



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

计算机体系结构期末实验报告

cache 替换策略和数据预取

卢麒萱

学号：2010519

专业：计算机科学与技术

2023 年 1 月 4 日

目录

| | |
|--------------------------|----------|
| 一、 概述 | 1 |
| (一) 预取器 | 1 |
| (二) cache 替换策略 | 1 |
| 二、 GHB 步长预取器 | 1 |
| (一) 实验原理 | 1 |
| (二) 具体实现 | 2 |
| 三、 cache 替换策略 | 4 |
| (一) 实验原理 | 4 |
| 1. LFU | 4 |
| 2. MRU | 4 |
| 3. MFU | 4 |
| (二) 具体实现 | 4 |
| 1. LFU | 4 |
| 2. MRU | 5 |
| 3. MFU | 6 |
| 4. 策略结合 | 7 |
| 四、 性能测试及分析 | 7 |
| (一) score | 8 |
| (二) 其他性能指标 | 10 |
| (三) 总结 | 10 |

一、 概述

(一) 预取器

本次实验我在 L2 cache 上实现了 GHB 步长预取器，并通过更改多种 cache 替换策略来测试组合性能。

(二) cache 替换策略

本次实验我在 LLC 上实现了 3 个 cache 替换策略：

1. LFU
2. MRU
3. MFU

并通过多种策略搭配，更改预取算法来测试组合性能。

二、 GHB 步长预取器

(一) 实验原理

步长预取 (stride prefetching) 检测并预取连续访问之间相隔 s 个缓存数据块的数据，其中 s 即是步长的大小。硬件实现需要使用访问预测表，记录访问的地址，步长以及访存指令的 pc 值。

GHB prefetcher : Global History Buffer, 将 cache miss 的资料暂存，根据这些资料选择较佳的 prefetching method。使用 FIFO，可避免 stable data，提高精准度。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。

它由两个主要结构组成：

1. 索引表 (Index Table, IT): 一个通过程序属性 (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。
2. 全局历史缓冲区 (GHB): 一个循环队列，存储 L2 缓存观察到的缓存行地址序列。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。

具体结构如图1所示。

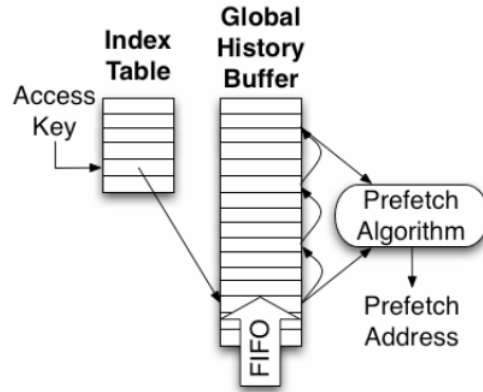


Figure 1b: Global History Buffer Prefetch Structure

图 1: GHB 预取器结构

基于 GHB 的步长预取器算法过程：对于每个二级缓存访问（命中和未命中），该算法使用访问的 PC 索引到 IT 中，并将缓存行地址（例如 A）插入 GHB。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。如果两个步长相等（步长为 d），预取器只向缓存行 $A+ld$ 、 $A+(l+1)d$ 、 $A+(l+2)d$ 、 \dots 、 $A+(l+n)d$ 发出预取请求，其中 l 是事先设定好的预取 look-ahead，n 是度。

（二） 具体实现

结构体定义出索引表 IT 和全局历史缓冲区 GHB，编号皆为 0-255，设置初始编号为 NLL = 256，代表当前未使用。

首先初始化设置当前 GHB 块的前置项，同时更新 IT 表。然后顺着 GHB 往前查找是否至少有三次 miss，若成立则计算三次地址之间相邻步长是否相等，相等则预取。

