体系结构大作业 - cache 替换策略和数据预取

- 姓名：管昀玫
- 学号：2013750
- 专业：计算机科学与技术

Prefetcher

IP stride预取

基于指令指针IP的步进预取器（IP-based stride prefetcher）。这个预取器跟踪单条读存指令。如果侦测到读操作有固定的有规律的步长，则以当前地址加上步长为源地址预取下一个数据行。预取器可以侦测前向（地址升序）或者后向（地址降序）的访存模式，可以侦测的步长最长可达2K字节。

基于指令指针（IP）的预取器，如IP stride预取器提供了巨大的性能优势，有助于减少昂贵的DRAM访问数量。IP stride预取器为一个IP学习一个循环的步长，一旦它对学习的步长有了信心，它就开始用学习的步长进行预取。使用基于IP的预取器的优点之一是，它不容易受到不同内存区域的IP重排和随机访问的影响，这可能会混淆非IP预取器。

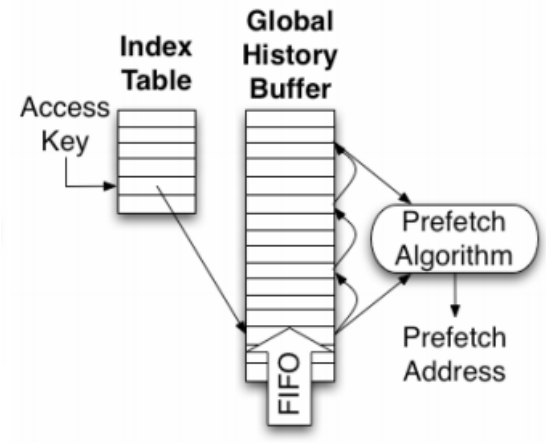
在本次实验中，我将使用给定的ip stride L2预取器作为baseline，测试其他预取器的性能。

GHB步长预取

目标是通过函数在 L2 缓存上实现一个基于全局历史缓冲区(Global History Buffer)的步长预取器。 预取器记录 L1 数据缓存中丢失的负载的地址的访问模式，并将预测的数据预取到 L2 缓存中。

基于GHB的步长预取算法需要维护两个数据结构：

- 索引表(Index Table，IT)：一个通过用key(例如程序计数器 PC)进行索引的表，它存储指向 GHB 条目的指针。
- 全局历史缓冲区(GHB)：一个循环队列，有n个条目的FIFO表，存储n个最近的 L2 缺失地址。每个 GHB 条目存储一个指针(这里称为 prev_ptr)，该指针指向具有相同 IT索引的最后一个缓存行地址。通过遍历 prev_ptr，可以获得指向同一 IT 条目的缓存行地址的时间序列。



数据结构的建立如下所示：

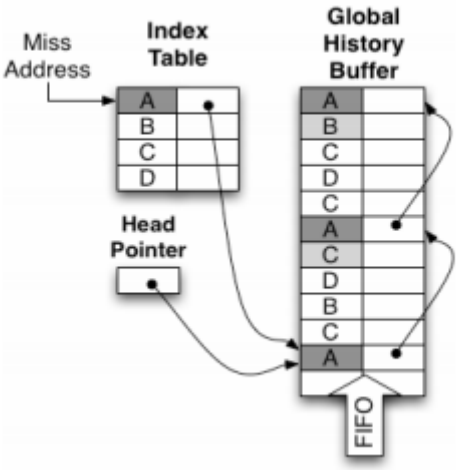
```
typedef struct GHB_t {  //全局历史缓冲区
    uint64_t cl        = 0;
    int16_t prev_entry = 0;
} GHB;

typedef struct It_t {   //索引表
    int16_t ghb_entry = 0;
} IT;

//二者的大小都设置为256
GHB gh_buffer[256];
IT index_table[256];
```

> 为了实现Stride Prefetching（PC/CS），GHB结构以**恒定的stride**检测加载指令。使用LOAD指令的PC作为索引表的索引，创建的地址列表是给定**PC的地址序列**。负载的跨度可以通过计算地址列表中**连续条目之间的差异**。如果检测到一个恒定的跨度，例如，如果前*x*个计算的跨度相同，那么地址*a+s, a+2s, ... a+ds*被预取，其中*a*是当前错过的address，*s*是检测到的跨度，*d*是预取degree。

该算法使用 PC 和 GHB 条目中的链接指针，检索访问二级缓存对应条目中的最后 3 个地址。步长是通过取序列中 3 个连续地址之间的差来计算的。

对于每个新的miss，GHB以FIFO方式更新。miss的地址被放入由头部指针指向的GHB entry中，其link entry被赋予Index中的当前值。然后，索引表的link entry被更新为头部指针，它指向新添加的entry。最后，头部指针被递增以指向下一个GHB entry。

此部分的实现都在`l2c_prefetcher_operate()`函数中。




```
    // 链接的entry是Index中的当前值
    gh_buffer[GHB_index].prev_entry = index_table[ip_mask].ghb_entry;
    index_table[ip_mask].ghb_entry  = GHB_index;

    uint64_t last_access[3] = {0};
    last_access[0] = gh_buffer[GHB_index].cl;

    // 检索访问二级缓存对应条目中的最后 3 个地址
    if (gh_buffer[GHB_index].prev_entry != GHB_INVALID_ENTRY) {
        last_access[1] = gh_buffer[gh_buffer[GHB_index].prev_entry].cl;
        if (gh_buffer[gh_buffer[GHB_index].prev_entry].prev_entry !=
GHB_INVALID_ENTRY)
            last_access[2] =
gh_buffer[gh_buffer[gh_buffer[GHB_index].prev_entry].prev_entry].cl;
```

```
        else {
            GHB_index = (GHB_index + 1) % GHB_SIZE;
            return metadata_in;
        }
    } else {
        GHB_index = (GHB_index + 1) % GHB_SIZE;
        return metadata_in;
    }
    // 头部指针递增 指向下一个GHB entry
    GHB_index = (GHB_index + 1) % GHB_SIZE;
```

步长是通过取序列中 3 个连续地址之间的差来计算的。如果两个步长相等(步长为 d)，预取器只向缓存行 A+ld、A+(l+1)d、A+(l+2)d、...、A+(l+n)d 发出预取请求，其中 l 是事先设定好的预取 look-ahead，n 是度。**此处将l设置为4，将n也设置为4。**

```
// 跨度：计算地址列表中连续条目之间的差异
    int64_t previous_stride = 0;
    for (size_t i = 0; i < 2; i++) {
        int64_t stride = 0;
        if ((last_access[i] - last_access[i + 1]) > 0)
            stride = last_access[i] - last_access[i + 1];
        else {
            stride = last_access[i + 1] - last_access[i];
            stride *= -1;
        }
        // 如果两个步长相等(步长为 d)
        if (previous_stride == stride) {
            // 预取 line A + ld
            for (size_t i = 0; i < PREFETCH_DEGREE; i++) {
                uint64_t pf_address = (cl_addr + PREFETCH_LOOKAHEAD * (stride *
(i + 1))) << LOG2_BLOCK_SIZE;

                // prefetch的地址与当前需求地址必须在同一page(4KB)内
                if ((pf_address >> LOG2_PAGE_SIZE) != (addr >> LOG2_PAGE_SIZE))
                    break;

                prefetch_line(ip, addr, pf_address, FILL_L2, 0);
            }
        }
        previous_stride = stride;
    }
```

与传统的表预取方法相比，用全局历史缓冲区进行预取有明显的优势。

1. 首先，使用**FIFO**历史缓冲区可以通过消除表中的陈旧数据来提高相关预取的准确性。
2. 索引表和GHB表的大小是分开的，索引表只需要大到足以容纳预取key的working set。 GHB会更大一些，但它大小独立，足以容纳错过的地址流的代表性部分。
3. 全局历史缓冲区包含了一个更完整的缓存缺失历史，为设计更有效的预取方法创造了机会。

马尔科夫预取

马尔科夫模型在预取中被广泛地应用，它为后续访问估计和利用地址转移概率。距离预取是常见的Markov模型预取器的概括，它使用deltas而不是地址来建立更通用的模型。

Delta转换的马尔科夫链是一个有向图，Delta作为状态/节点，概率作为加权转换/弧。delta是两个连续地址之间的差，如下图所示：

```
Address:      1    4    2    7    8    9
Delta:          3   -2    5    1    1
```

马尔科夫预取法将miss的地址流建模为一个马尔科夫图——也可以说是概率状态机。马尔可夫图中的每个节点都是一个地址，节点之间的弧被标记为该弧的源节点地址将紧随目标节点地址的概率。相关表的每个条目代表相关马尔可夫图中的一个节点，其内容代表具有最高概率的弧。

在实际应用中由于有page的限制，需要限制deltas的范围。预取时会丢弃超过page限制的预测地址。

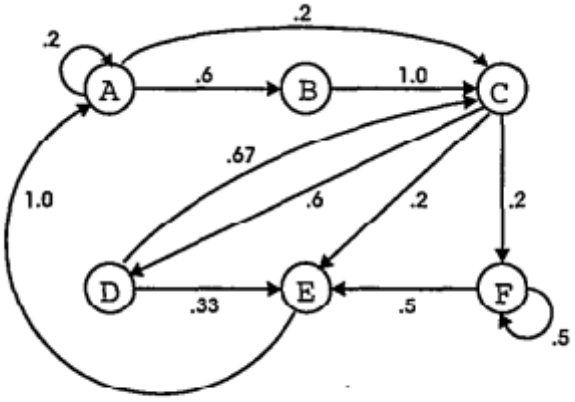


Figure 3: Markov model representing the previous reference string *via* transition probabilities.

