南開大學
Nankai University

计算机学院

计算机体系结构期末报告

Cache 替换策略和数据预取探究

姓名：刘宇轩

学号：2012677

专业：计算机科学与技术

2023 年 1 月 7 日

摘要

本次实验，根据实验基本要求，基于 Champsim，在 L2C 上完成了 GHB 数据预取策略，在 LLC 上完成了基于 LFU 的 cache 替换策略。通过调整 GHB 中的相关参数，在给定的两个 trace 上，平均 IPC 达到 **1.02**。除了完成了实验基本要求外，笔者还进行了额外的探究，根据最近在数据预取和 cache 替换相关的比赛以及发表的论文，学习并复现了 Pangloss 数据预取策略，SHiP++ 和 ReD 的 cache 替换策略。此外，笔者还对 Champsim 提供的相关算法和新实现的算法之间进行了组合测试，并对比了不同组合策略之间的性能差异，综合对比之后，笔者最终采用了在 L1D 和 L2C 中实现了 Pangloss 数据预取策略，在 LLC 中实现了 SHiP++ 配合 ReD 的 cache 替换策略，并调整了其中的参数，在给定的两个 trace 上，平均 IPC 达到了 **1.127**，达到了无数据预取基于 LRU 的 cache 替换策略的方法的 **2.12** 倍，但是这种提高了 IPC 的策略必定是通过消耗较大的硬件资源换取的，硬件开销达到了 103.6KB。

关键词：数据预取，cache 替换策略，pangloss，SHiP++，ReD

目录

1 引言

cache 替换策略和数据预取策略是提高 cache 命中率的两个肿瘤方式，通过数据预取策略，能够提前将需要的数据存入到 cache 中，以便在下一次访存的时候提前准备好数据，降低 cache miss 导致的访存开销。通过 cache 替换策略，能够在有限的 cache 资源中实现尽可能高的 cache 命中，将一些优先级较低的 cache line 替换出去，给优先级更高的 cache line 提供充足的资源。

有关数据预取，当前存在的一些困难主要包含如何高效记录以往的访问信息，如何准去根据缓存的访问信息预测之后的访问情况，以及需要预取多少数据到 cache 中来。CPU 硬件资源十分有限，因此不能够使用非常多的空间来存储过往的访问信息，需要采用一些高效的手段或者合理的替换策略来保存有效历史信息。而预测的准确与否与之前保存的数据信息十分相关，并且其结果会极大影响到 cache 的效率。如果预取的数据偏差很大，并且预取深度很高，则会导致 cache 污染或者颠簸。有关 cache 替换策略，当前存在的问题也主要集中在如何有效记录历史访问信息和决策替换目标上。

本次实验要求咋 Champsim 中实现数据预取算法和 cache 替换策略，并在给定的 trace 中进行测试。笔者首先按照基本要求，在 L2C 中完成了 GHB 数据预取策略，在 LLC 中完成了基于 LFU 的 cache 替换策略。此外笔者还根据近年来数据预取和 cache 替换相关的竞赛和论文，总结了基于访存间隔构建马尔可夫链进行预取预测的 pangloss 策略，该策略能够很好地应用在 L1D 和 L2C 中。并且基于 SHiP 策略参考了更细粒度进行更新改进的 SHiP++ 策略，和在 LLC 上对 cache line 进行 bypass 的 ReD 策略。将上述三种方法结合起来，最终实现了 IPC 的较大提高。

2 基础要求的实现

2.1 GHB 预取策略的实现

GHB 预取策略 [2] 的核心思想是维护了一个全局的历史记录（Global History Buffer），这个历史记录表中，采用 FIFO 的策略管理在之前访问产生的访问记录。此外 GHB 预取策略还建立了一个索引表（Index Table），这个索引表根据 IP 的哈希值进行索引，在本次实验中是通过取 IP 的低 8 位实现的。IT 表中通过 IP 进行索引，能够获取到该指令 IP 最近一次访问的数据地址 cache line 在 GHB 中的索引。GHB 表中的每一项包含两部分，一部分是 cache line 地址，另一部分是上一次该 IP 访问的数据对应的 GHB 表项的索引，通过这部分，能够将该指令 IP 最近访问过的数据的地址串联在一起。GHB 的基本架构如图2.1所示。

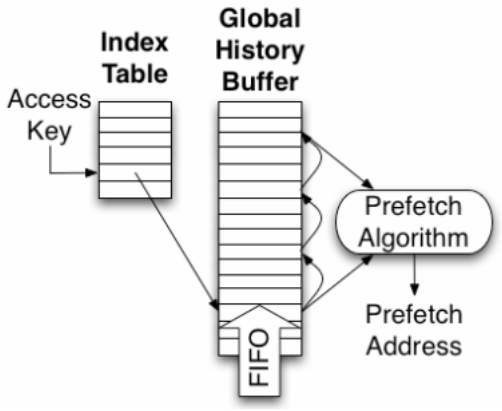

图 2.1: GHB 整体架构图 [2]

当进行预取的时候，首先根据指令 IP 在 IT 表中进行索引，找到该指令上一次访问到的数据地址在 GHB 中对应的表项，然后寻找该指令前三次访存对应的表项，根据其中保存的 cl_addr 计算 stride，判断这两次的 stride 是否相等，如果相等的话则证明当前的数据访问可能存在一定的规律性，需要进行预取。而如何进行预取是由 lookahead 以及 degree 决定的，lookahead 决定在预取的时候需要向前看多远的距离，而 degree 决定要预取多少步。

在完成预取之后，需要更新 GHB。由于 GHB 采用了 FIFO 策略进行管理，因此需要将本次新产生的 GHB 表项，插入 GHB 的表头，并查询 IT 表，修改该 IP 对应的历史访问数据链。

具体算法实现如下

GHB 算法实现细节

```
#define INDEX_TABLE_SIZE 256
#define GHB_SIZE 256
#define PREFETCH_LOOKAHEAD 1
#define PREFETCH_DEGREE 12

// 定义 GHB element
struct GHBElement {
    uint64_t cl_addr;
    int16_t prev;
};

// 定义 index table
int16_t IT[INDEX_TABLE_SIZE];

// 定义 GHB
GHBElement GHB[GHB_SIZE];
int16_t GHB_end = 0;
......
uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t cache_hit,
    uint8_t type, uint32_t metadata_in)
{
    // 如果GHB满了，需要将 GHB_end 处的 entry 移出
    // 即使没满 GHB_end 处的 entry 是无效的 因此不会产生错误
    // 重置 IT 和 GHB 中对应项
    for(int i = 0; i < INDEX_TABLE_SIZE; i++) {
        if(IT[i] == GHB_end) {
            IT[i] = -1;
        }
    }
    for(int i = 0; i < GHB_SIZE; i++) {
        if(GHB[i].prev == GHB_end) {
            GHB[i].prev = -1;
        }
    }
    // 确定 IT 表中的索引
    int indexIT = ip % INDEX_TABLE_SIZE;
    uint64_t cl_addr = addr >> LOG2_BLOCK_SIZE;
    // 如果 IT 表中该索引位是有效的 则需要将新的 push_entry 串入之前的链表中
```

```
38      GHB[GHB_end].cl_addr = cl_addr;
39      GHB[GHB_end].prev = IT[indexIT];
40      IT[indexIT] = GHB_end;
41      // 判断最近三次的 stride 是否一致
42      int16_t index = GHB_end;
43      GHB_end = (GHB_end + 1) % GHB_SIZE;
44      uint64_t last_cl_addr[3];
45      for(int i = 0; i < 3; i++) {
46          if(index == -1) {
47              return metadata_in;
48          }
49          last_cl_addr[i] = GHB[index].cl_addr;
50          index = GHB[index].prev;
51      }
52      int64_t stride1 = 0;
53      if (last_cl_addr[0] >= last_cl_addr[1])
54          stride1 = last_cl_addr[0] - last_cl_addr[1];
55      else {
56          stride1 = last_cl_addr[1] - last_cl_addr[0];
57          stride1 *= -1;
58      }
59      int64_t stride2 = 0;
60      if (last_cl_addr[1] >= last_cl_addr[2])
61          stride2 = last_cl_addr[1] - last_cl_addr[2];
62      else {
63          stride2 = last_cl_addr[2] - last_cl_addr[1];
64          stride2 *= -1;
65      }
66      // 如果相邻三次的 stride 相等 则进行预取
67      if(stride1 == stride2) {
68          for(int i = 0; i < PREFETCH_DEGREE; i ++) {
69              uint64_t pref_addr = (cl_addr + PREFETCH_LOOKAHEAD * (stride1 * (i + 1)))
                    << LOG2_BLOCK_SIZE;
70              // 只有在同一个4kb的page中才进行预取
71              if((pref_addr >> LOG2_PAGE_SIZE) != (addr >> LOG2_PAGE_SIZE)) {
72                  break;
73              }
74              // 判断是否需要填充 LLC
75              if(MSHR.occupancy < (MSHR.SIZE >> 1)) {
76                  prefetch_line(ip, addr, pref_addr, FILL_L2, 0);
77              }
78              else {
79                  prefetch_line(ip, addr, pref_addr, FILL_LLC, 0);
80              }
81          }
82      }
83      return metadata_in;
84  }
```