GHB_stride.h

```

1  #include "cache.h"
2
3  #define GHB_SIZE 256
4  #define IT_SIZE 256
5  #define look_head 1
6  #define degree 20
7  #define NLL 256
8
9  uint64_t cur_idx = 0;
10
11 struct IT_entry {
12     uint32_t ghb_ptr = NLL;
13 } IT[IT_SIZE];
14
15 struct GHB_entry {
16     uint64_t addr;
17     uint32_t prev = NLL;
18 } GHB[GHB_SIZE];

```

```

19
20 void ghb_init(uint64_t addr, uint64_t ip) {
21     if(IT[ip % IT_SIZE].ghb_ptr == NLL)
22         GHB[cur_idx].prev = NLL;
23     else
24         GHB[cur_idx].prev = IT[ip % IT_SIZE].ghb_ptr;
25
26     GHB[cur_idx].addr = addr >> LOG2_BLOCK_SIZE;
27     IT[ip % IT_SIZE].ghb_ptr = cur_idx;
28 }
29
30 bool find3miss() {
31     uint64_t second_idx = GHB[cur_idx].prev;
32     if(second_idx != NLL && GHB[second_idx].prev != NLL)
33         return 1;
34     return 0;
35 }
36
37 uint64_t check_stride() {
38     uint64_t second_idx = GHB[cur_idx].prev;
39     uint64_t third_idx = GHB[second_idx].prev;
40     uint64_t s1 = GHB[cur_idx].addr - GHB[second_idx].addr;
41     uint64_t s2 = GHB[second_idx].addr - GHB[third_idx].addr;
42     if(s1 == s2)
43         return s1;
44     return 0;
45 }

```

GHB 步长预取器

```

1 ...
2 uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t
   cache_hit, uint8_t type, uint32_t metadata_in)
3 {
4     ghb_init(addr, ip);
5     uint64_t s;
6     if(find3miss() && (s = check_stride())) {
7         for(int i = look_head; i <= look_head + degree; i++) {
8             uint64_t new_addr = (GHB[cur_idx].addr + i * s) <<
               LOG2_BLOCK_SIZE;
9             prefetch_line(ip, addr, new_addr, FILL_L2, 0);
10        }
11    }
12
13    cur_idx = (cur_idx + 1) % GHB_SIZE;
14    return metadata_in;
15 }
16 ...

```

三、cache 替换策略

(一) 实验原理

1. LFU

LFU(least-frequently used), 即最不频繁使用, 所以, LFU 算法会淘汰掉使用频率最低的数据。如果存在相同使用频率的数据, 则再根据使用时间间隔, 将最久未使用的数据淘汰。

LFU 也需要使用类似 LRU 的数据结构实现, 不过 LFU 算法淘汰数据是基于使用频率的, 所以, 我们需要快速找到同一频率的所有节点, 然后按照需要淘汰掉最久没被使用过的数据。所以, 首先我们要有一个 hash 表来存储每个频次对应的所有节点信息, 同时为了保证操作效率, 节点与节点之间同样要组成一个双向链表。

hash 表中的 key 表示访问次数, value 就是一个双向链表, 链表中所有节点都是被访问过相同次数的数据节点。相比较于 LRU 算法中的节点信息, LFU 算法中节点的要害中除了包含具体的 key 和 value 之外, 还包含了一个 freq 要素, 这个要素就是访问次数, 同 hash 表中的 key 值一致。这样做的好处是当根据 key 查找得到一个节点时, 我们可以同时得到该节点被访问的次数, 从而得到当前访问次数的所有节点。还有一个重要功能, 就是如何根据 key 值获取 value 值。参考 LRU 算法的数据结构, 还需要有一个 hash 表来存储 key 值与节点之间的对应关系。