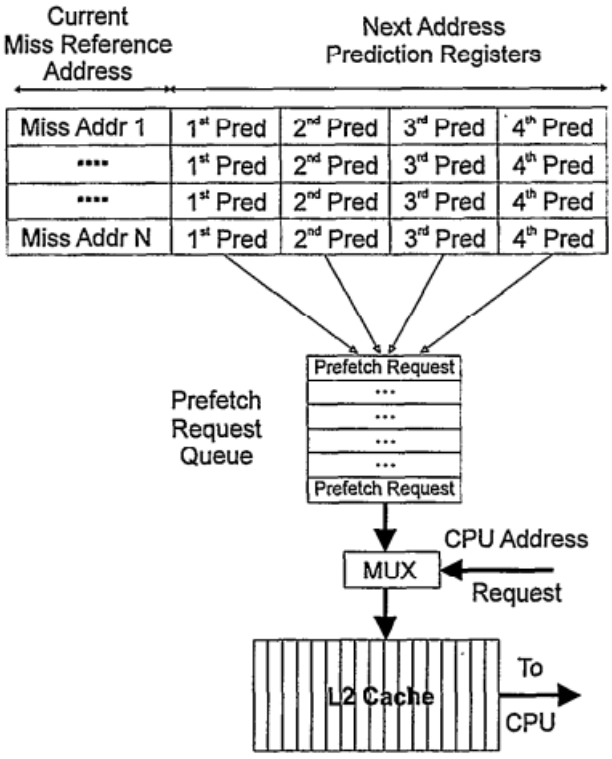


Figure 4: Hardware Used to Approximate Markov Prediction Prefetcher

实现马尔科夫链需要$N * N$大小的矩阵(N为状态的数量)，用以维护邻接矩阵的过度概率。但我的最终目的是构建pangloss预取器，因此将全局表改造为pangloss也使用的两个数据结构：Delta Cache和Page Cache，这两个结构将在Pangloss部分详细说明，此处直接展示代码：

```
// Delta Cache: 代表马尔科夫链
#define DELTA_NUM_SETS 128 // -64 到 +63 possible offsets
#define DELTA_NUM_WAYS 16

delta_t DC_delta_next[DELTA_NUM_SETS][DELTA_NUM_WAYS];      // D_next
uint64_t DC_counter[DELTA_NUM_SETS][DELTA_NUM_WAYS];        // DC counter
```

```
// Page Cache: 跟踪同一页的上一个addr/deltas
#define PAGE_NUM_SETS 256   // 256 sets
#define PAGE_NUM_WAYS 12    // 12 ways
#define PAGE_TAG_BITS 10    // 10-bit page tags

// 这里用的是64bit，pangloss直接用int型
delta_t PC_delta_prev[PAGE_NUM_SETS][PAGE_NUM_WAYS];        // D_prev
uint64_t PC_offset_prev[PAGE_NUM_SETS][PAGE_NUM_WAYS];      // O_prev
uint64_t PC_ptags[PAGE_NUM_SETS][PAGE_NUM_WAYS];            // Page tag
uint64_t PC_LRUbit[PAGE_NUM_SETS][PAGE_NUM_WAYS];           // LRU bit
```

值得注意的是，更新Delta Cache使用的是LRU策略，而更新Page Cache使用的是NRU策略。

在每一次L2查找中，prefetcher执行以下步骤：

1. 从页面缓存中查找当前页地址的前一个 delta (D_{prev}) 和 page offset (O_{prev})
2. 对于每一次page cache hit，更新delta和page cache ($O_{curr} - O_{prev}$)
   对于每一次page cache miss，存储当前的 delta 和 offset。为了防止预测错误，delta cache不会更新
3. 遍历马尔科夫链(delta cache)，从当前的delta预测下一个地址，直到达到prefetch degree

Markov与pangloss不同的是：Markov需要所有可能的路径中找到最佳路径，将会产生巨大开销，这一点将在pangloss中优化。其`l2c_prefetcher_operate()`代码如下所示：

```
uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t
cache_hit, uint8_t type, uint32_t metadata_in)
{
    uint64_t cl_addr = addr >> LOG2_BLOCK_SIZE;
    uint64_t page_addr = addr >> LOG2_PAGE_SIZE;
    uint64_t curr_page_offset = cl_addr & BLOCK_OFFSET_MASK;

    // 在Page Cache中查找同一页 上一条entry
    uint64_t pc_set = page_addr % PAGE_NUM_SETS;
    int pc_way = -1;
    uint64_t pc_tag = page_addr & ((1 << PAGE_TAG_BITS) - 1);
    for (int i = 0; i < PAGE_NUM_WAYS; i++) {
        if (PC_ptags[pc_set][i] == pc_tag) {
            pc_way = i;
            break;
        }
    }
    delta_t curr_delta = norm_delta(1); // 无PC hit, curr_delta置1
    // page cache hit和cache miss时 更新delta cache
    if ((pc_way != -1) && (cache_hit == 0)) {
        delta_t prev_delta = PC_delta_prev[pc_set][pc_way];
        uint64_t prev_page_offset = PC_offset_prev[pc_set][pc_way];
        // 更新delta ($O_{curr}-O_{prev}$)
        curr_delta = norm_delta(curr_page_offset - prev_page_offset);
        // 更新delta cache
        update_delta_cache(prev_delta, curr_delta);
        // 更新page cache
        PC_delta_prev[pc_set][pc_way] = curr_delta;
    }
    // page cache miss: 找到page cache替换块
    if (pc_way == -1) {
        // NRU策略
```

```
        for (int i = 0; i < PAGE_NUM_WAYS; i++) {
            if (PC_LRUbit[pc_set][i] == 0) {
                pc_way = i;
                break;
            }
            // 全是1，则替换第一块，并且全部翻转为0
            if (pc_way == -1) {
                pc_way = 0;
                for (int i = 0; i < PAGE_NUM_WAYS; i++) {
                    PC_LRUbit[pc_set][i] = 0;
                }
            }
        }
    }
    // 更新Page Cache
    PC_ptags[pc_set][pc_way] = pc_tag;
    PC_offset_prev[pc_set][pc_way] = curr_page_offset;
    PC_LRUbit[pc_set][pc_way] = 1;

    // 动态调整degree
    int degree = (MSHR.SIZE-MSHR.occupancy)*2/3;
    if ((type==PREFETCH) && (cache_hit==0)) degree/=2;

    int num_pre_issued = 0;
    // 从当前delta开始遍历马尔科夫链
    uint64_t next_delta = curr_delta;
    uint64_t next_addr = cl_addr;
    for (int j = 0; j < degree && num_pre_issued <= degree; j++) {
        // 计算LRU counter的总数 以便计算概率
        float sum = 0;
        for (int i = 0; i < DELTA_NUM_WAYS; i++) {
            sum += DC_counter[next_delta][i];
        }
        //找到最佳概率的delta
        float best_prop = 0;
        for (int i = 0; i < DELTA_NUM_WAYS; i++) {
            float prop = DC_counter[next_delta][i] / sum;
            // 概率大于1/3 才有可能预取
            if (prop >= (1/3.0)) {
                // 计算预取地址
                uint64_t pf_addr =
                    (next_addr + denorm_delta(DC_delta_next[next_delta][i]))
                    << LOG2_BLOCK_SIZE;
                uint64_t pf_page_addr = pf_addr >> LOG2_PAGE_SIZE;
                // 在page范围限制之内，则可成功预取
                if (page_addr == pf_page_addr) {
                    prefetch_line(ip, addr, pf_addr, FILL_L2, 0);
                    num_pre_issued++;
                }
                // 更新best_prop
                if (prop > best_prop) {
                    best_prop = prop;
                    next_delta = DC_delta_next[next_delta][i];
                    next_addr = pf_addr >> LOG2_BLOCK_SIZE;
                }
            }
        }
        // 概率都小于1/3，则丢弃
```

```
            if (best_prop == 0) {
                break;
            }
        }
        return metadata_in;
    }
```

Pangloss预取

由于实现马尔科夫链需要$N * N$大小的矩阵(N为状态的数量)，用以维护邻接矩阵的过度概率，会导致很高的稀疏性，现有的方法是使用关联结构进行近似，通常采用最近LRU或FIFO的替换策略，但这两种策略都容易因抖动而失去对重要转换的跟踪。

Pangloss 也是一种马尔可夫预取器，它使用地址delta来预测复杂的访问模式。使用初始地址和deltas，预取器能够重建地址流。delta同样受page边界限制，因为页面分配不是连续的。因此，马尔可夫模型记录每页的deltas而不是全局的delta。

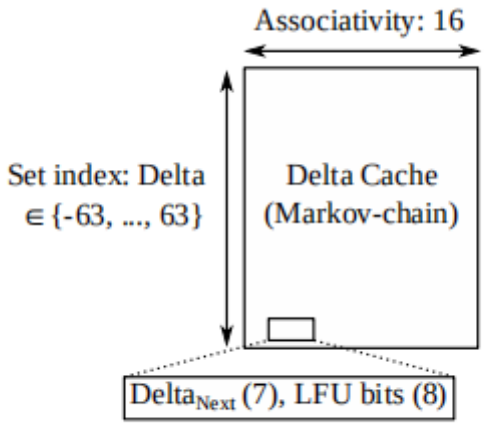


Figure 3: Delta Cache (in L2 prefetcher)