2.2 LFU 替换策略的实现

LFU 是一种简化版近似的 LRU 替换策略，即通过计数 cache line 被访问的频率来替换出最近最少使用过的 cache line。在实现层面，只需要给每一个 cache line 增加一个计数域，当这个 cache line 被访问的时候，就递增计数域。当需要逐出的时候，则在当前的 set 中寻找一个 LFU 计数域最小的 way 逐出。

具体的实现也非常简单。只需要在 Champsim 提供的框架中，根据 LRU 替换策略提供的接口进行修改即可。在更新状态的时候直接将 block 中的 LRU 计数域递增即可，在进行逐出的时候，则在当前的 set 中寻找 LRU 计数域最小的 way 逐出。

具体算法实现如下：

LFU 算法实现细节

```
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;

    // 现在当前的 set 里找一个还没有用过的 way
    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; });

            break;
        }
    }

    // 如果当前 set 中的所有 way 都已经使用了 则需要替换出去一个最近最少使用过的 way
    uint32_t min_freq = 1e9;
    if (way == NUM_WAY) {
        for (int i =0; i < NUM_WAY; i++) {
            if(block[set][i].lru < min_freq) {
                way = i;
                min_freq = block[set][i].lru;
            }
        }
    }

    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;
32      cout << dec << " lru: " << block[set][way].lru << endl; });
33
34      if (way == NUM_WAY) {
35          cerr << "[" << NAME << "] " << __func__ << " no victim! set: " << set << endl;
36          assert(0);
37      }
38
39      return way;
40  }
41
42  // called on every cache hit and cache fill
43  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)
44  {
45      string TYPE_NAME;
46      if (type == LOAD)
47          TYPE_NAME = "LOAD";
48      else if (type == RFO)
49          TYPE_NAME = "RFO";
50      else if (type == PREFETCH)
51          TYPE_NAME = "PF";
52      else if (type == WRITEBACK)
53          TYPE_NAME = "WB";
54      else
55          assert(0);
56
57      if (hit)
58          TYPE_NAME += "_HIT";
59      else
60          TYPE_NAME += "_MISS";
61
62      if ((type == WRITEBACK) && ip)
63          assert(0);
64
65      // uncomment this line to see the LLC accesses
66      // cout << "CPU: " << cpu << "  LLC " << setw(9) << TYPE_NAME << " set: " <<
            setw(5) << set << " way: " << setw(2) << way;
67      // cout << hex << " paddr: " << setw(12) << paddr << " ip: " << setw(8) << ip <<
            " victim_addr: " << victim_addr << dec << endl;
68
69      // baseline LRU
70      if (hit && (type == WRITEBACK)) // writeback hit does not update LRU state
71          return;
72
73      // return lru_update(set, way);
74      block[set][way].lru++;
75  }
```

3 改进提高实现

3.1 Pangloss 预取策略的实现

马尔可夫图已经广泛用于预取策略，主要包含两种构造策略，一种是基于访存地址进行构造，即预测访问了当前地址之后，下一次可能访问的数据地址，这个策略存在较大的资源浪费，由于在程序中访问的内存地址会非常多，因此需要为每一个地址都建立一个节点，并且很有可能访问的顺序差异很大，导致构造出的马尔可夫图节点和边都非常多，但其实有效信息并不多，造成了较大的资源浪费。第二种是基于访问间隔建立的，即计算两次访问之间内存地址的间隔，并使用这个间隔建立马尔可夫图，这样的好处是，对于不同的数据地址，在访问序列中可能表现出相同的间隔模式，因此这样的马尔可夫图能够很好被各种访问序列应用，并且由于在构造的时候综合考虑了各种访存序列的间隔情况，因此对于整体访存间隔的预测也会更加准确。

例如给出如下的访存序列，可以计算出 Delta，进而构建出马尔可夫图如图3.2所示。进而可以根据马尔可夫图计算出当前访存产生 Delta 的情况下，下一个 Delta 的可能值。在本例中，当 Delta=3 的时候，下一次分别由 0.5 的概率产生-2 或 2 的 Delta。

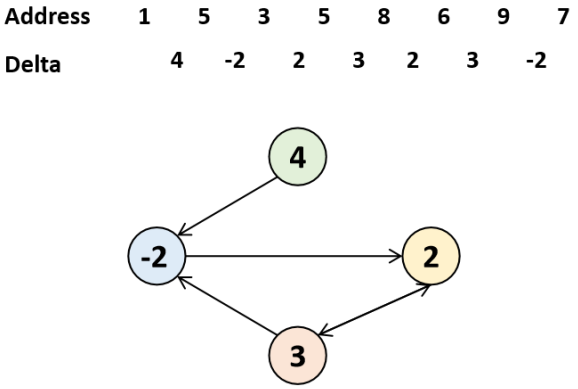

图 3.2: 基于 Delta 构建马尔可夫图示例

Philippos 等人 [3] 通过对实际应用中的一些具体场景构建马尔可夫图，并可视化了其邻接矩阵，如图3.3所示。其中 x 轴表示当前的 Delta，y 轴表示下一个 Delta。从图中可以发现一些规律性的结论。临界矩阵中的值集中分布在 x 轴、y 轴以及负对角线上，并且集中分布在靠近原点的附近。1）对角线的情况看起来像是随机访问，但其实也是一种有规律的访问，例如任何一个流式访问在访问中都会是 (δ, δ) 的形式，而当其需要第二次访问的时候，就会产生 $(\delta', -\delta' + \delta)$ 这样的转移形式。2）贴近 x 轴和 y 轴的情况也和上一个样例有关，因为在产生这样的二次访问之前和之后，需要 (δ, δ') 和 $(-\delta' + \delta, \delta)$ 的转移形式。3）而之所以所有的点都靠近原点就是因为访存会收到页大小的限制，大部分的连续访存都不会跨页进行。基于上述的实验探究，Philippos 等人提出了一种基于马尔可夫链的预取策略，通过记录每页中的 Delta 信息来构建一个基于 Delta 的全局马尔可夫预测模型。

3.1.1 Delta Cache 的实现

Delta Cache 是一种以 Delta 作为索引的缓存表，用来提供在当前 Delta 情况时，下一次的 Delta 的取值情况信息。通过组相联的方式，在同一个 Delta 索引下提供多个下一次 Delta 的可能情况。由于可能会发生冲突而进行替换，所以在组内采用 LFU 的替换策略，优先保留最近使用过的 Delta 值，当被访问到的时候，LFU 域进行递增，如果发生了溢出，则该 set 中的全部元素的 LFU 位都折半，这样就能够使用较少的位精确估计每个 way 中使用情况，并且保证了他们之间的相对大小关系。