2. MRU

MRU (most-recently used), 即最近最常用, 当某组的行已满, 需要替换时, 选择替换最近最少用的那行。理念与 LRU 正好对应相反。

3. MFU

MFU(most-frequently used), 即最频繁使用, MFU 算法会淘汰掉使用频率最高的数据。理念与 MRU 正好对应相反。

(二) 具体实现

1. LFU

通过分析先有的 LRU 算法, 发现在 block.h 中已经定义了链表等数据结构以及记录页面留存时间的元素 lru, 因此只需添加一个记录访问次数的元素 ref_count, 在替换页面后更新 lru 位值的 lru_update() 函数中将访问的页面的 ref_count 也每次顺带加 1。在寻找替换页面时顺序遍历找到 ref_count 为最小值的页面进行替换即可。

LFU cache 替换算法

```
1 uint32_t CACHE::lru_victim(uint32_t cpu, uint64_t instr_id, uint32_t set,
2   const BLOCK *current_set, uint64_t ip, uint64_t full_addr, uint32_t type)
3 {
4     uint32_t way = 0;
5     uint32_t new_way = 0;
6     // fill invalid line first
7     for (way=0; way<NUM_WAY; way++) {
8         if (block[set][way].valid == false) {
9             DP ( if (warmup_complete[cpu]) {
```

```

10         cout << "[" << NAME << "]" " << __func__ << " instr_id: " <<
        instr_id << " invalid set: " << set << " way: " << way;
11     cout << hex << " address: " << (full_addr >> LOG2_BLOCK_SIZE) << "
        victim address: " << block[set][way].address << " data: " <<
        block[set][way].data;
12     cout << dec << " lru: " << block[set][way].lru << endl; });
13     new_way = way;
14     break;
15 }
16 }
17 BLOCK blk = block[set][0];
18 // LFU victim
19 if (way == NUM_WAY) {
20     for (way=0; way<NUM_WAY; way++) {
21         if (block[set][way].ref_count < blk.ref_count) {
22             DP ( if (warmup_complete[cpu]) {
23                 cout << "[" << NAME << "]" " << __func__ << " instr_id: " <<
                instr_id << " replace set: " << set << " way: " << way;
24                 cout << hex << " address: " << (full_addr >> LOG2_BLOCK_SIZE)
                << " victim address: " << block[set][way].address << "
                data: " << block[set][way].data;
25                 cout << dec << " lru: " << block[set][way].lru << endl; });
26                 blk = block[set][way];
27                 new_way = way;
28             }
29         }
30     }
31     if (new_way == NUM_WAY) {
32         cerr << "[" << NAME << "]" " << __func__ << " no victim! set: " << set
        << endl;
33         assert(0);
34     }
35     return new_way;
36 }

```

2. MRU

LRU 的替换算法中总是寻找第一个 $lru == NUM_WAY - 1$ （最不频繁使用）的页面进行替换，则 MRU 总是寻找第一个 $lru == 0$ （最近频繁使用）的页面进行替换。

由于主要代码框架相同，只展示判断条件不同处的部分代码。

MRU cache 替换算法

```

1  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)
2  {
3      ...
4      // MRU victim
5      if (way == NUM_WAY) {

```

```

6   for (way=0; way<NUM_WAY; way++) {
7       if (block[set][way].lru == 0) {
8           DP ( if (warmup_complete[cpu]) {
9               cout << "[" << NAME << "]" " << __func__ << " instr_id: " <<
                instr_id << " replace set: " << set << " way: " << way;
10              cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) <<
                " victim address: " << block[set][way].address << " data: "
                << block[set][way].data;
11              cout << dec << " lru: " << block[set][way].lru << endl; });
12
13              break;
14          }
15      }
16  }
17  ...
18  }