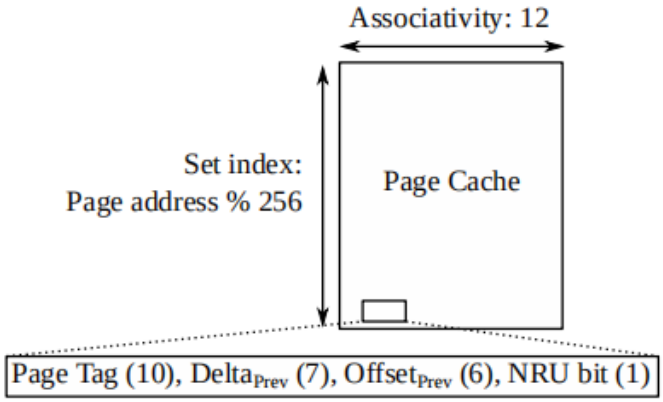


Figure 4: Page Cache (in L2 prefetcher)

Delta Cache

Delta Cache是以当前delta为索引的，每一组保存最频繁的下一个delta。二级缓存使用的是cache line地址（64位），所以在一个4KB的页面中有64个可能的偏移，即所有的delta在`(-64, +63]`之内(7 bits)。

关于替换，使用最不经常使用（LFU）策略，目的是为了保证正确的transition概率，也给被淘汰的delta提供快速丢弃的机会。此外，一个集合中的每一个块都包含下一个delta和一个counter，counter在每次cache hit时都会递增；当出现溢出时，相应集合中的所有块的counter都会减半。通过这种方式，可以保留了几乎相同的计数比例，降低的精度有利于更高的数值。为了找到过渡概率，我们将数值除以集合中所有数值的总和，这也可以逐步计算。保持比例在替换和优先预取中都很重要。

```
// Delta Cache
#define DC_range (128)  // (-64, 63] index: delta
#define DC_ways 16      // associality: 16
#define DC_LFU_max 256   // FLU counter最大值

int DC_delta_next[DC_range][DC_ways]; // D_next
int DC_LFUbits[DC_range][DC_ways];   // LFU bit
```

Page Cache

Page Cache旨在帮助重构意外或临时的页面转换所导致的混乱的delta转换。修改算法，以保持每页最后一个address/delta，从而使产生有效过渡的概率最大化。

如Figure 4，Page Cache是路组关联的，以页地址为索引，每块有4个字段：

- Page tag: 标识区分页面。
- D_{prev}: 前一个delta，使用它来找到过渡。长度为7 bits。初始值为-64，表明没有前一个delta。
- O_{prev}: 前一个地址偏移量。用来计算基于新地址的当前delta，共6 bits（从0到63）。
- NRU位：用于近似LRU替换策略，1 bit，驱逐最近未使用（NRU）块。

```
// Page Cache
#define PC_sets 256     // 256 sets
#define PC_ways 12      // 12 ways
#define PC_tags 10  // 10-bit page tags

int PC_delta_prev[PC_sets][PC_ways];    // D_prev
int PC_offset_prev[PC_sets][PC_ways];   // O_prev
int PC_ptags[PC_sets][PC_ways];     // Page tag
int PC_NRUbit[PC_sets][PC_ways];    // NRU bit
```

在每一次L2查找中，prefetcher执行以下步骤：

1. 从页面缓存中查找当前页地址的前一个 delta (D_{prev}) 和 page offset (O_{prev})
2. 对于每一次page cache hit，更新delta和page cache ($O_{curr} - O_{prev}$)
   对于每一次page cache miss，存储当前的 delta 和 offset。为了防止预测错误，delta cache不会更新
3. 遍历马尔科夫链(delta cache)，从当前的delta预测下一个地址，直到达到prefetch degree

如果prefetcher必须在所有可能的路径中找到最佳路径，将会产生巨大开销。因此使用论文中提到的方法：以大于1/3的概率预取child delta中出现的地址，并在下一次迭代中使用最高概率的delta，直到达到prefetch degree。只有当地址位于同一page时，才会发出预取（不能超出page范围）。如果产生的预取地址超出了当前页面，它将被丢弃，但路径仍然有效。这样做是为了保留对同一页的后续访问。

`l2c_prefetcher_operate()` 函数如下：

```
uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t
cache_hit, uint8_t type, uint32_t metadata_in)
{
    unsigned long long int cl_address = addr>>6;
    unsigned long long int page = cl_address>>6;
    int page_offset = cl_address&63;

    // 在Page Cache中查找同一页 上一条entry
    int way=-1;
    for (int i=0; i<PC_ways; i++){
        if (PC_ptags[page%PC_sets][i]==get_page_tag(page)){
```

```
            way=i;
            break;
        }
    }
    int cur_delta = 1 + DC_range/2;  // 没找到则 delta = 1
    int matched=0;
    // Page Cache hit情况，且不是prefetch miss
    if ( (way!=-1) && !((type==PREFETCH) && (cache_hit==0)) ){
        int ldelta_l2c=PC_delta_prev[page%PC_sets][way];
        int loff_l2c=PC_offset_prev[page%PC_sets][way];
        // 计算当前delta
        cur_delta = page_offset-loff_l2c + DC_range/2;
        matched=1;
        // 使用新的delta transition更新Delta Cache
        updata_Delta_Cache(ldelta_l2c, cur_delta);
    }
    int next_delta=cur_delta;
    uint64_t addr_n=addr;
    int count=0;
    // 动态调整degree
    int degree = (MSHR.SIZE-MSHR.occupancy)*2/3;
    if ((type==PREFETCH) && (cache_hit==0)) degree/=2;
    if (NUM_CPUS>1) degree/=4;  //本次实验就用一个核，所以这句没什么用
    for (int i_=0; i_<degree && count<degree; i_++){
        // 基于当前delta，找到下一个概率最大的delta
        int best_delta = get_next_best_transition(next_delta);
        // 没找到的情况(概率<1/3)
        if (best_delta==-1) break;
        // 计算当前set中LFU counter的总次数
        int sum=0;
        for (int j=0; j<DC_ways; j++){
            sum+=DC_LFUbits[next_delta][j];
        }
        // 查找最有可能的top2 child deltas
        int used[DC_ways] = {0};
        for (int i=0; i<2; i++){
            int max_way = -1;
            int max_value =  -1;
            for (int j=0; j<DC_ways; j++){
                if((DC_LFUbits[next_delta][j]>max_value) && (!used[j])){
                    max_way = j;
                    max_value = DC_LFUbits[next_delta][j];
                }
            }
            if (max_way == -1) continue;
            // 概率>1/3
            if((count<degree) && ( (float)DC_LFUbits[next_delta][max_way]/sum >
1/3.0)){
                used[max_way]=1;
                uint64_t pf_addr = ((addr_n>>LOG2_BLOCK_SIZE)+
(DC_delta_next[next_delta][max_way]-DC_range/2)) << LOG2_BLOCK_SIZE;
                unsigned long long int pf_page = pf_addr>>12;
                // 且在同一个page内，则完成预取
                if (page==pf_page){
                    prefetch_line(ip, addr, pf_addr, FILL_L2, 0);
                    count++;
                }
            }
```

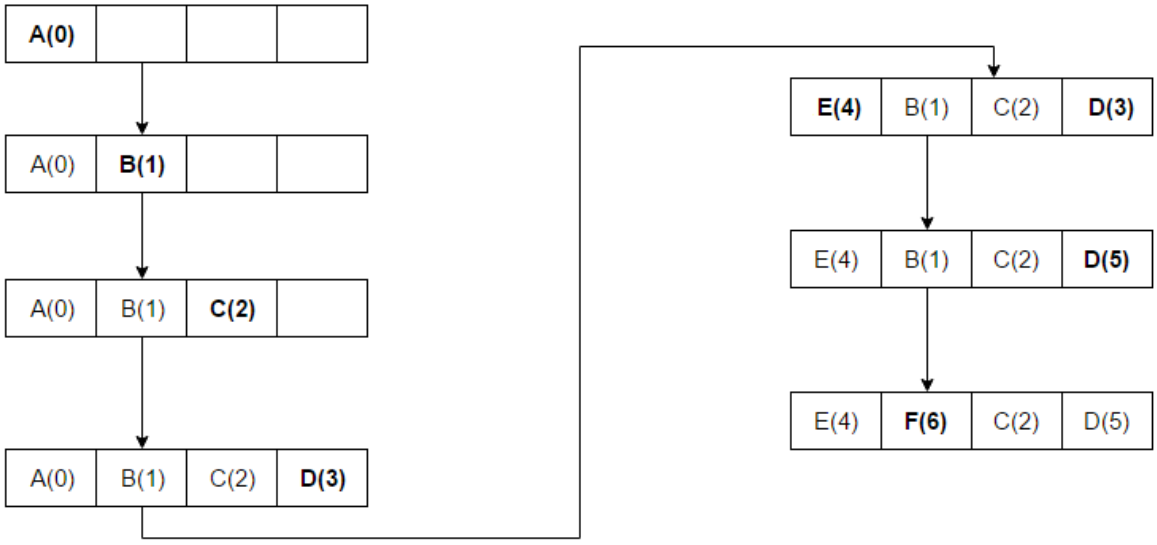
```
            }
            // 根据最高概率delta值更新下一个delta
            next_delta = best_delta;
            uint64_t pf_addr = ((addr_n>>LOG2_BLOCK_SIZE)+(best_delta-DC_range/2))
<< LOG2_BLOCK_SIZE;
            addr_n = pf_addr;
        }
        // miss的情况：替换该块，使用NRU策略
        if (way==-1) {
            // 找到NRU位为0的
            for (int i=0; i<PC_ways; i++){
                if (PC_NRUbit[page%PC_sets][i]==0){
                    way=i;
                    break;
                }
            }
            if (way==-1){ // 若全为1则反转成0
                way=0; // 选择第一块替换
                for (int i=0; i<PC_ways; i++)
                    PC_NRUbit[page%PC_sets][i]=0;
            }
        }
        // hit的情况，更新Page Cache entry
        if (matched)
            PC_delta_prev[page%PC_sets][way]=cur_delta;
        else
            // miss: 没有该entry或delta transition超限的情况，则将delta值置0
            // 0: delta = -64，会跳到另一个页中
            PC_delta_prev[page%PC_sets][way]=0;
        PC_offset_prev[page%PC_sets][way]=page_offset;
        PC_ptags[page%PC_sets][way]=get_page_tag(page);
        PC_NRUbit[page%PC_sets][way]=1;

        return metadata_in;
}
```