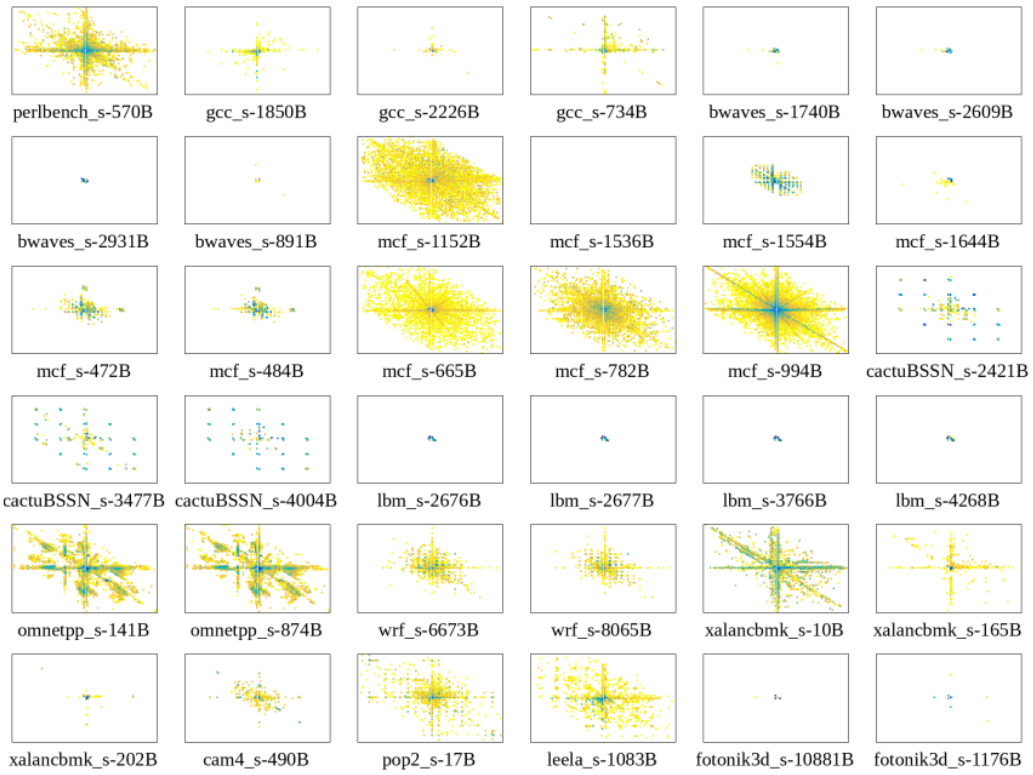

图 3.3: 在多个 trace 上的马尔可夫邻接矩阵可视化 [3]

对于 Delta Cache 的具体实现，沿用了论文中的默认参数，在 L1d cache 中，选用了 1023 个 set，16 个 way，每一个 entry 中，10 位记录 $Delta_{next}$，6 位记录 LFU。在 L2c cache 中，选用 127 个 set，16 个 way，每个 entry 中，7 位记录 $Delta_{next}$，8 位记录 LFU。以 L2c cache 为例，展示 Delta cache 的结构，如图3.4所示。

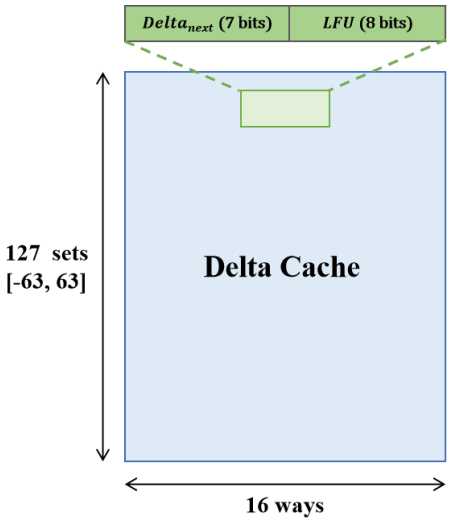

图 3.4: Delta Cache 结构示意图

3.1.2 Page Cache 的实现

现代操作系统出于安全考虑，通常会将内存进行分页管理，在这种机制下即使是连续的内存对应的也不一定是连续的物理内存页，因此在计算 Delta 的时候，只有必要对于同一个 page 中的访存计算 Delta，因为不同页之间的 delta 没有意义。所以 Pangloss 策略建立了一个基于 page 索引的 Page Cache 缓存表，用来存放每一页中，上一次访问的偏移和产生的 Delta。

因此当每次产生预取的时候，都会先根据当前的地址，到 Page Cache 中查询对应的 page 中缓存的上一次访问的 $offset_{prev}$ 和 $delta_{prev}$，并根据 $offset_{prev}$ 和本次的 $offset$ 计算出本次访存产生的 $delta$，然后利用上一次的 $delta_{prev}$ 在 Delta Cache 中刷新本次产生的 $delta$ 的 LFU 项。完成更新 Delta Cache 的更新之后，再将本次产生的 $delta$ 写入到 Page Cache 中。此时同样需要更新 Page Cache 中对应项的 NRU 位。

对于 Page Cache 的具体实现，沿用了论文中的默认参数，在 L1d cache 中，选用了 256 个 set，12 个 way，每一个 entry 中，10 位标志 Page Tag，10 位记录 $Delta_{prev}$，9 位记录 $offset_{prev}$，1 位记录 NRU。在 L2c cache 中，选用 256 个 set，12 个 way，每个 entry 中，10 位标志 Page Tag，7 位记录 $Delta_{prev}$，6 位记录 $offset_{prev}$，1 位记录 NRU。以 L2c cache 为例，展示 Page cache 的结构，如图3.5所示。

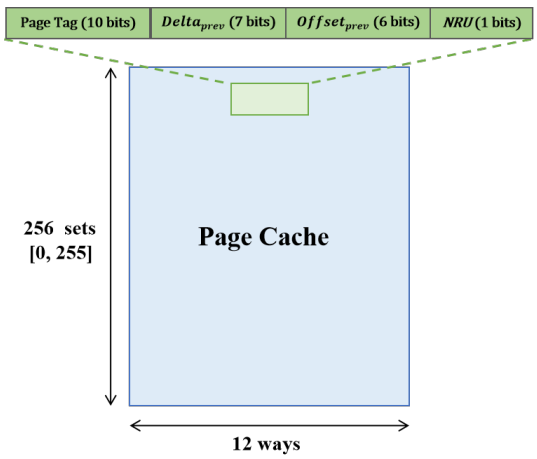

图 3.5: Page Cache 结构示意图

3.1.3 基于马尔科夫链的预取决策

在获取到了当前访问的地址之后，需要决定如何进行预取。如果想要准确预测下一次访问将会产生的 delta，需要在建立起的马尔可夫图上进行随机游走，但是这样的计算开销非常大，因此 Philippos 等人 [3] 指出，可以采用一种启发式的算法，即只预取概率超过 1/3 的 delta。这样做的目的有两个，一方面是选取到了概率较大的进行预取，另一方面是概率超过 1/3 的 delta 最多不会超过 2 个，能够提高效率。在估算概率的时候，只以当前 delta 中缓存的所有项的 LFU 记录作为依据计算概率。根据预取度，每次选择了一个 delta 进行预取之后，下一次需要在选择的这个 delta 上进行预测。