```

3. MFU

LFU 的替换算法中总是寻找访问次数最少的页面进行替换, 则 MFU 总是寻找访问次数最多的页面进行替换。

MFU cache 替换算法

```

1  uint32_t CACHE::lfu_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)
2  {
3      ...
4      // MFU victim
5      if (way == NUM_WAY) {
6          for (way=0; way<NUM_WAY; way++) {
7              if (block[set][way].ref_count > blk.ref_count) {
8                  DP ( if (warmup_complete[cpu]) {
9                      cout << "[" << NAME << "]" " << __func__ << " instr_id: " <<
                          instr_id << " replace set: " << set << " way: " << way;
10                     cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) <<
                          " victim address: " << block[set][way].address << " data: "
                          << block[set][way].data;
11                     cout << dec << " lru: " << block[set][way].lru << endl; });
12                     blk = block[set][way];
13                     new_way = way;
14                 }
15             }
16         }
17         ...
18     }

```


4. 策略结合

本实验我尝试几种将两种置换策略结合的置换策略：

1. LRU+LFU
2. MRU+LFU
3. LRU+MFU
4. MRU+MFU

以下以 MRU+LFU 为例，选择 lru 值和 ref_count 值同时较小的页面进行置换，其他方法通过改变大小于符号类似。

MRU+LFU cache 替换算法

```

1  uint32_t CACHE::lfu_victim(uint32_t cpu, uint64_t instr_id, uint32_t set,
2      const BLOCK *current_set, uint64_t ip, uint64_t full_addr, uint32_t type)
3  {
4      ...
5      // LFU+MRU victim
6      if (way == NUM_WAY) {
7          for (way=0; way<NUM_WAY; way++) {
8              if (block[set][way].lru <= blk.lru && (block[set][way].ref_count <=
9                  blk.ref_count)) {
10
11                  DP ( if (warmup_complete[cpu]) {
12                      cout << "[" << NAME << "]" " << __func__ << " instr_id: " <<
13                          instr_id << " replace set: " << set << " way: " << way;
14                      cout << hex << " address: " << (full_addr>>LOG2_BLOCK_SIZE) <<
15                          " victim address: " << block[set][way].address << " data: "
16                          << block[set][way].data;
17                      cout << dec << " lru: " << block[set][way].lru << endl; });
18                      blk = block[set][way];
19                      new_way = way;
20                  }
21              }
22          }
23      }
24      ...
25  }

```

四、性能测试及分析

本次实验只讨论在 L2 cache 上实现预取算法，在 LLC 上实现 cache 替换策略，以获得最明显的指标数据。已经测试，在其他层上加入预取算法和替换策略同样会使分数提高，为了控制变量数目，在此不做讨论。

且由于给定测试数据只有两个，由数据特定性产生的结果对 LRU/MRU、LFU/MFU 两组相反策略影响较大，因此只做这两个数据集上的分析，最后再由多组数据跑总分来判定最优策略。

表 1: 组合策略 score

| replacement policies \ prefetchers | no | next_line | GHB_stride |
|------------------------------------|----------|-----------|------------|
| LRU | 0.528585 | 0.700725 | 0.936405 |
| MRU | 0.57464 | 0.743885 | 0.957595 |
| LFU | 0.53228 | 0.70871 | 0.94113 |
| MFU | 0.59585 | 0.76382 | 0.9623 |

本次实验的预取器有 no、next_line、GHB_stride 三种，cache 替换策略有 LRU、MRU、LFU、MFU 五种，以及叠加组合策略四种，首先生成 12 种非叠加模型。

首先以 cumulative IPC 作为性能指标进行评分，通过运行 score.py 产生两组数据测试的平均 score，生成图表进行直观对比，再根据分析结果在数据集上运行模型展开其他数据指标的分析。

