用，甚至可以说有很多的缺点，因为这种做法常常伴随着大量的替换。

本次实验我们通过基于 GHB 的预取算法和基于 Markov 的预取算法，尝试在不直接使用人工智能相关方法的条件下，最大程度地利用我们的访存模式，对之后的预取进行指导，同时基于 LFU 原理替换掉数据，以此来优化我们的 Cache。

二、 预取算法的实现

(一) 基于 GHB 的步长预取算法的实现

统的步长预取机制，传统的步幅预取使用一个表来存储关于单条 load 指令的与步幅相关的历史信息。因此，它使用 load 指令的 PC 值来索引历史表。每个表项记录对应的 load 指令的最近访存时发生缓存缺失的两个地址之间的步幅以及最近的一个发生缓存缺失的访存地址。当发生缓存缺失时，预取机制会使用当前的 load 指令的 PC 值来索引历史表，并将当前 load 指令的访存地址 addr 与表项中记录的最近一次的缓存缺失地址相减。得到的结果 s 如果与表项中记录的步幅一致，则这个时候预取器会感觉未来很有可能在 addr + n * s 的地址处也发生缓存缺失，此时，它便会向更低一级的 cache 发出预取请求，请求的地址就是 addr + s、addr + 2s、...、addr + (1 + d)s，其中，d 就叫做预取度。

传统的表预取有一些缺点，比如容易有过时的数据；预取键容易造成冲突；每个表项能存储的历史信息较少等不足。GHB 预取机制在一定程度上对这些问题进行了解决。GHB 预取机制主要有两个部件组成，分别是索引表 (Index table) 和全局历史缓冲区 (Global history buffer)。其中，索引表主要用于根据预取键来索引 GHB 缓冲区中的具体条目。全局历史缓冲区，是一个循环队列。其中每项存储一个地址以及一个指针。通过指针按时间顺序将一个预取键对应的地址给链起来。

类似于传统的基于表的步幅预取，基于 GHB 的步幅预取的预取键也是 load 指令的 PC 值。预取器通过预取键来索引 GHB 中的条目，这里我们使每个预取键对应的 GHB 中的条目为预取键对应的 load 指令的访存地址。并通过指针将这些地址链起来，我们只需要取出链上的最后三个地址，计算出来对应的两个步幅，如果两个步幅相等，就可以像步幅预取那样预取新的缓存行了。

下面我们根据 PC/CS、和 PC/DC 两种预取器的原理进行代码编写。

1. 视野开阔:PC/CS

索引表数据结构如下，定义结构体 IT 表示一个索引表表项，其中存储了该项索引的 GHB 表项的下标。定义 IT 数组来模拟索引表，索引表使用直相联，一个 PC 值对应的表项为 it[PC % IT_SIZE]。定义结构体 GHB 表示一个 GHB 表项，其中 cachineline_addr 为实际访问的缓存行地址，pc 为访问该缓存行的指令的 pc 值，prev 为 GHB 指针，指向 GHB 表中上一个该 pc 对应的表项。定义 GHB_ENTRY 数组来模拟全局历史缓冲区，下方代码的 curr_idx 为新的 GHB 表项要存入的 GHB 表的位置下标。

数据结构

```
static double cache_ac = 0, cache_miss = 0;
static ui cur_idx = 0;
struct GHB {
    ull pc;
    ull cacheline_addr;
    ui prev;
};
static GHB my_GHB[GHB_SIZE];
static map <int, ui> it;
```

当 L2C 接收到一个访问请求时，GHB 步长预取机制会进行一下以下几步操作:

- 将请求的地址插入到 GHB 表中，并更新索引表。
- 得到 GHB 中该 pc 对应的地址链，并计算地址链上最后三个地址之间的步长。
- 若计算得到的两个步长相等，则发出预取请求，根据规则预取若干缓存行到 L2C 中；若两个步长不相等，则不进行任何操作。

主要实现的步骤如下，首先，计算出请求的缓存行地址。然后根据 ip 来索引索引表，得到当前 ip 对应的 GHB 表项。若 GHB 表项中的 pc 值等于 ip，则表示该索引表项是有效的，将新插入的 GHB 表项的指针指向索引表项指向的 GHB 表项即可; 否则表示索引表无效，即索引表项索引的并不是对应于当前 ip 的 GHB 表项，此时将新插入的 GHB 表项的指针指向新插入的 GHB 表项自身，从而表示链表的结尾。之后将当前缓存行地址和当前 ip 赋值到新插入的 GHB 表项的对应属性中即可。然后再更新索引表，将当前 ip 对应的索引表项指向新插入的 GHB 表项。之后就是第三步，这里为了简化流程，采用类似状态机的写法，每次执行步骤，如果中间不符合要求则直接退出循环，否则继续进行下一步操作。