算法具体实现如下：

Pangloss 算法实现细节

```
1 #define WORD_SIZE_OFFSET 2
2 #define PAGE_SIZE_OFFSET 12
3
```

```
// ========== 定义 l1d cache 预取的相关数据结构 ==========

#define L1D_PREFETCH_DEGREE 36

// l1d Delta Cache 相关数据
// l1d Delta Cache 大小为 1024 sets * 16 ways
#define L1D_DELTA_CACHE_SETS 1024
#define L1D_DELTA_CACHE_WAYS 16
// LFU 计数的最大值 由于 LFU 共8位 所以取值 128
#define L1D_DELTA_CACHE_MAX_LFU 128
// l1d Delta Cache 表项
struct L1DDeltaCacheEntry {
    int next_delta;
    int LFU_count;
};
// l1d Delta Cache
static L1DDeltaCacheEntry L1D_Delta_Cache[L1D_DELTA_CACHE_SETS][L1D_DELTA_CACHE_WAYS];

// l1d Page Cache 相关数据
// l1d Page Cache 大小为 256 sets * 12 ways
#define L1D_PAGE_CACHE_SETS 256
#define L1D_PAGE_CACHE_WAYS 12
// page tag 位宽为10
#define L1D_PAGE_CACHE_TAGE_BITS 10
// l1d Page Cache 表项
struct L1DPageCacheEntry {
    int page_tag;
    int last_delta;
    int last_offset;
    int NRU_bit;
};
// l1d Page Cache
static L1DPageCacheEntry L1D_Page_Cache[L1D_PAGE_CACHE_SETS][L1D_PAGE_CACHE_WAYS];

/*
* 根据给定的page 返回 page tag
*/
static int get_l1d_page_tag(uint64_t page) {
    // 取高位
    uint64_t high_bits = page / L1D_PAGE_CACHE_SETS;
    // 取高位中的低10位
    int page_tag = high_bits & ((1 << L1D_PAGE_CACHE_TAGE_BITS) - 1);
    return page_tag;
}

/*
* 更新 l1d cache
* 增加了一个 delta transition (delta_from -> delta_to)
*/
```

```
53 static void update_l1d_delta_cache(int delta_from, int delta_to) {
54   // 首先检查 delta_to 是否在 Delta Cache 中命中、
55   // 此时同时检查 LFU 最小的 way
56   int lfu_way = 0;
57   int min_uses = 1e9;
58   for (int i = 0; i < L1D_DELTA_CACHE_WAYS; i++) {
59     if (L1D_Delta_Cache[delta_from][i].next_delta == delta_to) {
60       L1D_Delta_Cache[delta_from][i].LFU_count++;
61       // 如果发生了溢出 则整体折半
62       if (L1D_Delta_Cache[delta_from][i].LFU_count == L1D_DELTA_CACHE_MAX_LFU) {
63         for (int j = 0; j < L1D_DELTA_CACHE_WAYS; j++) {
64           L1D_Delta_Cache[delta_from][j].LFU_count /= 2;
65         }
66       }
67       return;
68     }
69     if(L1D_Delta_Cache[delta_from][i].LFU_count < min_uses) {
70       min_uses = L1D_Delta_Cache[delta_from][i].LFU_count;
71       lfu_way = i;
72     }
73   }
74
75   // 如果这个 delta transition 不在 Delta Cache 中
76   // 则需要根据 LFU 逐出一项 并插入新的 delta transition
77   L1D_Delta_Cache[delta_from][lfu_way].next_delta = delta_to;
78   L1D_Delta_Cache[delta_from][lfu_way].LFU_count = 1;
79 }
80
81
82 /*
83 * 根据当前的 delta 确定下一次最可能选择的 next_delta
84 */
85 static int get_l1d_next_best_transition (int delta) {
86   // 计算在当前 set 中 LFU_count 的总和 进而计算概率
87   // 并且找到最大的 LFU 值和对应的 way
88   int set_LFU_sum = 0;
89   int max_LFU = -1;
90   int max_LFU_way = 0;
91   for(int j = 0; j < L1D_DELTA_CACHE_WAYS; j++) {
92     int lfu_count = L1D_Delta_Cache[delta][j].LFU_count;
93     set_LFU_sum += lfu_count;
94     if(lfu_count > max_LFU) {
95       max_LFU = lfu_count;
96       max_LFU_way = j;
97     }
98   }
99   // 如果最大概率低于 1/3 则返回无效值 -1
100  if(L1D_Delta_Cache[delta][max_LFU_way].LFU_count * 3 < set_LFU_sum) {
101    return -1;
```

```
102   }
103   return L1D_Delta_Cache[delta][max_LFU_way].next_delta;
104 }
105
106 void CACHE::l1d_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t cache_hit,
        uint8_t type)
107 {
108   uint64_t block = addr >> LOG2_BLOCK_SIZE;
109   uint64_t page = addr >> PAGE_SIZE_OFFSET;
110   int page_index = page % L1D_PAGE_CACHE_SETS;
111   // 判断当前的 page 是否在 Page Cache 中 hit
112   int page_way;
113   for(page_way = 0; page_way < L1D_PAGE_CACHE_WAYS; page_way++) {
114     if(L1D_Page_Cache[page_index][page_way].page_tag == get_l1d_page_tag(page)){
115       break;
116     }
117   }
118
119   int new_delta = 1 + L1D_DELTA_CACHE_SETS / 2;
120   int page_offset = (addr >> WORD_SIZE_OFFSET) % (L1D_DELTA_CACHE_SETS / 2);
121
122   // 如果 page_way!= L1D_PAGE_CACHE_WAYS 则证明 page 在 Page Cache 中 hit
123   if(page_way != L1D_PAGE_CACHE_WAYS) {
124     int last_delta = L1D_Page_Cache[page_index][page_way].last_delta;
125     int last_offset = L1D_Page_Cache[page_index][page_way].last_offset;
126     // 计算新的 delta
127     new_delta = page_offset - last_offset + L1D_DELTA_CACHE_SETS / 2;
128     // 更新 l1d Delta Cache
129     update_l1d_delta_cache(last_delta, new_delta);
130   }
131
132   // 进行预取
133   int next_delta = new_delta;
134   uint64_t next_addr = addr;
135   for(int i = 0, prefetch_count = 0; i < 128 && prefetch_count < L1D_PREFETCH_DEGREE;
        i++) {
136     // 获取当前 next_delta 对应的最好的预取策略
137     int best_delta = get_l1d_next_best_transition(next_delta);
138     // 如果找不到合适的 delta 则退出
139     if(best_delta == -1) {
140       break;
141     }
142     // 预取在 delta cache 构成的 Markov 图中概率超过 1/3 的节点
143     else {
144       // 由于概率超过 1/3 的节点不可能超过2个
145       int candidate_way[2];
146       int max_LFU[2] = {0};
147       // 计算在当前 set 中 LFU_count 的总和 进而计算概率
148       int set_LFU_sum = 0;
```

```
149         for(int j = 0; j < L1D_DELTA_CACHE_WAYS; j++) {
150           int lfu_count = L1D_Delta_Cache[next_delta][j].LFU_count;
151           set_LFU_sum += lfu_count;
152           if(lfu_count > max_LFU[0]) {
153             max_LFU[0] = lfu_count;
154             candidate_way[0] = j;
155           }
156           else if(lfu_count > max_LFU[1]) {
157             max_LFU[1] = lfu_count;
158             candidate_way[1] = j;
159           }
160           else ;
161         }
162         // 接下来判断前两个候选是否满足要求
163         for(int j = 0; j < 2; j++) {
164           // 如果满足预取条件 则进行预取
165           if(max_LFU[j] * 3 > set_LFU_sum) {
166             // 计算预取地址
167             uint64_t pref_addr = ((next_addr >> WORD_SIZE_OFFSET)
168                 + (L1D_Delta_Cache[next_delta][candidate_way[j]].next_delta -
                        L1D_DELTA_CACHE_SETS / 2))
169                 << WORD_SIZE_OFFSET;
170             uint64_t pref_block = pref_addr >> LOG2_BLOCK_SIZE;
171             uint64_t pref_page = pref_addr >> PAGE_SIZE_OFFSET;
172             // 判断预取的 block 是否在当前 page 中 并且确实需要进行预取
173             if((page == pref_page) && (block != pref_block)) {
174               prefetch_line(ip, addr, pref_addr, FILL_L1, 0);
175               prefetch_count++;
176               if(prefetch_count == L1D_PREFETCH_DEGREE) {
177                 break;
178               }
179             }
180           }
181         }
182       }
183
184       // 向前走一步
185       next_delta = best_delta;
186       uint64_t pref_addr = ((next_addr >> WORD_SIZE_OFFSET)
187           + (best_delta - L1D_DELTA_CACHE_SETS / 2))
188           << WORD_SIZE_OFFSET;
189       next_addr = pref_addr;
190     }
191
192     // 如果在 Page Cache 中 miss
193     // 则需要根据 NRU 逐出一项
194     int evict_page_way = -1;
195     if(page_way == L1D_PAGE_CACHE_WAYS) {
196       // 选择 NRU_bit 为0的项
```

```
197     for(int i = 0; i < L1D_PAGE_CACHE_WAYS; i++) {
198       if(L1D_Page_Cache[page_index][i].NRU_bit == 0) {
199         evict_page_way = i;
200         break;
201       }
202     }
203     // 如果所有都为1 则反转 NRU_bit
204     if(evict_page_way == -1) {
205       evict_page_way = 0;
206       for(int i = 0; i < L1D_PAGE_CACHE_WAYS; i++) {
207         L1D_Page_Cache[page_index][i].NRU_bit = 0;
208       }
209     }
210   }
211
212   // 如果 page 在 Page Cache 中 hit 则需要刷新 Page Cache 中的 last_delta
213   // 如果 page 在 Page Cache 中 miss 则需要将逐出项对应的 last_delta 清空
214   // 对这两种情况 都需要更新 last_offset page_tag 和 NRU_bit
215   if(page_way != L1D_PAGE_CACHE_WAYS) {
216     L1D_Page_Cache[page_index][page_way].last_delta = new_delta;
217     L1D_Page_Cache[page_index][page_way].last_offset = page_offset;
218     L1D_Page_Cache[page_index][page_way].page_tag = get_l1d_page_tag(page);
219     L1D_Page_Cache[page_index][page_way].NRU_bit = 1;
220   }
221   else {
222     L1D_Page_Cache[page_index][evict_page_way].last_delta = 0;
223     L1D_Page_Cache[page_index][evict_page_way].last_offset = page_offset;
224     L1D_Page_Cache[page_index][evict_page_way].page_tag = get_l1d_page_tag(page);
225     L1D_Page_Cache[page_index][evict_page_way].NRU_bit = 1;
226   }
227 }
```