(一) score

部分运行截图如下：

```
python ./score.py bin/perceptron-no-next_line-no-lfu-icore traces
bin_path: bin/perceptron-no-next_line-no-lfu-icore
trace file path: traces
please wait...
traces/462.libquantum-714B.champsimtrace.xz score: 0.532450
traces/482.sphinx3-1100B.champsimtrace.xz score: 0.884970
avg score: 0.708710
```

nextline 预取 + LFU cache 替换

```
python ./score.py bin/perceptron-no-next_line-no-mru-icore traces
bin_path: bin/perceptron-no-next_line-no-mru-icore
trace file path: traces
please wait...
traces/462.libquantum-714B.champsimtrace.xz score: 0.549430
traces/482.sphinx3-1100B.champsimtrace.xz score: 0.938340
avg score: 0.743885
```

nextline 预取 + MRU cache 替换

```
python ./score.py bin/perceptron-no-GHB_stride-no-mfu-icore traces
bin_path: bin/perceptron-no-GHB_stride-no-mfu-icore
trace file path: traces
please wait...
traces/462.libquantum-714B.champsimtrace.xz score: 0.914420
traces/482.sphinx3-1100B.champsimtrace.xz score: 1.010180
avg score: 0.962300
```

GHB 步长预取 + MFU cache 替换

```
python ./score.py bin/perceptron-no-GHB_stride-no-lfu-icore traces
bin_path: bin/perceptron-no-GHB_stride-no-lfu-icore
trace file path: traces
please wait...
traces/462.libquantum-714B.champsimtrace.xz score: 0.887720
traces/482.sphinx3-1100B.champsimtrace.xz score: 0.994540
avg score: 0.941130
```

GHB 步长预取 + LFU cache 替换

综合数据绘制表格如表1，柱状图如图2。

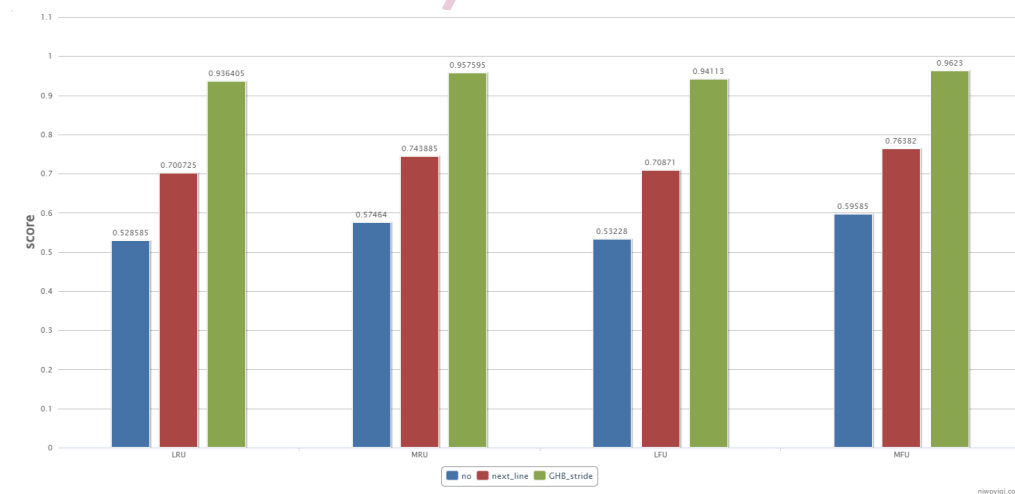


图 2: 组合策略跑分结果

表 2: 叠加替换策略 score

| LRU | MRU | LFU | MFU | LRU+LFU | LRU+MFU | MRU+LFU | MRU+MFU |
|----------|----------|---------|--------|---------|---------|---------|---------|
| 0.936405 | 0.957595 | 0.94113 | 0.9623 | 0.93849 | 0.96075 | 0.96196 | 0.95706 |

分析图表可知, 预取算法中, 性能 $GHB_stride > next_line > no$, 不同预取策略的选取对于性能有显著影响。

而 cache 替换策略中, 在测试数据集上, 性能 $MFU > MRU > LFU > LRU$, 但不同策略的选取对于分数影响并不大。由于本次测试数据集只有两个, 推测不同替换策略受数据影响较大。通过查阅资料发现, LFU/MFU 算法都不常用, 需要一直保存统计每块使用次数的大常数, 并且每次选择替换页时需要遍历所有页, 或者额外使用排序算法, 相对于 LRU/MRU 会消耗更多 CPU 资源。