Cache Replacement Policies

LRU: Least recently used

这个缓存算法将最近使用的条目存放到靠近缓存顶部的位置。当一个新条目被访问时，LRU将它放置到缓存的顶部。当缓存达到极限时，较早之前访问的条目将从缓存底部开始被移除。这里会使用到昂贵的算法，而且它需要记录“年龄位”来精确显示条目是何时被访问的。此外，当一个LRU缓存算法删除某个条目后，“年龄位”将随其他条目发生改变。

例如访问 A B C D E D F:



在这个例子中，一旦 A B C D 被安装在有序号的块中(每个新的 Access 增量1)，当访问 E 时，出现 miss，需要安装在其中一个块中。根据 LRU 算法，由于 A 具有最低的Rank(A (0))，E 将取代 A。在第二个到最后一个步骤中，访问 D，因此更新序列号。最后，F 被访问，取代了目前最低Rank(B (1))的 B。

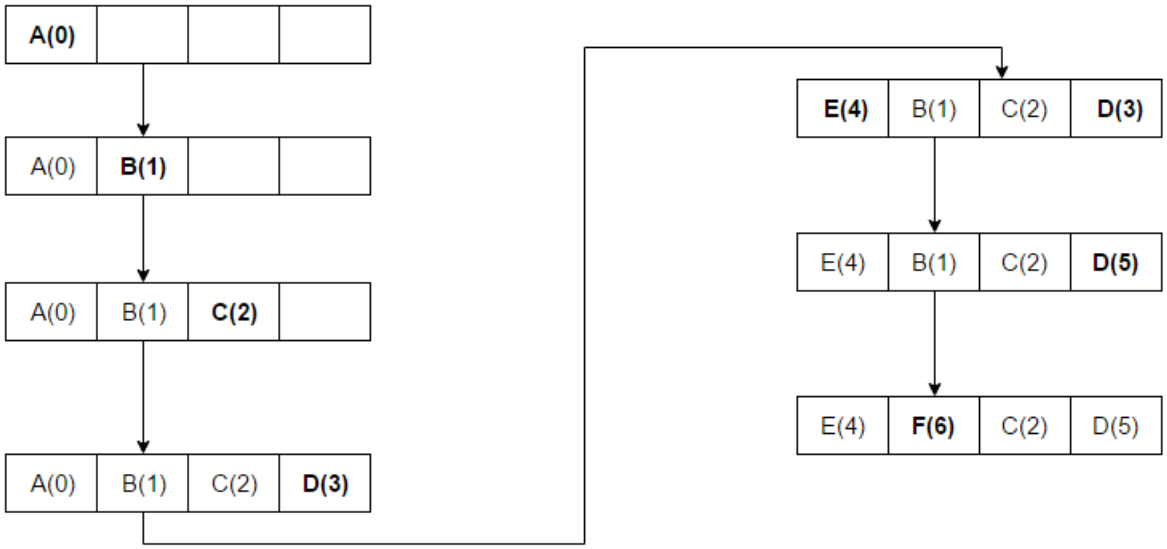
使用LRU作为baseline，来测试其他cache替换策略的性能。

LFU: Least frequently used

LRU淘汰最长时间没有被使用的页面，LFU淘汰一段时间内使用次数最少的页。

这个缓存算法使用一个计数器来记录条目被访问的频率。通过使用LFU缓存算法，最低访问数的条目首先被移除。这个方法并不经常使用，因为它无法对一个拥有最初高访问率之后长时间没有被访问的条目缓存负责。这种技术的一般实现需要为缓存线保留 "年龄位"，并根据年龄位跟踪 "最近使用最少的 "缓存线。在这样的实现中，每当一个缓存线被使用时，所有其他的缓存线的年龄都会发生变化。

例如访问A B C D E D F:



在这个例子中，一旦A B C D被安装在序列号的区块中（每一个新的访问都增加1），当E被访问时，发生miss，它需要被安装在一个区块中。根据LRU算法，由于Rank(A(0))最低，E将取代A。

在倒数第二步，D被访问，因此序列号被更新。

最后，F被访问，取代了目前Rank(B(1))最小的B的位置。

```
// initialize replacement state
void CACHE::llc_initialize_replacement()
```

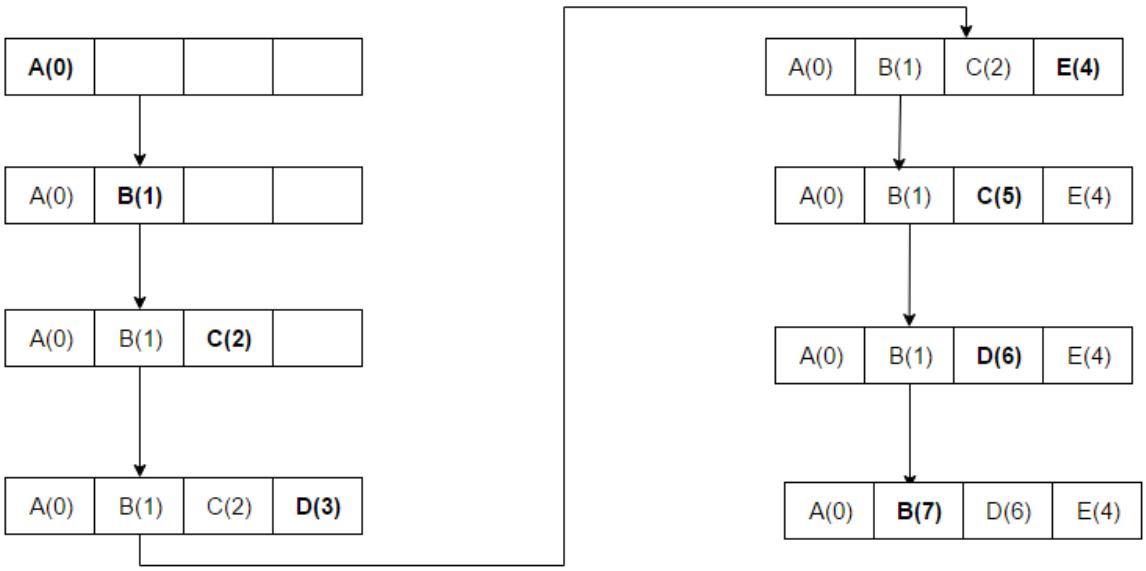
```
{
    for (int i=0; i<LLC_SET; i++) {
        for (int j=0; j<LLC_WAY; j++) {
            hits_[i][j] = 0;
        }
    }
    miss_counter = 0;
    time_counter = 0;
}

// find replacement victim
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)
{

    uint32_t new_way = -1;
    uint64_t new_time = hits_[set][0];
    // fill invalid line first
    uint32_t way = 0;
    for (way=0; way<NUM_WAY; way++) {
        if (block[set][way].valid == false) {

            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " invalid set: " << set << " way: " << way;
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][way].address << " data: " << block[set]
[way].data;
            cout << dec << " lru: " << block[set][way].lru << endl; });
            new_way = way;
            break;
        }
    }
    if(way == NUM_WAY){
        for (uint32_t i=0; i<LLC_WAY; i++) {
            if (hits_[set][i] <  new_time) {
                new_time = hits_[set][i];
                new_way = i;
            }
        }
    }
    if (new_way >= 0 && new_way < LLC_WAY) {
        return new_way;
    }

    return 0;
}

    //update时(在llc_update_replacement_state函数中)
    time_counter++;
    if (time_counter == 50) {
        miss_counter = 0;
        time_counter = 0;
    }
```

为了实现LRU算法，使用一个数组`hits_[LLC_SET][LLC_WAY]`来记录访问命中。初始化时，将该数组全部初始化为0，只有当命中时才会+1。在`llc_find_victim()`中，我们首先找到invalid line并填充；若没有，则找到hits次数最小的块，并替换它。使用一个时间计数器，每过50次访问后清空`miss_counter`和`time_counter`。

MRU: Most recently used

这个缓存算法最先移除最近最常使用的条目。对于随机访问模式和大数据集的重复扫描（有时被称为循环访问模式），MRU缓存算法比LRU有更多的hit rate，因为他们倾向于保留较早的数据。MRU算法在项目越早，越有可能被访问的情况下最有用。

例如访问：A B C D E C D B



在这里，A B C D被放置在缓存中，因为还有可用的空间。在第5次访问E时，我们看到存放D的区块现在被E取代了，因为这个区块最近被使用过。再一次访问C，在下一次访问D的时候，C被替换掉了，因为它是在D之前访问的区块，以此类推。