预取的具体步骤

```
    uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip,
        uint8_t cache_hit, uint8_t type, uint32_t metadata_in)
{
    ++ cache_ac;
    if (!cache_hit)
        ++ cache_miss;
    uint64_t Cacheline_addr = addr >> LOG2_BLOCK_SIZE;
    my_GHB[cur_idx].cacheline_addr = Cacheline_addr;
    my_GHB[cur_idx].pc = ip;
    if (my_GHB[it[ip % IT_SIZE]].pc == ip)
```

```
        my_GHB[cur_idx].prev = it[ip % IT_SIZE];
    else
        my_GHB[cur_idx].prev = cur_idx;
    it[ip % IT_SIZE] = cur_idx;

    uint64_t prefetch_addr;
    ll stride1 = my_GHB[cur_idx].cacheline_addr;
    ui ele_idx = cur_idx;
    ll stride2 = 0;

    for (int i = 0; i < 4 && my_GHB[ele_idx].prev != ele_idx && my_GHB[
        ele_idx].pc == ip; ++ i)
    {
        ui prev_idx = my_GHB[ele_idx].prev;
        switch (i)
        {
            case 1:
                ele_idx = prev_idx;
                stride2 = my_GHB[ele_idx].cacheline_addr;
                stride1 = stride1 - stride2;
                break;
            case 2:
                stride2 = stride2 - my_GHB[prev_idx].cacheline_addr;
                break;
            case 3:
                if (stride1 == stride2)
                {
                    for (int offset = LOOK_AHEAD * stride1; offset <= (
                        LOOK_AHEAD + PREFETCH_DEGREE) * stride1; offset +=
                        stride1)
                    {
                        prefetch_addr = (Cacheline_addr + offset) <<
                            LOG2_BLOCK_SIZE;
                        prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
                    }
                }
                break;
        }
    }

    cur_idx = (cur_idx + 1) % GHB_SIZE;
    return metadata_in;
}
```


2. 局部最优:PC/DC

PC/DC 是基于局部历史表的预取算法，其中，GHB 表示局部历史表，它记录局部地址被访问的次数。它以某个局部地址被访问的次数作为其访问优先级的依据，将访问次数较多的地址放

在预取队列中，以便提高缓存命中率。GHB PC/DC 算法将以最近的两个步长作为基准，不断向前寻找地址链上是否存在相同的步长组合。这和之前 PC/CS 实现有很大区别，之前的算法强制要求两个步长一定相等, 这样的策略在某些情况下并不合理。我们采用另外的一种设计思路，增加能够预取的阶段，和 PC/CS 代码主要有区别的为下面这段代码, 我们通过增加了 3 阶段和 4 阶段，使得我们可以去寻找匹配步长相同的地址，后面实践中也可以发现，这样做确实效果更好。

预取的具体步骤

```
for (int i = 0; i < 5 && my_ghb[elem_idx].prev != elem_idx && my_ghb[elem_idx].pc == ip; ++ i)
    {
        unsigned int prev_idx = my_ghb[elem_idx].prev;
        switch (i)
        {
        case 0:
            stride2 = my_ghb[prev_idx].cl_addr;
            stride1 = stride1 - stride2;
            dels.insert(dels.begin(), stride1);
            elem_idx = prev_idx;
            break;
        case 1:
            delta = my_ghb[prev_idx].cl_addr;
            stride2 = stride2 - delta;
            dels.insert(dels.begin(), stride2);
            elem_idx = prev_idx;
            break;
        case 2:
            if (stride1 == stride2)
            {
                need_next = 0;
                for (int i = LOOK_HEAD; i <= LOOK_HEAD + PREFETCH_DEGREE; i++)
                {
                    prefetch_addr = (cl_addr + i * stride1) << LOG2_BLOCK_SIZE;
                    prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
                }
                i = 6;
            }
            break;
        case 3:
            delta = delta - my_ghb[prev_idx].cl_addr;
            dels.insert(dels.begin(), delta);
            if (delta != stride1) i --;
            delta = my_ghb[prev_idx].cl_addr;
            elem_idx = prev_idx;
            break;
        case 4:
            delta = delta - my_ghb[prev_idx].cl_addr;
```

```
            dels.insert(dels.begin(), delta);
            if (delta == stride2)
            {
                need_next = 0;
                for (int i = 2; i - 2 < PREFETCH_DEGREE2 && i < dels.size();
                    i++)
                {
                    prefetch_addr = (cl_addr + dels[i]) << LOG2_BLOCK_SIZE;
                    prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
                }
                i = 6;
            }
            else
            {
                i = 2;
            }
            delta = my_ghb[prev_idx].cl_addr;
            elem_idx = prev_idx;
            break;
        }
    }
```