综上推测 LRU/MRU 算法适合较大的文件, LFU/MFU 算法适合较小的文件和零碎的文件。

提交助教测试多组数据后得到总分结果如图3。由此可以初步认定, cache 替换策略中, 对于大部分数据, 性能 $LFU > LRU > MFU > MRU$ 。

| | |
|-----|----------|
| mfu | 4.26327 |
| lfu | 4.345788 |
| mru | 4.247778 |
| lru | 4.344702 |

图 3: 多数据汇总跑分结果

以下测试四种叠加 cache 替换策略, 固定预取算法为 GHB_stride 。得到表格如表2, 柱状图如图4。

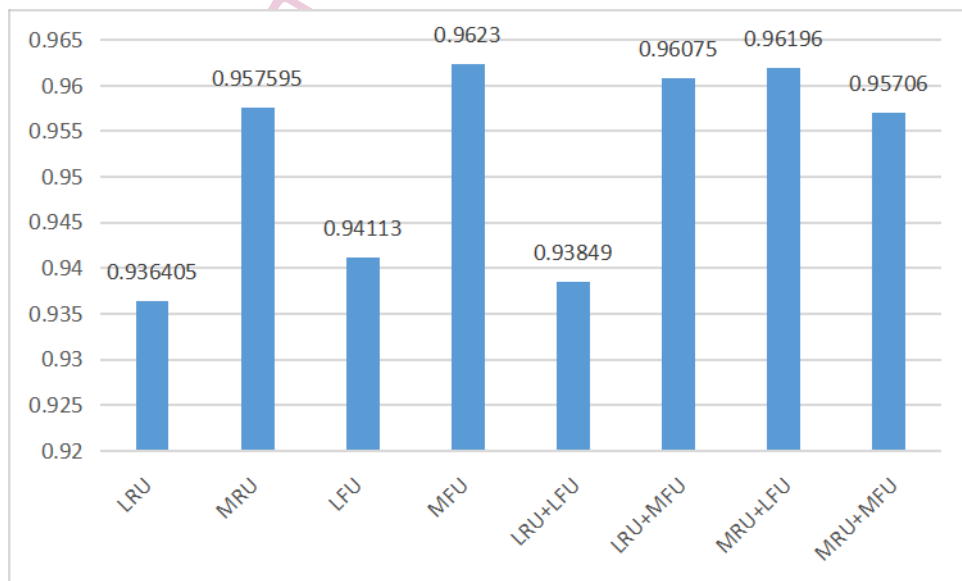


图 4: 叠加替换策略跑分结果

分析图表可知, LRU + LFU/MFU 相比 LRU 性能有所提升, MRU + LFU 相比 MRU/LFU

表 3: GHB_stride+MFU L2 性能指标

| total_access | total_hit | total_miss | loads | load_hit | load_miss | RFOs | RFO_hit | RFO_miss | prefetches | prefetch_hit |
|---------------|------------|---------------|----------------|--------------------|-----------------|-----------------|------------------|---------------|----------------------|--------------|
| 9645814 | 8320620 | 1325194 | 1325137 | 1129164 | 195973 | 12 | 0 | 12 | 7754948 | 6625739 |
| prefetch_miss | writebacks | writeback_hit | writeback_miss | prefetch_requested | prefetch_issued | prefetch_useful | prefetch_useless | prefetch_late | average_miss_latency | |
| 1129209 | 565717 | 565717 | 0 | 83502678 | 48975933 | 3739405 | 128 | 160306 | 267.96096 | |