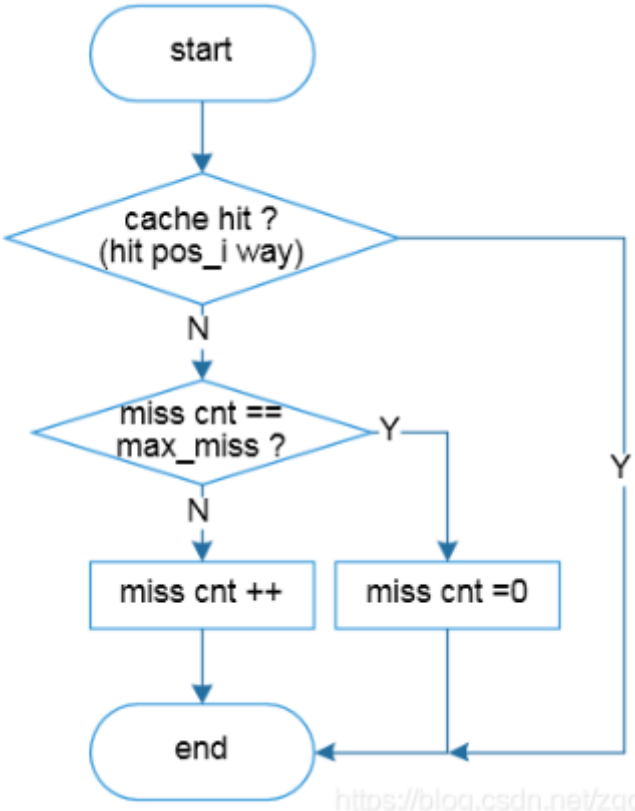
```
uint32_t CACHE::mru_victim(uint32_t cpu, uint64_t instr_id, uint32_t set, const
BLOCK *current_set, uint64_t ip, uint64_t full_addr, uint32_t type)
{
    uint32_t way = 0;
    uint32_t new_way = 0;
    // fill invalid line first
    for (way=0; way<NUM_WAY; way++) {
        if (block[set][way].valid == false) {

            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " invalid set: " << set << " way: " << way;
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][way].address << " data: " << block[set]
[way].data;
            cout << dec << " lru: " << block[set][way].lru << endl; });
            new_way = way;
            break;
        }
    }
    // MRU victim
    if (way == NUM_WAY) {
```

```
        for (way=0; way<NUM_WAY; way++) {
            if (block[set][way].lru == 0) {
                DP ( if (warmup_complete[cpu]) {
                cout << "[" << NAME << "] " << __func__ << " instr_id: " <<
instr_id << " replace set: " << set << " way: " << way;
                cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][way].address << " data: " << block[set]
[way].data;
                cout << dec << " lru: " << block[set][way].lru << endl; });
                //blk = block[set][way];
                //new_way = way;
                break;
            }
        }
    }
    if (new_way == NUM_WAY) {
        cerr << "[" << NAME << "] " << __func__ << " no victim! set: " << set <<
endl;
        assert(0);
    }
    return new_way;
}
```

MRU的实现较为简单，基本思路与LFU相反。LRU判断的是`if(block[set][way].lru == NUM_WAY-1)`，则MRU通过判断`if (block[set][way].lru == 0)`即可。

BIP: Bimodal Insertion Policy

Bimodal Insertion Policy（BIP）替换策略，是LRU和MRU的结合体，大概率采用MRU替换，小概率采用LRU策略。



设置一个固定的`threshold`，用于控制采用LRU/MRU的概率，其概率为$1/threshold$。具体实现时，用一个随机的概率来控制采用LRU还是MRU策略，当`(rand() % 100) <= BIMODAL_THRESH`时采用LRU替换策略，发生替换时将pos_0 way id 移动到MRU位置；否则则采用MRU替换策略，发生替换时保持pos_0 way id不变。

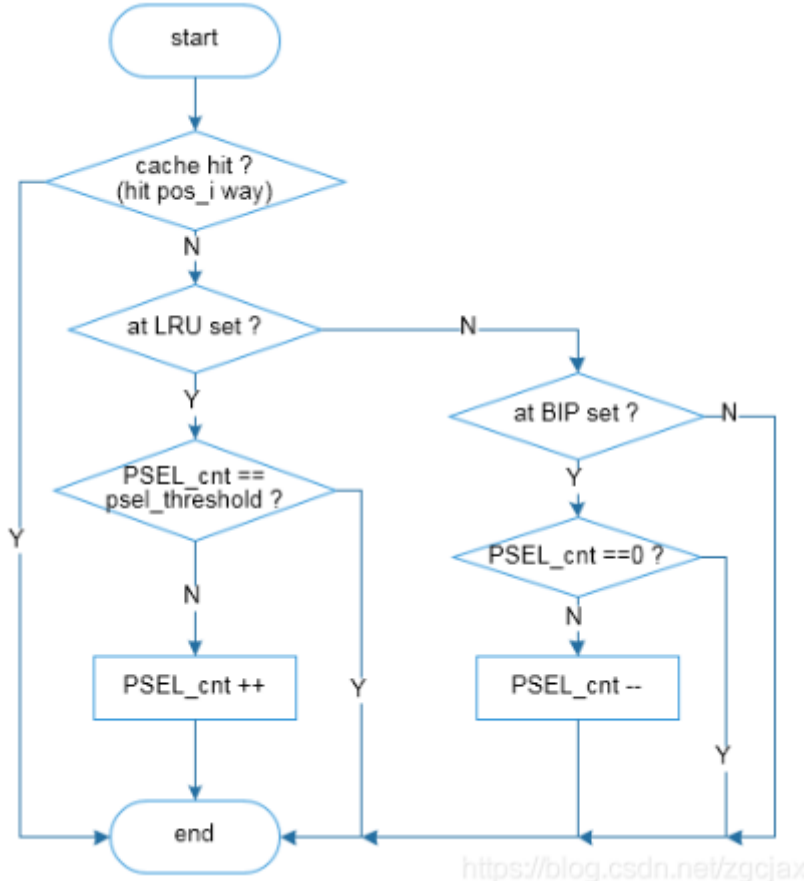
初始化时，将所有BLOCK的lru都设置为`MAX_LRU_POS`，即在LRU位。

```
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 && (type == WRITEBACK)) // WB不更新LRU位
        return;
    else if(hit){
        // hit情况：promote towards MRU
        if(block[set][way].lru > 0)
            block[set][way].lru--;
    }
}

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)
{
    uint32_t way = 0, new_way = 0;

    // fill invalid line first
    for (way=0; way<LLC_WAY; way++) {
        if (block[set][way].valid == false) {

            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " invalid set: " << set << " way: " << way;
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][way].address << " data: " << block[set]
[way].data;
            cout << dec << " lru: " << block[set][way].lru << endl; });

            break;
        }
    }
    // 没有空，则LRU
    if(way == LLC_WAY){
        for (way=0; way<LLC_WAY; way++) {
            if (block[set][way].lru > block[set][new_way].lru) {
                new_way = way;
            }
            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " replace set: " << set << " way: " << new_way;
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][new_way].address << " data: " << block[set]
[new_way].data;
            cout << dec << " lru: " << block[set][new_way].lru << endl; });
        }
    }
    else
        new_way = way;
```

```
    if (new_way == LLC_WAY) {
        cerr << "[" << NAME << "] " << __func__ << " no victim! set: " << set <<
endl;
        assert(0);
    }
    // BIP策略：以6%的概率把该块放到MRU位；否则就放到LRU位(LIP策略)
    if((rand() % 100) <= BIMODAL_THRESH)
        block[set][new_way].lru = 0;
    else
        block[set][new_way].lru = MAX_LRU_POS;

    return new_way;
}
```

DIP: Dynamic Insertion Policy

Dynamic Insertion Policy （DIP）替换策略，是以LRU和BIP替换策略为基础，在整个cache块中实施的一种方法。



LRU、MRU、Pseudo LRU和BIP策略都是对每一个set来讲的，各个set间没有联系。而DIP策略可以认为是，根据某些set (LRU-set 和BIP-set)的采样情况来动态决定其他follower set的替换策略。

假设某1M 的cache，有1024个set，每个set有16个way。分别选择其中16个set作为LRU-set 和BIP-set，其他set为follower set。Set index 为A [15:6]，Set的选择可以这样：A[15:11] == A [10:6] 设置为LRU-set；A[15:11] + id== A [10:6] 设置为BIP-set，这个id比如可以设置为2。

用一个Policy Select Counter (`PSEL_cnt`)来记录LRU-set 和BIP-set的miss情况。其中，当LRU-set发生一次miss时，`PSEL_cnt` 加1；当BIP-set发生一次miss时，`PSEL_cnt` 减1。

当`SEL_cnt`比较大时，表明当前执行的程序，使得采用LRU策略会发生的miss量超过了BIP策略，那就是应该让follower set采用BIP。但由于程序执行时对cache的冲击可能会有一定的波动性，故需要设定一个阈值(PSEL threshold)，当`PSEL_cnt`大于等于这个阈值时，再让follower set 采用BIP；小于阈值时，依然采用LRU。`PSEL_cnt`在超过PSEL threshold时，会做一个取模的操作，也就是说`PSEL_cnt`的取值范围为`[0, PSEL threshold]`。