3.2 ReD 替换策略的实现

Javier 等人 [1] 通过实验研究发现，并不是所有访问的数据块都需要缓存在 LLC 中，由于 LLC 中的局部性原理不如更高层 cache，因此 LLC 中的数据块的访问频次相对较低。对于一些数据块，如果不考虑其被重复访问的概率而贸然加入到 LLC 中，可能会导致 LLC 的污染，进而由于占据了 LLC 的空间，使得本应该加入到 LLC 的数据块需要通过替换才能够进入。

基于上述的实验探究，笔者根据 Javier 等人提出算法原理，复现了一种在 LLC 中基于 Reuse Detection 的 bypass 策略。这种策略的核心思想是，基于保存的历史访问次数来预测一条指令第一次访问数据时后续重复访问的概率。对于概率较低的指令，即使在 LLC 中并没有命中，也不会将其加入到 LLC 中。接下来笔者将会详细介绍 ReD 替换策略的实现细节。

在 ReD 策略中，维护了一个 Address Reuse Table(ART)，用来存储最近在 LLC 中 miss 的数据地址。当一条指令访问一个数据块的时候，这个数据块在 LLC 和 ART 中的命中情况共有以下两种：

1. 这个数据块在 LLC 和 ART 中都没有命中，则说明这条指令是对于这个数据块的第一次访问 (initial request)，而对于当前只访问过一次的数据块，很有可能在后续并不会被重复访问，因此

并不将其加入的 LLC 中，只缓存在 ART 中，这也就实现了 ReD 的 bypass 机制。

2. 这个数据块在 ART 中命中，而在 LLC 中没有命中，则说明这条指令是对该数据块的第一次重复访问 (first-reused), 该数据块很有可能在后续被重复访问，因此将其加入到 LLC 中。

在 ART 中，笔者沿用了论文中给出的默认参数，采用了组相联的方式，使用数据块的地址作为索引，共有 512 个 set，16 路，并且采用 FIFO 的机制在每一个 set 中进行替换。同时降低硬件开销，每 4 个相邻的 block 共用同一个 ART entry，在 entry 中通过 page address tag 和 valid 位进行区分标记。ART 结构如图3.6所示。

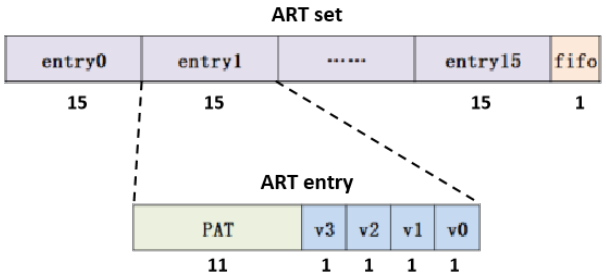

图 3.6: ART 结构示意图

在 ReD 策略中，如果只采用 ART 机制的话，则会导致一个重复访问的块会遇到两次 LLC miss，这是因为 initial request 和 first-reused request 都会在 LLC 中 miss。为了减少这种开销，ReD 假设数据块的重复访问与 initial request 相关。基于此假设，ReD 引入了一个使用 PC 作为索引的 PC Reuse Table (PCRT)，其中包含两个域段 #reused 和 #not_reused。PCRT 结构如图3.7所示。

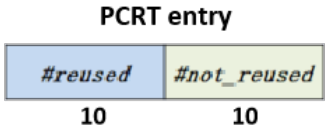

图 3.7: PCRT 结构示意图

在 PCRT 中，笔者沿用了论文中给出的默认参数，表中共有 256 个表项。

在 ReD 中，为了能够提高 ART 的效率，维护了一个 ART Sample Set (ARTSS) 进行采样。对于 PCRT 的计算并没有使用全部的 ART，而是采取 25% 的抽样比例，通过 ARTSS 进行计算。此外为了能够提高 ART 的效率，对于复用率低于 1/64 或者复用率高于 1/4 的情况，并不将其插入到 ART 中。通过上述机制，降低了插入概率之后，能够有效保证 ART 中数据的有效性。算法流程如图3.8所示。

3.3 SHiP++ 替换策略

SHiP 替换策略 [5] 的核心思想是，使用一种基于签名的 cache 替换策略，来预测基于当前签名的插入操作是否会在将来被重新访问。一种很朴素的想法就是如果当前在 LLC 中的插入操作对应的签名是 re-referenced 的话，那么这个插入操作在之后仍有可能 re-reference。

为了能够很好地学习到这种 re-referenced 的模式，SHiP 替换策略提出了一种历史签名记录表 (SHCT)，SHCT 使用签名作为索引，当 cache line 命中的时候，其签名对应的表项会进行递增，而

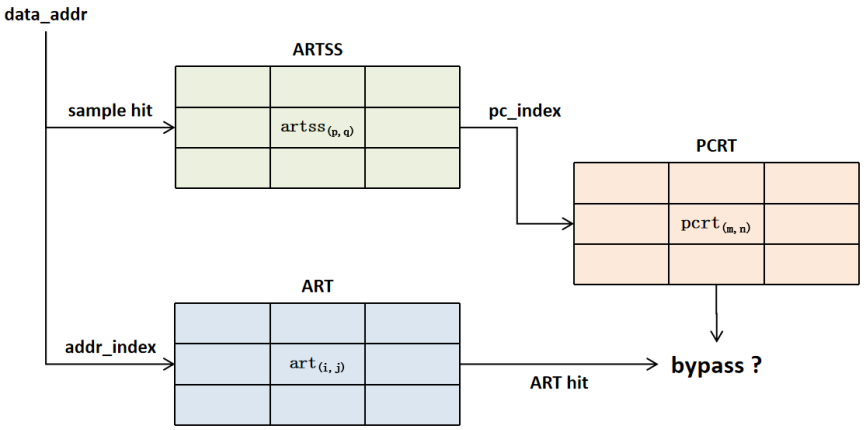

图 3.8: PCRT 结构示意图

当 cache line 被逐出的时候，其签名对应的表项会递减。其中，SHCT 对应的表项为 0 的时候，说明当前的 cache line 有较远的重新引用间隔。

SHiP 替换策略是在 SRRIP 的基础上引入了 SHCT，来辅助对于 RRPV 表的赋值操作。对于一个在 cache line 中 miss 的操作，将其加入到 RRPV 中的时候，其初始值的赋值需要参考 SHCT。如果其签名对应的 SHCT 表项的值足够大，说明其重新引用间隔较小，因此不应该将其尽快逐出，所以给 RRPV 表项赋最大阈值减 1，而对于 SHCT 表项为 0 的情况，则说明其重新引用距离很大，因此应该尽快逐出，给 RRPV 表项赋最大阈值，当 cache line 命中的时候，其对应的 RRPV 项被赋值为 0。在真正确定逐出元素的时候，就是在 RRPV 表项中找等于最大阈值的表项逐出，如果都不符合条件则全部递增 1，知道找到符合条件的表项。

在 SHiP 策略的基础上，SHiP++[4] 提出了 5 点优化改进。其中 1、3、5 项主要贡献在对于 RRPV 的更新上，实现了对 RRPV 更细粒度的赋值操作。2、4 项主要贡献在对于 SHCT 的更新上，实现了对于 SHCT 更细粒度的赋值操作。

1. cache 插入的提升不同于 SHiP 中，对于 SHCT 表项为 0 的给 RRPV 赋最大阈值，不为 0 的给 RRPV 赋最大阈值减 1，SHiP++ 认为当新插入的 cache line 对应的 SHCT 表项达到最大值的时候，极有可能在最近发生重复引用，因此只需要在第一次插入的时候将 RRPV 的值直接置零即可。但是需要区分预取操作，对于预取操作赋值为 1.