表 4: GHB_stride+MFU LLC 性能指标

| total_access | total_hit | total_miss | loads | load_hit | load_miss | RFOs | RFO_hit | RFO_miss | prefetches | prefetch_hit |
|---------------|------------|---------------|----------------|--------------------|-----------------|-----------------|------------------|---------------|----------------------|--------------|
| 1890911 | 96080 | 1794831 | 35669 | 1737 | 33932 | 12 | 0 | 12 | 1289513 | 61402 |
| prefetch_miss | writebacks | writeback_hit | writeback_miss | prefetch_requested | prefetch_issued | prefetch_useful | prefetch_useless | prefetch_late | average_miss_latency | |
| 1228111 | 565717 | 32941 | 532776 | 0 | 0 | 555 | 3656775 | 0 | 225.08416 | |

性能有所提升，而 MRU + MFU 相比 MRU/MFU 性能降低了，总体来看，MFU 单独的性能还是最优的，LRU 可以叠加 MFU 获得相对较大的性能提升。但鉴于性能提升幅度与之前提到的 LFU 和 MFU 消耗内存的短板相较好处并不大，叠加方法还是尽量在文件较小的情况下使用。

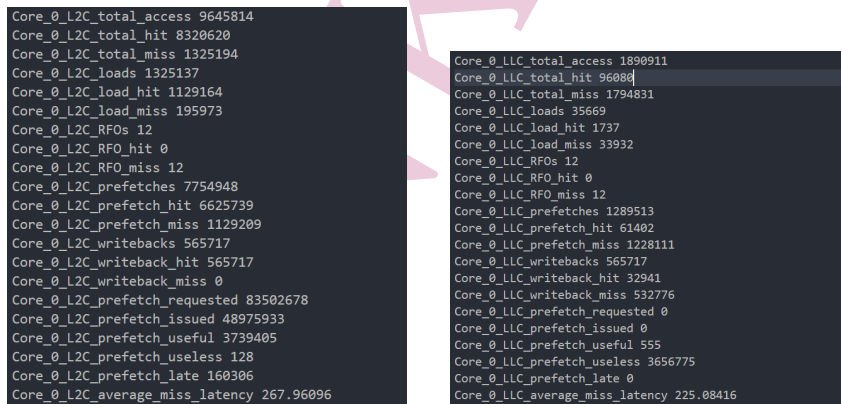
(二) 其他性能指标

此处选择在测试数据上表现最优的两个模拟器在 462 数据集上分别进行评测进行简略分析。预取器都使用 GHB_stride（对于 MFU 使用了 no prefetcher 作分析 L2 hit 的对比），cache 替换策略使用 MFU、MRU+LFU。

对于三组实验都使用 100M 条指令预热，模拟 50M 条指令。

MFU 评测输出截图如图5，绘制表格如表3、4。

图 5: MFU 的表现



| | |
|---|---|
| Core_0_L2C_total_access 9645814 | Core_0_LLC_total_access 1890911 |
| Core_0_L2C_total_hit 8320620 | Core_0_LLC_total_hit 96080 |
| Core_0_L2C_total_miss 1325194 | Core_0_LLC_total_miss 1794831 |
| Core_0_L2C_loads 1325137 | Core_0_LLC_loads 35669 |
| Core_0_L2C_load_hit 1129164 | Core_0_LLC_load_hit 1737 |
| Core_0_L2C_load_miss 195973 | Core_0_LLC_load_miss 33932 |
| Core_0_L2C_RFOs 12 | Core_0_LLC_RFOs 12 |
| Core_0_L2C_RFO_hit 0 | Core_0_LLC_RFO_hit 0 |
| Core_0_L2C_RFO_miss 12 | Core_0_LLC_RFO_miss 12 |
| Core_0_L2C_prefetches 7754948 | Core_0_LLC_prefetches 1289513 |
| Core_0_L2C_prefetch_hit 6625739 | Core_0_LLC_prefetch_hit 61402 |
| Core_0_L2C_prefetch_miss 1129209 | Core_0_LLC_prefetch_miss 1228111 |
| Core_0_L2C_writebacks 565717 | Core_0_LLC_writebacks 565717 |
| Core_0_L2C_writeback_hit 565717 | Core_0_LLC_writeback_hit 32941 |
| Core_0_L2C_writeback_miss 0 | Core_0_LLC_writeback_miss 532776 |
| Core_0_L2C_prefetch_requested 83502678 | Core_0_LLC_prefetch_requested 0 |
| Core_0_L2C_prefetch_issued 48975933 | Core_0_LLC_prefetch_issued 0 |
| Core_0_L2C_prefetch_useful 3739405 | Core_0_LLC_prefetch_useful 555 |
| Core_0_L2C_prefetch_useless 128 | Core_0_LLC_prefetch_useless 3656775 |
| Core_0_L2C_prefetch_late 160306 | Core_0_LLC_prefetch_late 0 |
| Core_0_L2C_average_miss_latency 267.96096 | Core_0_LLC_average_miss_latency 225.08416 |