一般的介绍资料中会提到，当PSEL_cnt的MSB (Most Significant Bit，最高有效位)为1时，选择BIP，否则选择LRU。在RTL实现时，比如准备设置PSEL threshold = 1024，则PSEL cnt就需要11 bit，当PSEL_cnt的MSB为1时，就是PSEL_cnt = PSEL threshold。

```
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 && (type == WRITEBACK)) // WB不更新LRU位
        return;
    else if(hit){
        if(block[set][way].lru > 0)
            block[set][way].lru--;  // hit情况：promote towards MRU
    }
    // miss的时候更新PSEL
    // 当LRU-set发生一次miss时，PSEL_cnt + 1
    // 当BIP-set发生一次miss时，PSEL_cnt - 1
    else{
        if(IS_LIP_POS(set))
            PSEL = (PSEL+1) % PSEL_MAX; //不会超过PSEL threshold
        else if(IS_BIP_POS(set) && PSEL > 0)
            PSEL--;
    }
}

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)
{
    uint32_t way = 0, new_way = 0;
    // fill invalid line first
    for (way=0; way<LLC_WAY; way++) {
        if (block[set][way].valid == false) {
            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " invalid set: " << set << " way: " << way;
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][way].address << " data: " << block[set]
[way].data;
            cout << dec << " lru: " << block[set][way].lru << endl; });
            break;
        }
    }
    // 没有空，则LRU
    if(way == LLC_WAY){
        for (way=0; way<LLC_WAY; way++) {
            if (block[set][way].lru > block[set][new_way].lru) {
                new_way = way;
            }
            DP ( if (warmup_complete[cpu]) {
            cout << "[" << NAME << "] " << __func__ << " instr_id: " << instr_id
<< " replace set: " << set << " way: " << new_way;
```

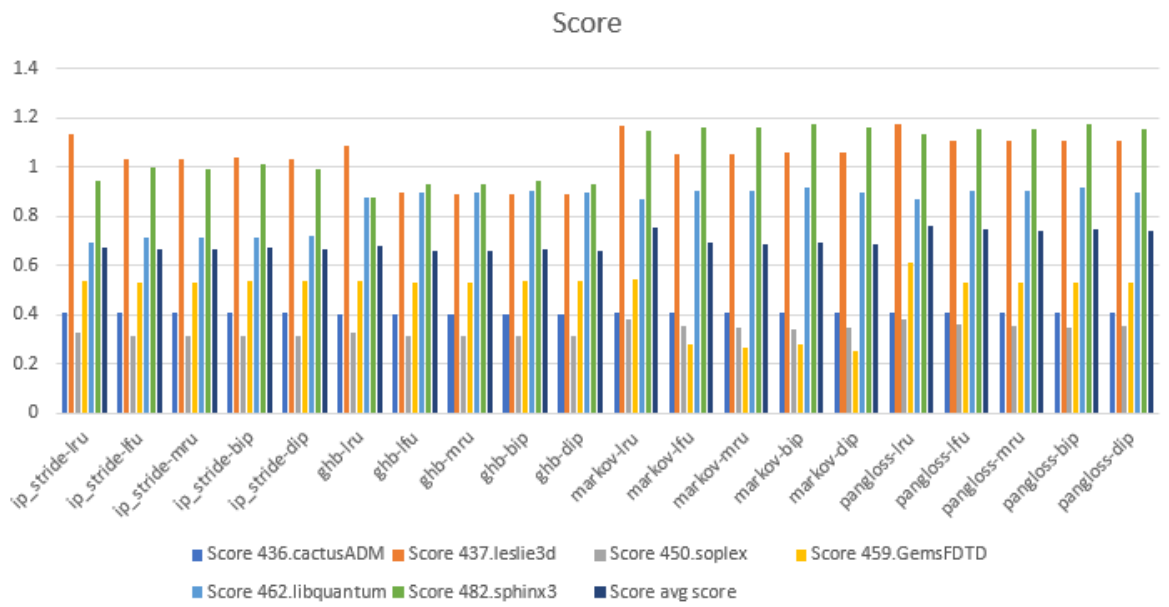
```
            cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) << "
victim address: " << block[set][new_way].address << " data: " << block[set]
[new_way].data;
            cout << dec << " lru: " << block[set][new_way].lru << endl; });
        }
    }
    else
        new_way = way;

    if (new_way == LLC_WAY) {
        cerr << "[" << NAME << "] " << __func__ << " no victim! set: " << set <<
endl;
        assert(0);
    }
    // PSEL比较小时，采用 DIP 策略
    if(IS_LIP_POS(set) || (IS_FOLLOWER(set) && PSEL < ((PSEL_MAX + 1)/2))){
        // LIP MSB是0时表明是follower
        if(block[set][new_way].lru > 0)
            block[set][new_way].lru = 0;
    }
    // PSEL比较大时，表明采用LRU策略会发生的miss量超过BIP，就要用BIP策略
    else if(IS_BIP_POS(set) || (IS_FOLLOWER(set) && PSEL >= ((PSEL_MAX + 1)/2)))
{
        // 小于这个阈值，依旧采用LRU策略，promoted to MRU position
        if((rand() % 100) <= BIMODAL_THROTTLE)
            block[set][new_way].lru = 0;
        // 大于阈值时，insert into LRU positions(LIP)
        else if(block[set][new_way].lru > 0)
            block[set][new_way].lru = MAX_LRU_POS;
    }
    return new_way;
}
```

性能测试

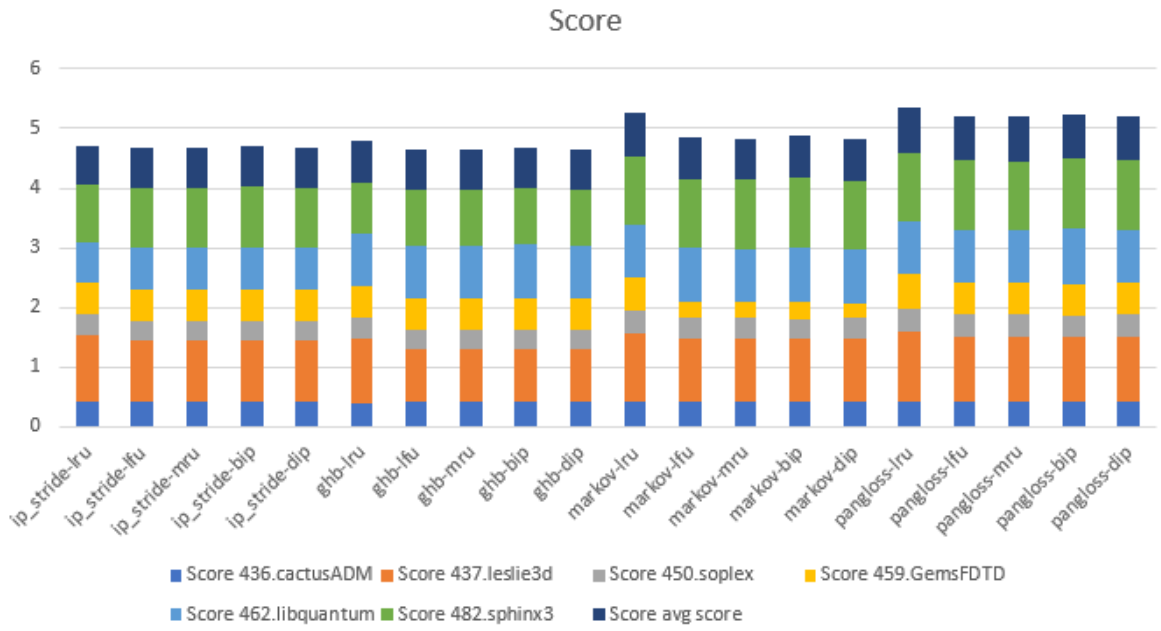
总体性能(score)

score: 测试结束时的cumulative IPC

Score							
	436.cactusADM	437.leslie3d	450.soplex	459.GemsFDTD	462.libquantum	482.sphinx3	avg score
ip_stride-lru	0.4097	1.1362	0.32733	0.53549	0.69346	0.94398	0.67436
ip_stride-lfu	0.41107	1.03565	0.3148	0.53121	0.71723	0.99736	0.667887
ip_stride-mru	0.41107	1.0344	0.31417	0.53009	0.71723	0.99534	0.66705
ip_stride-bip	0.41051	1.03738	0.31519	0.53511	0.71668	1.01526	0.671688
ip_stride-dip	0.41106	1.03416	0.31455	0.53428	0.71749	0.99503	0.667762
ghb-lru	0.40103	1.08604	0.32705	0.53566	0.87361	0.87656	0.683325
ghb-lfu	0.40429	0.89434	0.31599	0.53289	0.89804	0.93131	0.66281
ghb-mru	0.40429	0.8913	0.31539	0.53224	0.89804	0.92973	0.661832
ghb-bip	0.40355	0.89358	0.31557	0.53756	0.90301	0.94563	0.666483
ghb-dip	0.4044	0.89182	0.31495	0.53523	0.89737	0.93023	0.662333
markov-lru	0.40921	1.16559	0.38026	0.54549	0.87261	1.14552	0.753113
markov-lfu	0.40993	1.05488	0.35256	0.27847	0.90077	1.16482	0.693572
markov-mru	0.40993	1.05339	0.34989	0.26373	0.90077	1.16225	0.689993
markov-bip	0.40982	1.05717	0.34204	0.27737	0.9176	1.17425	0.696375
markov-dip	0.41019	1.05695	0.34945	0.25305	0.89986	1.16036	0.68831
pangloss-lru	0.40989	1.17348	0.38019	0.61145	0.87301	1.13785	0.764312
pangloss-lfu	0.4105	1.10739	0.35845	0.53262	0.90119	1.15775	0.74465
pangloss-mru	0.4105	1.10612	0.35611	0.52758	0.90119	1.15429	0.742632
pangloss-bip	0.41023	1.10729	0.34966	0.52825	0.91766	1.1729	0.747665
pangloss-dip	0.41045	1.10624	0.35703	0.5306	0.89945	1.15488	0.743108



从总体score中就可以看出，GHB步长预取相较于ip_stride来说会有一些优化效果，但作用并不是很明显。而Markov和Pangloss的性能提升较为显著，且Pangloss预取器的score分最高，平均达到了0.74左右。对比Cache替换算法可以发现，这些替换算法的性能表现各有千秋，在不同点数据集上差异很大；但是总体来说，LRU算法的性能依旧是比较好的，其score分基本大于其他4个Cache替换策略。