2. SHCT 训练的提升不同于 SHiP 中，每次发生 cache 命中的时候都对 SHCT 表项进行递增，每次 cache line 被逐出的时候 SHCT 表项都递减，SHiP++ 中仅在首次命中的时候对 SHCT 表项进行递增，来避免出现计数不均的情况。

3. 写回敏感的 RRPV 更新通常来说写回操作的重复引用间隔会很大，因此将其加入 cache 中会浪费可用资源，因此在 SHiP++ 中对于写回操作，一律将其 RRPV 项赋最大阈值。

4. 预取敏感的 SHCT 训练不同于 SHiP 中对于预取操作的 cache 访问不做区分，SHiP++ 中考虑到了预取操作与重新引用之间存在的差距，引入了 prefetch 标记位进行区分，并且在计算签名的时候将其纳入考虑。因此在 SHCT 表中预取操作和正常的重复引用就被区分开来，并且在对 RRPV 进行赋值的时候也能够进行区分。

5. 预取敏感的 RRPV 更新不同于 SHiP 在每次 cache 命中的时候都会将 RRPV 置为 0，SHiP++ 考虑到预取操作的特殊性，为每一个 cache line 增加了一个预取位，来表示其是否为预取块。当

一个正常的重复引用命中了一个预取块的时候，重置预取位并设置 RRPV 为 3。而当预取操作命中该块的时候则将 RRPV 置零。其余情况则不需要对 RRPV 进行更新。

3.4 ReD 与 SHiP++ 混合策略

由于 ReD 策略关注于一个 cache line 是否应该被加入到 LLC 中，而 SHiP++ 策略关注的是如何确定 cache line 的优先级。因此两个策略能够很好地结合在一起工作，最终实验选择采用 ReD 与 SHiP++ 混合策略作为 LLC 的替换策略。即当加入一个新的 cache line 的时候，首先用 ReD 策略判断当前的 cache line 是否应该被纳入到替换策略的考虑中来，如果不满足条件，则可以直接进行 bypass，如果需要纳入考虑，则通过 SHiP++ 策略进行管理，而后续进行 cache 替换以及状态更新的操作都采用 SHiP++ 提供的策略。

算法具体实现细节如下：

ReD 算法实现细节

```
#define LLC_BLOCK_OFFSET 6
#define LLC_WORD_OFFSET 2

// ============================== 定义 ART 相关数据 ==============================
#define ART_SETS 512
#define ART_WAYS 16
#define ART_PAT_BITS 11
#define ART_SECTOR_BLOCKS 4
#define ART_SET_FIFO_BITS 4

struct ART_Entry {
    uint16_t pat;
    bool valid[ART_SECTOR_BLOCKS];
};

struct ART_Set {
    struct ART_Entry entrys[ART_WAYS];
    uint8_t fifo_bits;
};


// ============================== 定义 ART Sampled set 相关数据 ==============================
#define ART_SAMPLED_SET_SETS 128
#define ART_SAMPLED_SET_WAYS 16
#define ART_SAMPLED_SET_BLOCKS 4
#define ART_SAMPLED_SET_PC_BITS 8

struct ART_SAMPLED_SET_Entry {
    uint8_t pc_indexes[ART_SAMPLED_SET_BLOCKS];
};


```

```
33 // ========================================== 定义 PCRT 相关数据 ==========================================
34 #define PCRT_SIZE 256
35
36 struct PCRT_Entry {
37     uint16_t not_reused;
38     uint16_t reused;
39 };
40
41
42 class ReD_Replacement {
43 private:
44     struct ART_Set ART[ART_SETS];
45     struct ART_SAMPLED_SET_Entry
           ART_SAMPLED_SET[ART_SAMPLED_SET_SETS][ART_SAMPLED_SET_WAYS];
46     struct PCRT_Entry PCRT[PCRT_SIZE];
47     int misses = 0;
48 public:
49     void initialize() {
50         // 初始化 ART
51         for(int i = 0; i < ART_SETS; i++) {
52             ART[i].fifo_bits = 0;
53             for(int j = 0; j < ART_WAYS; j++) {
54                 ART[i].entrys[j].pat = 0;
55                 for(int k = 0; k < ART_SECTOR_BLOCKS; k++) {
56                     ART[i].entrys[j].valid[k] = 0;
57                 }
58             }
59         }
60         // 初始化 ART Sampled set
61         for(int i = 0; i < ART_SAMPLED_SET_SETS; i++) {
62             for(int j = 0; j < ART_SAMPLED_SET_WAYS; j++) {
63                 for(int k = 0; k < ART_SAMPLED_SET_BLOCKS; k++) {
64                     ART_SAMPLED_SET[i][j].pc_indexes[k] = 0;
65                 }
66             }
67         }
68         // 初始化 PCRT
69         for(int i = 0; i < PCRT_SIZE; i++) {
70             PCRT[i].reused = 3;
71             PCRT[i].not_reused = 0;
72         }
73
74         misses = 0;
75     }
76
77     bool ART_find_block(uint64_t pc, uint64_t block) {
78         misses++;
79         // 首先确定 block 对应的 set 索引和 sector 索引
```

```
80          uint64_t set_index = (block / ART_SECTOR_BLOCKS) % ART_SETS;
81          uint16_t sector_index = block % ART_SECTOR_BLOCKS;
82          // 计算 page tag
83          uint16_t pat = (block / (ART_SECTOR_BLOCKS * ART_SETS)) % ART_SETS;
84          // 查询是否在 ART 中 hit
85          for(int i = 0; i < ART_WAYS; i++) {
86              if(ART[set_index].entrys[i].pat == pat
87                  && ART[set_index].entrys[i].valid[sector_index] == 1) {
88                  if(set_index % ART_SECTOR_BLOCKS == 0) {
89                      uint64_t pc_index = ART_SAMPLED_SET[set_index /
                            ART_SECTOR_BLOCKS][i].pc_indexes[sector_index];
90                      PCRT[pc_index].reused++;
91                      // 如果发生溢出 则折半
92                      if(PCRT[pc_index].reused > 1023) {
93                          PCRT[pc_index].reused /= 2;
94                          PCRT[pc_index].not_reused /= 2;
95                      }
96                      // 使得对应的 valid 位无效
97                      ART[set_index].entrys[i].valid[sector_index] = 0;
98                  }
99                  return 1;
100             }
101         }
102         return 0;
103     }
104
105     void ART_add_block(uint64_t pc, uint64_t block) {
106         uint64_t set_index = (block / ART_SECTOR_BLOCKS) % ART_SETS;
107         uint16_t pat = (block / (ART_SECTOR_BLOCKS * ART_SETS)) % ART_SETS;
108         uint16_t sector_index = block % ART_SECTOR_BLOCKS;
109         uint64_t pc_index = (pc >> LLC_WORD_OFFSET) % (1 << ART_SAMPLED_SET_PC_BITS);
110
111         // 查询在 ART 中是否命中
112         int where;
113         int way;
114         for(way = 0; way < ART_WAYS; way++) {
115             if(ART[set_index].entrys[way].pat == pat) {
116                 break;
117             }
118         }
119         // 如果命中
120         if(way != ART_WAYS) {
121             // 有效位置为 1
122             ART[set_index].entrys[way].valid[sector_index] = 1;
123             if(set_index % ART_SECTOR_BLOCKS == 0) {
124                 ART_SAMPLED_SET[set_index /
                        ART_SECTOR_BLOCKS][way].pc_indexes[sector_index] = pc_index;
125             }
126         }
```

```
127         // 如果没有命中
128         else {
129             where = ART[set_index].fifo_bits;
130             if(set_index % ART_SECTOR_BLOCKS == 0) {
131                 for(int j = 0; j < ART_SECTOR_BLOCKS; j++) {
132                     uint64_t evict_pc_index = ART_SAMPLED_SET[set_index /
                            ART_SECTOR_BLOCKS][where].pc_indexes[j];
133                     PCRT[evict_pc_index].not_reused++;
134                     if(PCRT[evict_pc_index].not_reused > 1023) {
135                         PCRT[evict_pc_index].reused /= 2;
136                         PCRT[evict_pc_index].not_reused /= 2;
137                     }
138                 }
139             }
140             // 进行替换
141             ART[set_index].entrys[where].pat = pat;
142             for(int j = 0; j < ART_SECTOR_BLOCKS; j++) {
143                 if(j == sector_index) {
144                     ART[set_index].entrys[where].valid[j] = 1;
145                 }
146                 else {
147                     ART[set_index].entrys[where].valid[j] = 0;
148                 }
149             }
150             if(set_index % ART_SECTOR_BLOCKS == 0) {
151                 ART_SAMPLED_SET[set_index /
                        ART_SECTOR_BLOCKS][where].pc_indexes[sector_index] = pc_index;
152             }
153             ART[set_index].fifo_bits++;
154             if(ART[set_index].fifo_bits == ART_WAYS) {
155                 ART[set_index].fifo_bits = 0;
156             }
157         }
158     }
159
160     bool bypass(uint64_t full_addr, uint64_t ip, uint32_t type) {
161         // 首先确定访问的 block 和 pc 对应的索引值
162         uint64_t block = full_addr >> LLC_BLOCK_OFFSET;
163         // pc 取字节对齐后的低8位
164         uint64_t pc_index = (ip >> LLC_WORD_OFFSET) % (1 << ART_SAMPLED_SET_PC_BITS);
165         if(type == LOAD || type == RFO || type == PREFETCH) {
166             // 如果在 ART 中不能找到对应的 block 则需要考虑进行 bypass
167             // 如果在 ART 中命中了 则不进行 bypass
168             if(!ART_find_block(ip, block)) {
169                 // 当满足了一定的 reused ratio 的时候
170                 // 对于 reused ratio 极低的情况考虑不加入 ART
171                 // 否则需要提前将本应该直接 bypass 的指令加入到 ART 中
172                 if(PCRT[pc_index].reused * 64 > PCRT[pc_index].not_reused
173                     && PCRT[pc_index].reused * 3 < PCRT[pc_index].not_reused
```

```
                    || (misses % 8 == 0)) {
                    ART_add_block(ip, block);
                }
                // 对于 reused ratio 较低的情况 考虑为 bypass
                if(PCRT[pc_index].reused * 3 < PCRT[pc_index].not_reused) {
                    return true;
                }
            }
        }
        return false;
    }
};
```