(二) 基于 Markov 预取器的预取实现

1. Pangloss 预取算法

这个主要是根据论文 [1] 的简化实现的，这个算法其实就是一种 Markov 预取器。距离预取是常见 Markov 模型预取器的推广，它使用增量而不是地址来构建更一般的模型。而在我们这个实验中，它是一个二级缓存预取方案，根据当前增量提供增量转换。它由当前增量索引，并且每个集合中的块表示最频繁的紧接着的增量。假设我们观察行地址），在 4KB 页面中有 64 个可能的位置（偏移）。增量大小总计为 7 位，表示从-64（不包括在内，因为它指向不同的页）到 +63 的值。架构图如下:

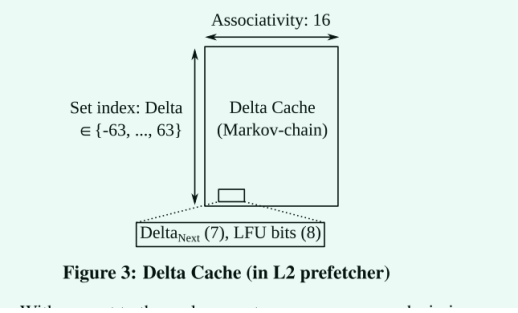

Figure 3: Delta Cache (in L2 prefetcher)

图 1: Pangloss 原理图

但是因为原论文的方法需要用到机器学习，我们选择对这种方法稍作改变，选择使用简单的启发式算法，我们的策略依旧沿用原论文的策略，具体的预取流程如下:

- 在页面缓存中查找该页面

- 如果页面没有找到，退化到 delta=1
- 如果命中了，并且不是来自预取未命中所取的页面，那么重新计算 delta 并且更新我们的 Cache
- 根据占用率即 Cache 的情况更新我们的预取深度
- 寻找最合适的 delta，并且根据这个 delta 进行预取，并更新最好的 delta 以便下一轮使用
- 如果在上面的步骤中发生了页缺失，那么我们需要调用 LFU 算法进行更新 (把频率置为 0 即可)。

在这个计算的过程中要始终保证论文中所提到的 1/3 原则，具体的代码如下 (参考了 Github 上某位印度老哥的代码)