在所有的组合中，表现做好的是pangloss+lru，其次是markov+lru；pangloss和其他几个Cache替换策略结合都有较好的表现。

对于数据集来说，这些组合在437.leslie3d-134B.champsimtrace.xz和482.sphinx3-1100B.champsimtrace.xz上都有较好的表现，而在450.soplex-247B.champsimtrace.xz和436.cactusADM-1804B.champsimtrace.xz上表现得较差。

Prefetcher性能

- L1D_PREFETCHER = no
- LLC_PREFETCHER = no
- LLC_REPLACEMENT = lru
- NUM_CORE = 1
- warm up指令数：50M；需要模拟的指令数：100M

1. **使用数据集437.leslie3d-134B.champsimtrace.xz**

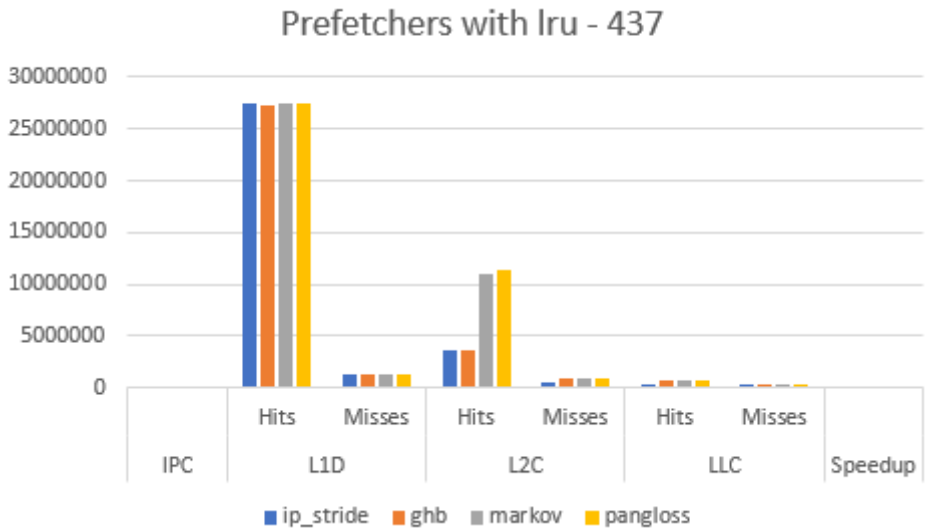
Prefetcher	IPC	L1D		L2C		LLC		Speedup
		Hits	Misses	Hits	Misses	Hits	Misses	
ip_stride	1.1362	27384650	1410642	3564917	628715	377427	388020	-
ghb	1.08604	27362722	1410443	3660810	936453	709450	388091	0.955853
markov	1.16559	27397140	1410670	10962240	978787	754943	388660	1.025867
pangloss	1.17348	27407088	1410666	11357532	946005	719465	388430	1.032811

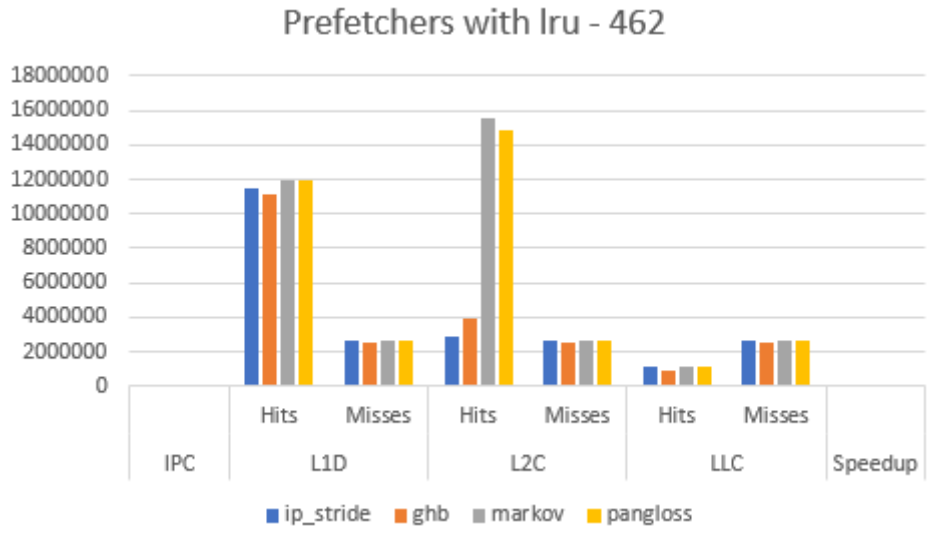
2. **使用数据集462.libquantum-714B.champsimtrace.xz**

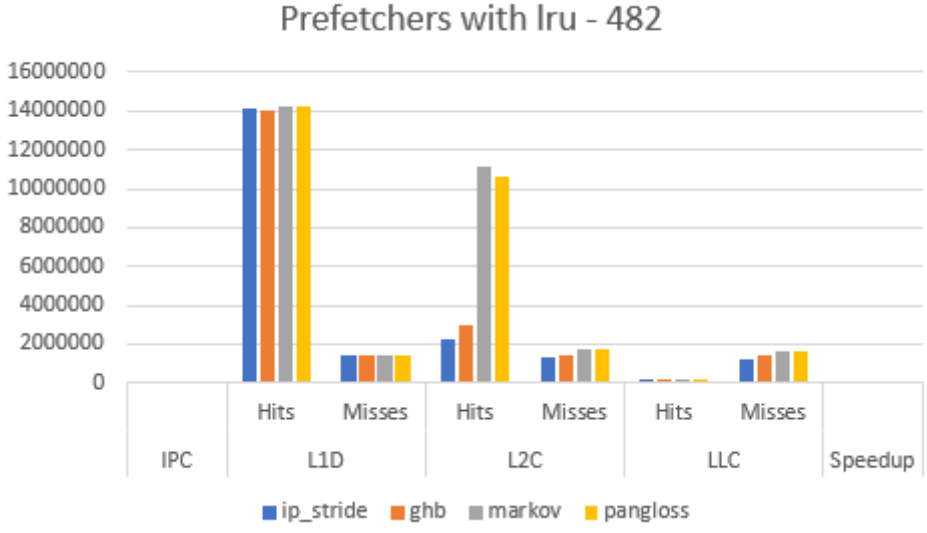
Prefetcher	IPC	L1D		L2C		LLC		Speedup
		Hits	Misses	Hits	Misses	Hits	Misses	
ip_stride	0.69346	11457341	2651651	2843297	2651663	1101274	2651663	-
ghb	0.87145	11165312	2576223	3903208	2576240	942706	2576241	1.256669
markov	0.87261	11906184	2651651	15504114	2651797	1101275	2651797	1.258342
pangloss	0.87301	11906764	2651651	14882686	2651836	1101275	2651836	1.258919

3. **使用数据集482.sphinx3-1100B.champsimtrace.xz**

Prefetcher	IPC	L1D		L2C		LLC		Speedup
		Hits	Misses	Hits	Misses	Hits	Misses	
ip_stride	0.94398	14160562	1467609	2273702	1301822	172511	1234118	-
ghb	0.87656	14082864	1467601	2979732	1465096	181175	1388986	0.928579
markov	1.14552	14276821	1467542	11182736	1759313	227245	1638296	1.2135
pangloss	1.13785	14277114	1467523	10639690	1713646	226722	1592566	1.205375







从这三张图表中可以很明显地看出，相较于basline: ip_stride Prefetcher，GHB、Markov、Pangloss Prefetcher都能增加L2C中的Hits；其中Markov和Pangloss较为显著，且Markov Prefetcher表现最佳。在数据集437.leslie3d-134B.champsimtrace.xz上GHB和ip_stride区分得并不明显，但是已经可以看到Markov和Pangloss Prefetcher的Hits rate是前二者的两倍之多；而在数据集462.libquantum-714B.champsimtrace.xz和482.sphinx3-1100B.champsimtrace.xz上表现得更为显著。

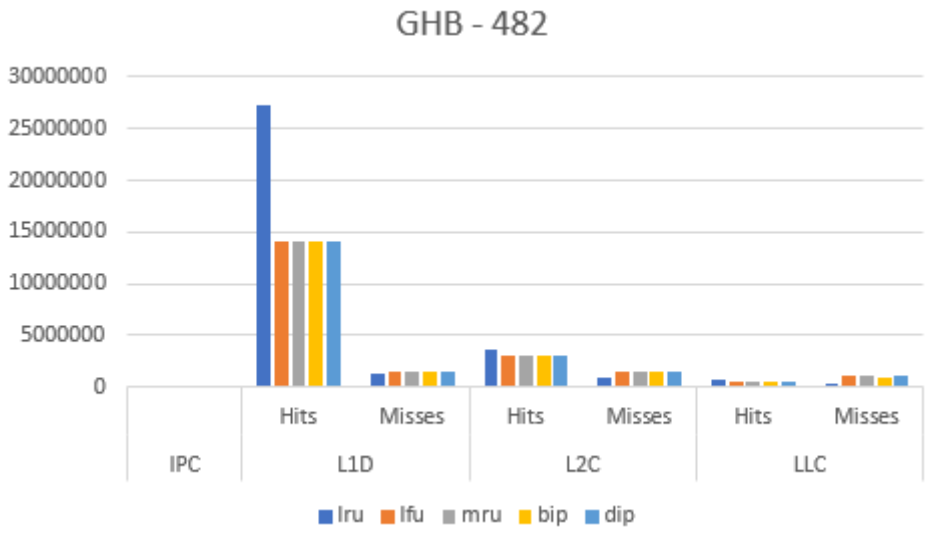
由此可以得出结论：Markov和Pangloss算法能够优化预取器在L2C上的性能，且Markov最为显著；GHB也能拥有一定的加速比，但需要针对数据集而言；在广泛的测试中，可以证明GHB的性能较ip_stride更优（见score部分）。