4 硬件开销

Strategy	Level	Part	Size(KB)
Pangloss	L1D	Delta Cache	34.8
		Page Cache	11.5
	L2C	Delta Cache	3.8
		Page Cache	9.2
	Total		59.4
ReD	LLC	ART	15.6
		ARTSS	8
		PCRT	0.6
	Total		24.2
SHiP++	LLC	RRPV	8
		is_pref	4
		SHCT	6
		Sample Set	1.875
		Sample Set ID	0.125
	Total		20
ToTal			103.6

表 1: 综合策略硬件开销

5 实验分析

在本次实验中，笔者尝试将不同的策略进行组合，并在给定的两个 trace 上进行模拟测试，比较不同组合之间性能差异，下表是实验的组合情况以及平均 IPC 对比，如表2所示。

Id	Prefetch			Replacement	IPC_{avg}
	L1D	L2C	LLC	LLC	
1	**no**	**no**	**no**	**lru**	**0.532**
2	**no**	**next_line**	**no**	**lru**	**0.703**
3	**no**	**ghb**	**no**	**lru**	**0.875**
4	**no**	**ghb**	**no**	**lfu**	**0.879**
5	**no**	**pangloss**	**no**	**lru**	**1.005**
6	**pangloss**	**pangloss**	**no**	**lru**	**1.063**
7	**pangloss**	**pangloss**	**no**	**red**	**1.099**
8	**pangloss**	**pangloss**	**no**	**shippp**	**1.111**
9	**pangloss**	**pangloss**	**no**	**shippp+red**	**1.127**

表 2: 实验对比整体概览

从上表的数据中我们可以看出，在预取方面的性能上，next_line<ghb<pangloss，在替换策略上，lru≈lfu<red<shippp<red+shippp。接下来笔者将采用更加具体的数据来分析带来性能提升的原因。

5.1 L2C 预取策略对比

在 L2C 上，我们实现了 next_line、GHB 和 pangloss 三种预取策略，并对这三种算法进行性能分析，分别测试了在各级 cache 上的命中率以及在 L2C 上预取的有效程。在进行实验对比的时候，L1D 和 LLC 上不进行数据预取，cache 替换策略选用 LRU，具体数据如表3所示。

L2C prefetcher		Hit_{L1D}	$Latency_{L1D}$	Hit_{L2C}	$Prefetch_{L2C}$	Hit_{LLC}	IPC
no	**trace1**	80.67%	193.92ns	29.35%	0	29.34%	0.49799
	trace2	90.01%	150.28ns	22.45%	0	12.63%	0.56547
next_line	**trace1**	81.28%	140.86ns	37.53%	99.99%	29.34%	0.53251
	trace2	90.41%	67.64ns	48.25%	98.69%	12.69%	0.87342
GHB	**trace1**	81.81%	56.71ns	73.79%	99.99%	29.62%	0.87291
	trace2	90.56%	68.97ns	67.04%	78.01%	11.56%	0.87656
pangloss	**trace1**	81.79%	46.72ns	84.88%	99.99%	29.34%	0.87301
	trace2	90.68%	39.84ns	86.12%	67.91%	12.64%	1.13762

表 3: L2C 预取策略对比

从数据中可以看出，随着我们算法的提升，IPC 在逐步提高。GHB 策略是对 next_line 的补充优化，能够根据历史信息来选择进行预取，由于补充了历史信息，因此这种预测相对于 next_line 会更加准确。pangloss 则是对 GHB 的大幅度优化改进，除了借鉴了 GHB 中有关历史信息的缓存，基于历史访问序列来预测，还引入了马尔科夫链模型来估计下一次访问的可能性。此外采用了访问序列间隔作为索引信息能够更好用有限的空间资源保存更多的历史信息。从数据中可以看出，从 next_line 策略到 pangloss 策略，L2C 的 cache 命中率逐渐提升，并且 L1D 由于 miss 产生的 latency 逐渐降低。这样的收益就是由 L2C 上合理的预取策略带来的。由于在 L2C 中进行了数据预取，因此能够在数据还没有被访问的时候提前取到 L2C 中，使得在时候的访问时原本会产生 miss 的数据能够在 cache 中命中，此外当 L1D 发生 miss 的时候，需要的 L2C 中查询，通过预取能够大大提升这类查询的命中率。从数据中也能够看出，性能提升主要是由 L1D 产生 miss 时的 latency 降低带来的。其实验结果可视化如图5.9所示。

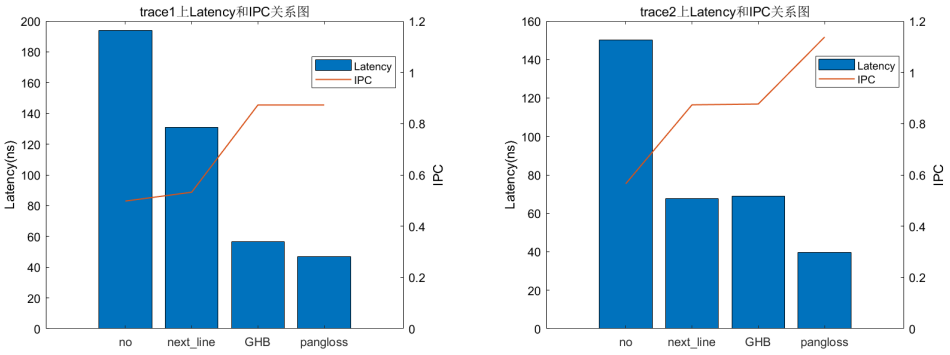

图 5.9: L1D Latency 和 IPC 关系图

除此之外，为了进一步证明预取策略的有效性，基于 Champsim 提供的实验数据，计算了 L2C 数据预取的有效性，如图5.10所 hi。从图像中能够充分证明，pangloss 策略的预取有效性是最高的，因此能够保证将合理的数据取入到 cache 中，既不会导致 cache 资源的浪费，又能够提高 cache 的命中率。