Pangloss 预取

```
unsigned long long int cl_address = addr>>6;
unsigned long long int page = cl_address>>6;
int page_offset = cl_address&63;

int way=-1;
for (int i=0; i<PC_ways_l2c; i++){
    if (PC_ptag_l2c[page%PC_sets_l2c][i]==get_page_tag_l2c(page)){
        way=i;
        break;
    }
}

int cur_delta = 1 + DC_range_l2c/2;
int matched=0;

if ( (way!=-1) && !((type==PREFETCH) && (cache_hit==0)) ){

    int ldelta_l2c=PC_ldelta_l2c[page%PC_sets_l2c][way];
    int loff_l2c=PC_loffset_l2c[page%PC_sets_l2c][way];

    cur_delta = page_offset-loff_l2c + DC_range_l2c/2;
    matched=1;

    update_DC_l2c(ldelta_l2c, cur_delta);
}


int next_delta=cur_delta;
uint64_t addr_n=addr;
int count=0;

int degree = (MSHR.SIZE-MSHR.occupancy)*2/3;
if ((type==PREFETCH) && (cache_hit==0)) degree/=2;
if (NUM_CPUS>1) degree/=4;
```

```
35
36 for (int i_=0; i_<degree && count<degree; i_++){
37
38     int best_delta = get_next_best_transition_l2c(next_delta);
39
40     if (best_delta==-1) break;
41     {
42         int sum=0;
43         for (int j=0; j<DC_ways_l2c; j++){
44             sum+=DC_LFUbits_l2c[next_delta][j];
45         }
46
47         int used[DC_ways_l2c] = {0};
48         for (int i=0; i<2; i++){
49             int max_way = -1;
50             int max_value =  -1;
51             for (int j=0; j<DC_ways_l2c; j++){
52                 if((DC_LFUbits_l2c[next_delta][j]>max_value) && (!used[j])){
53                     max_way = j;
54                     max_value = DC_LFUbits_l2c[next_delta][j];
55                 }
56             }
57             if (max_way == -1) continue;
58
59             if((count<degree) && ( (float)DC_LFUbits_l2c[next_delta][max_way
                   ]/sum > 1/3.0)){
60                 used[max_way]=1;
61                 uint64_t pf_addr = ((addr_n>>LOG2_BLOCK_SIZE)+(
                       DC_deltanext_l2c[next_delta][max_way]-DC_range_l2c/2)) <<
                        LOG2_BLOCK_SIZE;
62                 unsigned long long int pf_page = pf_addr>>12;
63
64                 if (page==pf_page){
65                     prefetch_line(ip, addr, pf_addr, FILL_L2, 0);
66                     count++;
67                 }
68             }
69         }
70     }
71
72     next_delta = best_delta;
73     uint64_t pf_addr = ((addr_n>>LOG2_BLOCK_SIZE)+(best_delta-DC_range_l2c/2)
           ) << LOG2_BLOCK_SIZE;
74     addr_n = pf_addr;
75 }
```

```
76
77  if (way==-1) {
78
79      for (int i=0; i<PC_ways_l2c; i++){
80          if (PC_NRUbit_l2c[page%PC_sets_l2c][i]==0){
81              way=i;
82              break;
83          }
84      }
85
86      if (way==-1){
87          way=0;
88          for (int i=0; i<PC_ways_l2c; i++)
89              PC_NRUbit_l2c[page%PC_sets_l2c][i]=0;
90      }
91  }
92
93  if (matched)
94      PC_ldelta_l2c[page%PC_sets_l2c][way]=cur_delta;
95  else
96      PC_ldelta_l2c[page%PC_sets_l2c][way]=0;
97
98  PC_loffset_l2c[page%PC_sets_l2c][way]=page_offset;
99  PC_ptag_l2c[page%PC_sets_l2c][way]=get_page_tag_l2c(page);
100 PC_NRUbit_l2c[page%PC_sets_l2c][way]=1;
101
102 return metadata_in;
```

三、 替换算法的实现

(一) LFU

LFU（Least Frequently Used）替换策略是一种缓存替换策略，它根据页面被访问的频率来进行替换，其核心思想是“最近最少使用”，即在所有页面中，访问次数最少的页面将被替换掉。这样就可以保证缓存中存放的页面都是最近使用频率最高的页面，从而提高查找效率。

1. LFU 的有序实现 & 无序实现

这两种实现方式都是为了实现最近最少使用的替换策略，当然，我们在算法具体的执行过程中，主要会涉及到一下两个步骤：

- 找到需要替换的，也就是访问频率最小的
- 当访问元素时，更新对应元素的频率

显然，对于有序实现来说，前者的代价为 $O(1)$，而后者的代价为 $O(N)$,N 为我们维护的数据总量，而对于无序的实现，前者的代价就是 $O(N)$，而后者的代价为 $O(1)$，那么这两种实现分别适用于反复替换以及多次成功访问的场景。

对于无序的实现来说，我们的算法非常简单，只需要维护一个全局的数组用来维护频率，通过遍历找到最小的项进行替换，而更新时直接 $O(1)$ 访问即可，如下:

无序实现

```
    int index = 0;
    int min_freq = freq[set][0];
    for (int i = 1; i < LLC_WAY; i++)
    {
        if (freq[set][i] < min_freq)
        {
            min_freq = freq[set][i];
            index = i;
        }
    }
    return index;
    .....

    if (hit)
        freq[set][way] += 1;
    else
        freq[set][way] = (type != PREFETCH);
```

对于有序实现，我们需要维护一个以频率为排序依据的有序的全局管理链表，然后在每次频率更新时更新链表的排序，核心代码如下:

有序实现的数据结构

```
class Node
{
public:
    PPU data;
    Node *prev, *next;
    void init_list() { next = prev = this; };
    void add_after(Node* nxt) { nxt->next = next; next->prev = nxt; next =
        nxt; nxt->prev = this; };
    void add_before(Node* pre) { pre->next = this; pre->prev = prev; prev->
        next = pre; prev = pre;};
    void del_after() { next = next->next; next->prev = this; };
    void del() { prev->next = next; next->prev = prev; };
    bool operator < (Node b) const { return Freq(data) <= Freq(b.data); };
    ~Node() {};
};
void swap_node(Node* pre, Node* nxt) { pre->del_after(); pre->add_before(nxt)
    ;}
}
```

这个我们可以发现就是一个链表而已，不过排序函数可以说是暗藏玄机，因为我们 LFU 在处理访问频率相同的元素时，会把之前最早的那一个元素替换掉，所以如果是老旧的数据，他即使频率和我们当前的相等，依然按照小于处理，之后是我们实现数组顺序改变的函数。这里同样

有一个小数据的优化，就是如果大家都是只访问 1 次这样，那么我们可以特判一下，避免重复大规模的遍历。

改序

```
void move_forward(Node* h, Node* ele)
{
    while ((*(ele->next)) < (*ele) && ele->next != h) {
        swap_node(ele, ele->next);
    }
}

void CACHE::llc_update_replacement_state(uint32_t cpu, uint32_t set, uint32_t
     way, uint64_t full_addr, uint64_t ip, uint64_t victim_addr, uint32_t
    type, uint8_t hit)
{
    ......
    if (hit)
    {
        Freq((freq_head[set][way + 1].data)) ++;
        move_forward(&freq_head[set][0], &freq_head[set][way + 1]);
    }
    else
    {
        Freq((freq_head[set][way + 1].data)) = (type != PREFETCH);
        if (Freq((freq_head[set][way].data)) > Freq((freq_head[set][way + 1].
            data))) {
            freq_head[set][way + 1].del();
            freq_head[set]->add_after(&freq_head[set][way + 1]);
            move_forward(&freq_head[set][0], &freq_head[set][way + 1]);
        }
        else move_forward(&freq_head[set][0], &freq_head[set][way + 1]);
    }
}
```

这样大费周折的处理，最后只是为了我们在寻找目标位置时可以在 $O(1)$ 的复杂度完成，核心代码如下

替换位置

```
uint32_t CACHE::llc_find_victim(uint32_t cpu, uint64_t instr_id, uint32_t set
    , const BLOCK *current_set, uint64_t ip, uint64_t full_addr, uint32_t
    type)
{
    ui idx = freq_head[set][0].next->data.first.second;
    return idx;
}
```

2. window-LFU

这个朴素的 LFU 算法主要是用来解决如下两个问题:

- 对于每个缓存项，LFU 都需要记录其访问次数，需要不小的额外内存开销；
- 近期不再访问的历史数据无法清理，导致缓存污染。

那么针对这种问题。window-LFU 应运而生，Window 是用来描述算法保存的历史请求数量的窗口大小的。Window-LFU 并不记录所有数据的访问历史，而只是记录过去一段时间内的访问历史。即当请求次数达到或超过 window 的大小后，每次有一条新的请求到来，都会清理掉最旧的一条请求，以保证算法记录的请求次数不超过 window 的大小。