Replacement Policies性能

- L1D_PREFETCHER = no
- LLC_PREFETCHER = no
- LLC_REPLACEMENT = lru/lfu/mru/bip/dip
- NUM_CORE = 1
- warm up指令数：50M；需要模拟的指令数：100M

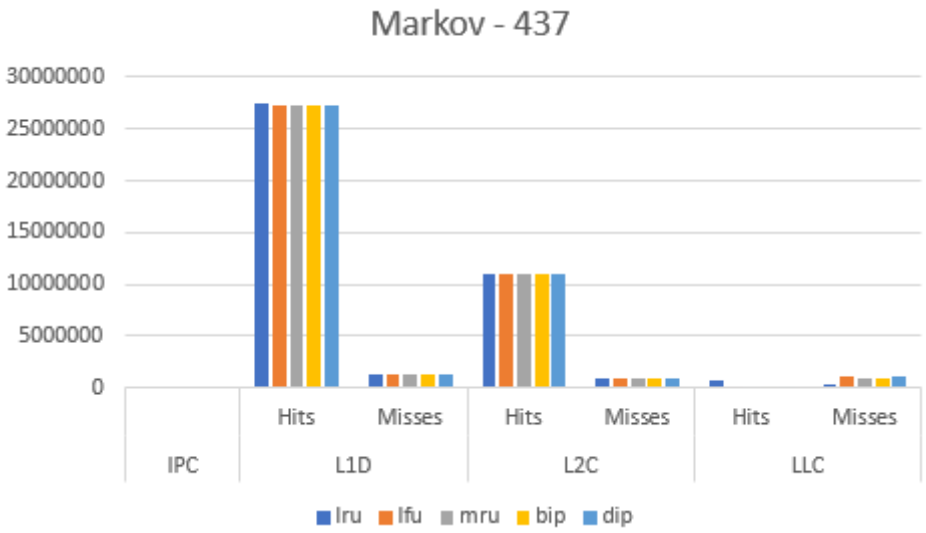
1. **GHB - 482.sphinx3-1100B.champsimtrace.xz**

Repl Policy	IPC	L1D		L2C		LLC	
		Hits	Misses	Hits	Misses	Hits	Misses
lru	1.08604	27362722	1410443	3660810	936453	709450	388091
lfu	0.93131	14130906	1467568	2976572	1465020	529515	1040570
mru	0.92973	14129185	1467571	2976445	1465020	528520	1041565
bip	0.94563	14133482	1467599	2987291	1461775	560997	1005731
dip	0.93023	14130714	1467588	2973611	1467451	532654	1039910



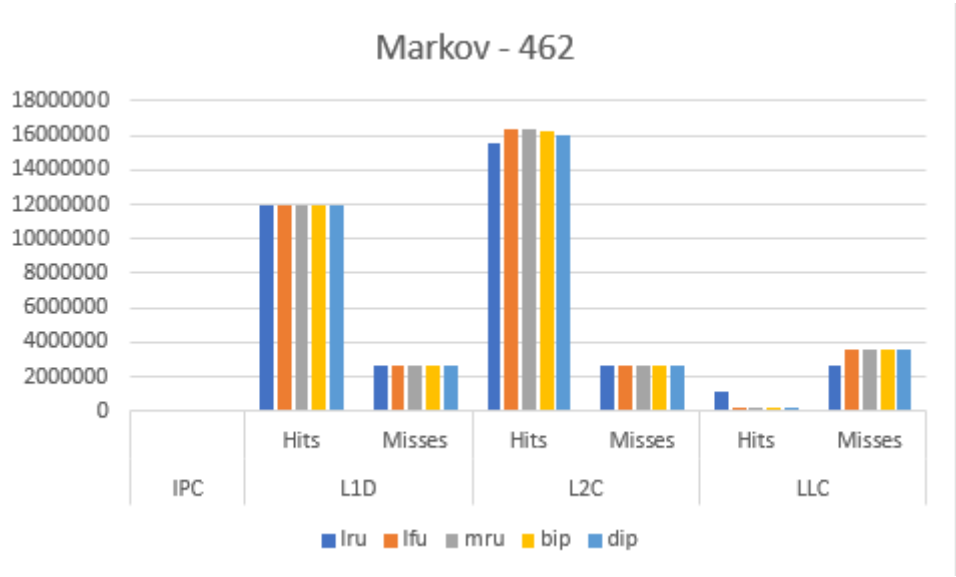
2. **Markov - 437.leslie3d-134B.champsimtrace.xz**

Repl Policy	IPC	L1D		L2C		LLC	
		Hits	Misses	Hits	Misses	Hits	Misses
lru	1.16559	27397140	1410670	10962240	978787	754943	388660
lfu	1.04864	27353065	1410680	10937420	982550	84758	1062787
mru	1.05339	27369304	1410692	10968990	978809	121818	1021647
bip	1.05717	27374075	1410705	11000055	978425	118519	1019218
dip	1.05695	27370060	1410731	10940306	987610	120946	1030764



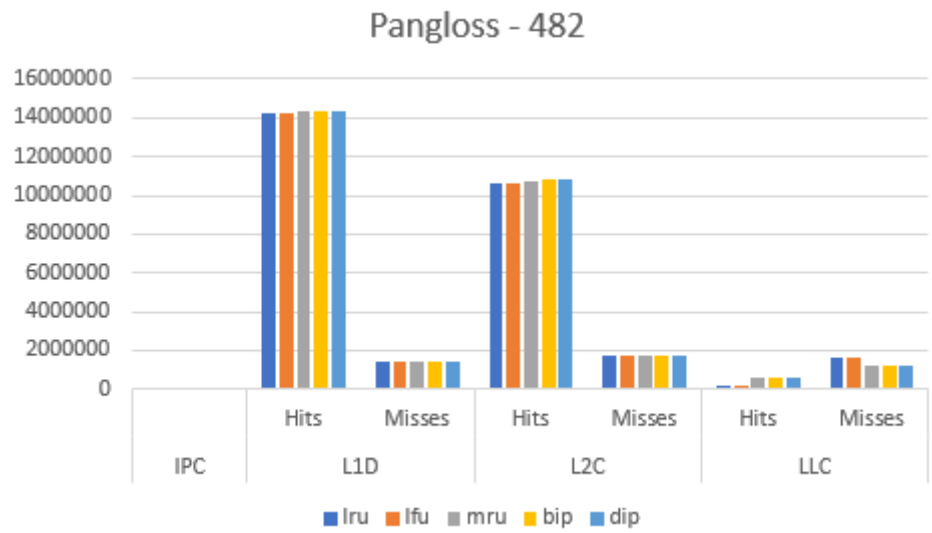
3. **Markov - 462.libquantum-714B.champsimtrace.xz**

Repl Policy	IPC	L1D		L2C		LLC	
		Hits	Misses	Hits	Misses	Hits	Misses
lru	0.87261	11906184	2651651	15504114	2651797	1101275	2651797
lfu	0.90077	11910879	2651651	16369320	2651797	217920	3535152
mru	0.90077	11910879	2651651	16369320	2651797	217920	3535152
bip	0.9176	11911389	2651651	16260025	2651835	221339	3531771
dip	0.89986	11909529	2651651	16076048	2651844	217966	3535153



Pangloss - 482.sphinx3-1100B.champsimtrace.xz

Repl Policy	IPC	L1D		L2C		LLC	
		Hits	Misses	Hits	Misses	Hits	Misses
lru	1.13785	14277114	1467523	10639690	1713646	226722	1592566
lfu	1.13785	14277114	1467523	10639690	1713646	226722	1592566
mru	1.15429	14321076	1467535	10759835	1713568	560378	1258824
bip	1.1729	14325483	1467513	10800987	1730224	603510	1232312
dip	1.1729	14325483	1467513	10800987	1730224	603510	1232312



从图表中可以看出：

1. GHB与LRU结合表现较好，在L1D拥有较高的Hits数。
2. 不同的数据集让Cache替换策略的算法性能有差异。比如对于Markov Prefetcher，在437数据集上，数据大多数在L1就能命中，因此IPC较高；而在462数据集上，数据大部分到L2C才命中，IPC有了一定程度的降低。
3. 使用Pangloss Prefetcher，在482数据集上，BIP和DIP算法性能表现较为优秀，IPC超过LRU。
4. 几个Cache替换策略的算法性能差别并不大，但总体来说LRU会更加优秀一些。

参考文献

1. S. Pakalapati and B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 118-131, doi: 10.1109/ISCA45697.2020.00021.
2. Nesbit, K. J., & Smith, J. E. (2004, February). Data cache prefetching using a global history buffer. In 10th International Symposium on High Performance Computer Architecture (HPCA'04) (pp. 96-96). IEEE.
3. Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. SIGARCH Comput. Archit. News 25, 2 (May 1997), 252–263. https://doi.org/10.1145/384286.264207
4. Papaphilippou P, Kelly P H J, Luk W. Pangloss: a novel Markov chain prefetcher[J]. arXiv preprint arXiv:1906.00877, 2019.
5. Cache替换算法. https://blog.csdn.net/zgcjaxj/article/details/104740111
6. Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies
7. Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. SIGARCH Comput. Archit. News 35, 2 (May 2007), 381–391. https://doi.org/10.1145/1273440.1250709