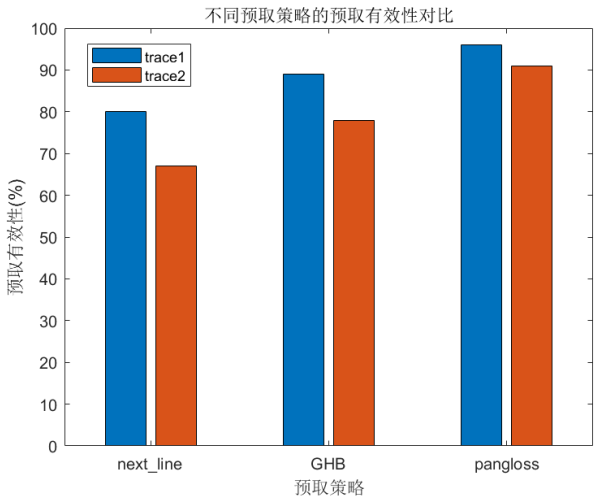

图 5.10: 不同预取策略预取有效性对比

5.2 预取策略在多级 cache 中的应用对比

在本次实验中，笔者除了在 L2C 中实现了 cache 预取策略，还在 L1D 中也增加了预取策略，在此实验设定时，选用的预取策略均为 pangloss，选用的替换策略均为 LRU，实验配置及平均 IPC 如表4所示。

ID	L1D	L2C	LLC	IPC
1	no	pangloss	no	1.005
2	pangloss	pangloss	no	1.063

表 4: 不同层级数据预取配置及 IPC 结果对比

此外，实验中还进一步分析了不同层级中的 cache 命中率，实验结果的对比如图5.11所示。从图像中可以看出，在 L1D 中增加了数据预取之后，能够极大提升 L1D 的 cache 命中率，虽然在 trace1 上 L2C 的命中率有所降低，但是 L1D 带来的性能提升已经非常显著，这样的下降不足以影响最终的性能提升。通过数据对比，能够得出结论，在 L1D 上实现数据预取带来的性能提升收益要更高，这样的结论很容易理解，因为 L1D 对应的数据访问会非常频繁，因此更应该注重提升 L1D 的 cache 命中率。

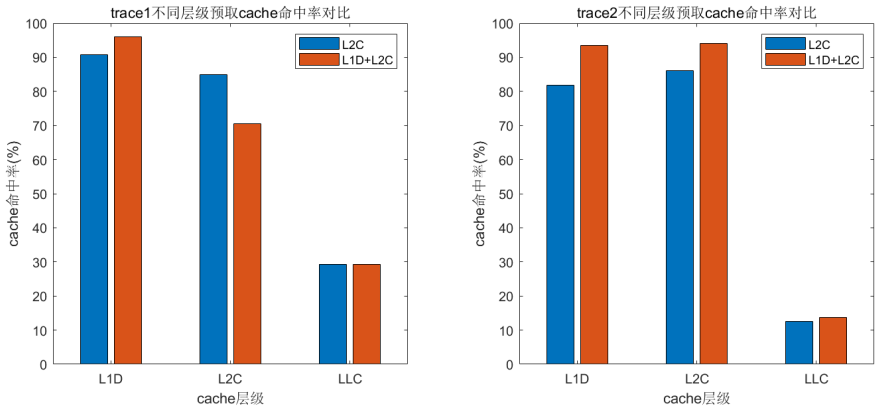

图 5.11: 不同层级数据预取 cache 命中率对比

5.3 LRU 和 LFU 对比

在本次实验中，笔者还以 GHB 作为预取策略基础，在 LLC 中分别采用 LRU 和 LFU 替换策略进行对比，得到数据如表5所示。

L2C prefetcher		L1D_Hit	L2C_Hit	LLC_Hit	IPC
LRU	**trace1**	81.81%	73.79%	29.62%	0.87291
	trace2	90.56%	67.04%	11.56%	0.87656
LFU	**trace1**	81.78%	73.71%	29.58%	0.87244
	trace2	90.57%	67.04%	12.78%	0.88562

表 5: LRU 和 LFU 替换策略对比

由于 LFU 是对 LRU 的近似，因此无论是各级 cache 的命中率还是最终的 IPC 上来看，两者的差距都不大，因此能够证明 LFU 是对于 LRU 的一种简化近似，并且能够保证结果的一致性。

5.4 LLC 替换策略对比

在本次实验中,笔者在 LLC 中实现了多种替换策略,包含 LRU、ReD、SHip++ 以及 ReD+SHiP++,为了对比分析这四种替换策略的性能差异，笔者选定在 L1D 和 L2C 上都使用 pangloss 预取策略，对比了这四种替换策略的性能差异。实验结果如表6所示。

LLC Replacement		L1D_Hit	L2C_Hit	LLC_Hit	IPC
LRU	**trace1**	93.44%	70.46%	29.34%	0.91293
	trace2	96.07%	93.99%	13.66%	1.21385
SHiP++	**trace1**	91.51%	65.85%	39.05%	0.92178
	trace2	95.89%	81.72%	37.43%	1.27791
ReD	**trace1**	91.54%	49.74%	5.83%	0.94639
	trace2	95.88%	81.62%	35.04%	1.27301
ReD+SHiP++	**trace1**	92.17%	59.61%	5.85%	0.97381
	trace2	95.92%	83.42%	35.55%	1.27632

表 6: 不同替换策略对比

从表中可以看出，SHiP++ 策略已经取得了一定的性能提升，原因是采用了更加细粒度的 cache 更新策略。而 ReD 性能略优于 SHiP++，其原因是 ReD 中在 LLC 上做了 bypass，将一些初次访问到的数据从 LLC 中 bypass，降低了 LLC 中被污染的概率。最终采用了 ReD+SHiP++ 的混合策略能够极大提高 L1D 的命中率，虽然在 LLC 上的命中率有所下降，但是 L1D 的访问是十分频繁的，因此对于 L1D 命中率的提升带来的性能收益是最高的。

6 总结

本次实验，根据实验基本要求，基于 Champsim，在 L2C 上完成了 GHB 数据预取策略，在 LLC 上完成了基于 LFU 的 cache 替换策略。通过调整 GHB 中的相关参数，在给定的两个 trace 上，平均 IPC 达到 **1.02**。除了完成了实验基本要求外，笔者还进行了额外的探究，根据最近在数据预取和 cache 替换相关的比赛以及发表的论文，学习并复现了 Pangloss 数据预取策略，SHiP++ 和 ReD 的 cache 替换策略。此外，笔者还对 Champsim 提供的相关算法和新实现的算法之间进行了组合测试，并对比了不同组合策略之间的性能差异，综合对比之后，笔者最终采用了在 L1D 和 L2C 中实现了 Pangloss 数据预取策略，在 LLC 中实现了 SHiP++ 配合 ReD 的 cache 替换策略，并调整了其中的参数，在给定的两个 trace 上，平均 IPC 达到了 **1.127**，达到了无数据预取基于 LRU 的 cache 替换策略的方法的 **2.12** 倍，但是这种提高了 IPC 的策略必定是通过消耗较大的硬件资源换取的，硬件开销达到了 103.6KB。本项目的详细代码位于GitHub。

参考文献

[1] Javier Díaz Maag, Pablo Enrique Ibáñez Marín, Teresa Monreal Arnal, Víctor Viñals Yúfera, and José M Llaberia Griñó. Red: A policy based on reuse detection for demanding block selection in last-level caches. In *The Second Cache Replacement Championship: workshop schedule*, pages 1–4, 2017.

[2] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96. IEEE, 2004.

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

[4] A. Jaleel V. Young, C.-C. Chou and M. Qureshi. Ship++: Enhancing signature-based hit predictor for improved cache performance. In *The Second Cache Replacement Championship: workshop schedule*.

[5] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, 2011.