比如说如下场景，window = 9，现在有 9 条缓存访问记录，按时间顺序从先到后分别是：A,B,C,D,A,B,C,D,A，即缓存队列。

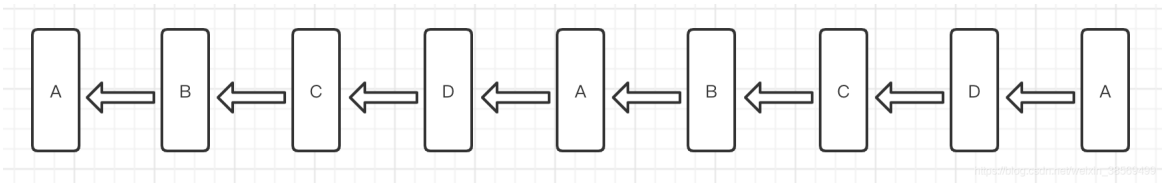

图 2: 具体场景介绍

那么如果此时新来一个访问记录 B，那么，我们就需要把第一个记录删去，把新的记录 B 加入到我们的缓存队列中，即

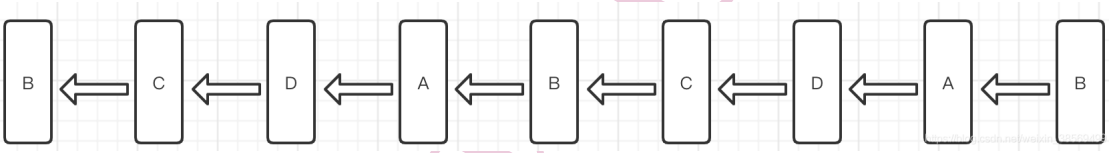

图 3: 具体场景介绍

其余部分和 LFU 差别并不大，我们的核心代码如下，首先是全局变量，我们主要维护一个窗口和全局的一个映射关系，初始化为 0 即可：

全局变量的初始化

```
#define WND_SIZE 25

std::vector<vector<int>> wnds;
int hash_map[LLC_SET][LLC_WAY];

// initialize replacement state
void CACHE::llc_initialize_replacement()
{
    cout << "Initialize LFU state" << endl;
    for (int i = 0; i < LLC_SET; i++)
    {
        wnds.push_back({});
    }
    memset(hash_map, 0, sizeof hash_map);
}
```

之后在具体的替换时，我们去寻找最小的频率即可

全局变量的初始化

```
uint32_t CACHE::llc_find_victim(uint32_t cpu, uint64_t instr_id, uint32_t set
    , const BLOCK *current_set, uint64_t ip, uint64_t full_addr, uint32_t
    type)
{
    int index = 0;
    int min_freq = hash_map[set][0];
    for (int i = 1; i < LLC_WAY; i++)
    {
        if (hash_map[set][i] < min_freq)
        {
            min_freq = hash_map[set][i];
            index = i;
        }
    }
    return index;
}
```

在具体的更新时，我们不仅要更新频率信息，更重要的是需要根据窗口当前的状态更新一下历史信息，代码如下

```
    if (wnds[set].size() >= WND_SIZE)
    {
        hash_map[set][wnds[set][0]] --;
        wnds[set].erase(wnds[set].begin());
    }
    if (hit || type != PREFETCH)
    {
        hash_map[set][way] ++;
        wnds[set].push_back(way);
    }
```

具体的代码细节在压缩包的文件中。

四、 性能评测

(一) 运行结果

我们分别利用不同的策略组合对数据集进行评估测量，最终数据集的分数如下

策略组合	数据集1	数据集2	Mean
LFU-sorted+GHB_PC/DC	0.87732	1.04189	0.959605
LFU-sorted+GHB_PC/CS	0.87729	0.94387	0.91058
LFU-sorted+Pangloss	0.87606	1.08657	0.981315
LFU-unsorted+GHB_PC/DC	0.89785	1.22277	1.06031
LFU-unsorted+GHB_PC/CS	0.8979	1.19066	1.04428
LFU-unsorted+Pangloss	0.89722	1.24278	1.07
LFU-Window+GHB_PC/DC	0.90955	1.21469	1.06212
LFU-Window+GHB_PC/CS	0.90966	1.17110	1.040385
LFU-Window+Pangloss	0.91916	1.23319	1.07615

图 4: 运行结果

(二) 总结

先说替换算法，我们根据上面的结果很容易发现，我们非常麻烦的实现了一个有序的 LFU，但是反而他的效果是最差的，这说明我们一般的使用场景中访存的次数要多于替换的次数 (这也很符合我们的直觉)，并且进入我们 Cache 中的内容会被经常访问，不过这一点至少证明我们的预取算法都是有效的:(。另外我们也可以发现，加了 Windows 的 LFU 效果确实优于其他两个，可能数据集中确实有很多前期高频访问，后期访问比较少的情况。

针对预取，我们也可以发现，GHB_PC_DC 的效果更好，主要原因是我的代码综合了一部分 GHB_PC_CS 的特点，提高了预取中匹配成功的次数，而 Pangloss，则略好于 GHB 的两种算法。我们细细思索就可以发现，预取的时候，我们条件定的越强，越难满足，反而会错失很多预取的良机，反而是条件宽松，通过一些基本的数据特性，比如 delta 值，会使我们预取的功效大大增强。

那么综合以上，我们可以得出这么几个结论，首先，预取算法的条件不宜太过具体、严格，要宽松，另外一方面，替换算法要尽可能避免长时间垃圾信息的干扰。

参考文献

[1] Philippos Papaphilippou, Paul HJ Kelly, and Wayne Luk. Pangloss: a novel markov chain prefetcher. *arXiv preprint arXiv:1906.00877*, 2019.