同时得到 no prefetcher + MFU 在 462 数据集上测试结果的 L2 部分输出如表5。

可以直观地看到，hit 率从 29.92% 上涨到 83.26% 左右，预取的 hit 率也有 85.44%，但平均 miss 访问时间也有相应的增加，但总体性能提升幅度非常大。

MRU+LFU 测试结果的 LLC 部分输出如表6。

分析图表可知，在 LLC 上，MFU 的总体 hit 率为 5.08%，prefetch hit 率为 4.76%，MRU+LFU 的总体 hit 率为 4.07% 左右，prefetch hit 率为 5.47%。但 writeback hit 率前者为 5.82%，后者只有 0.78%，因此总 hit 率前者更高。但纵观两者性能相差不大。

(三) 总结

文末再对于算法使用的内存空间进行分析：

表 5: no+MFU L2 性能指标

| total_access | total_hit | total_miss | loads | load_hit | load_miss | RFOs | RFO_hit | RFO_miss | prefetches | prefetch_hit |
|---------------|------------|---------------|----------------|--------------------|-----------------|-----------------|------------------|---------------|----------------------|--------------|
| 1890866 | 565717 | 1325149 | 1325137 | 0 | 1325137 | 12 | 0 | 12 | 0 | 0 |
| prefetch_miss | writebacks | writeback_hit | writeback_miss | prefetch_requested | prefetch_issued | prefetch_useful | prefetch_useless | prefetch_late | average_miss_latency | |
| 0 | 565717 | 565717 | 0 | 0 | 0 | 0 | 0 | 0 | 172.75384 | |

表 6: MRU+LFU LLC 性能指标

| total_access | total_hit | total_miss | loads | load_hit | load_miss | RFOs | RFO_hit | RFO_miss | prefetches | prefetch_hit |
|---------------|------------|---------------|----------------|--------------------|-----------------|-----------------|------------------|---------------|----------------------|--------------|
| 1890911 | 76994 | 1813917 | 35650 | 2024 | 33626 | 12 | 0 | 12 | 1289532 | 70555 |
| prefetch_miss | writebacks | writeback_hit | writeback_miss | prefetch_requested | prefetch_issued | prefetch_useful | prefetch_useless | prefetch_late | average_miss_latency | |
| 1218977 | 565717 | 4415 | 561302 | 0 | 0 | 298 | 3683794 | 0 | 225.91018 | |

LRU 和 MRU 算法使用 lru 元素标志时间戳, 需要存储空间为 $set * NUM_WAY * \log(set * NUM_WAY)bits$ 。LFU 和 MFU 需要记录所有页的访问次数, 本次实验我使用 uint32_t 类型来记录, 需要存储空间 $set * NUM_WAY * 32bits$ 。

综合全部分析, 可以得出推论: 保存用于预测的历史信息越多, 预测可能就越准确, 但对应消耗 CPU 资源就越多。因此选取预取策略时, 要综合预取对象文件大小和 CPU 性能等多种指标已达到最好效果。