南開大學
Nankai University

计算机学院

体系结构期末报告

cache 替换策略和数据预取

姓名：邵琦

学号：2011188

专业：计算机科学与技术

2023 年 1 月 6 日

目录

1 实验要求

在本实验中，你将使用 ChampSim 模拟器 (ChampSim 是一个基于 trace 的微体系结构模拟器) 实现和评估不同的 L2 Cache 数据预取和 LLC cacheline 替换策略。

我们将为你提供一些程序的 trace，以及 ChampSim 的源代码。我们还将提供对应接口，你的工作是补全接口实现不同的预取算法和缓存替换策略，并评估其性能。

2 实验原理

2.1 为何要进行预取

这是由于处理器和内存之间的性能差异越来越大导致的。两者发展的侧重点不同，CPU 主要是要快，往更快执行任务发展；而内存的发展有些偏向于增加内存的密度，让内存变得更大。1985 年至 2010 年间，CPU 的性能提升的数千倍，可是内存相关的性能只提升了不到 10 倍。如果等 CPU 需要执行相关指令或者需要修改数据的时候再从内存中去读取，那么大部分时间都会花费在等待数据上，这是不可容忍的。这时预取的重要性就体现了，将将要访问的内容提前从内存搬移到 Cache 中，CPU 就可以即时拿到所需的内容，避免了等待。当然，如果预取做得不好，是有可能导致性能下降的，由于 Cache 的大小是很宝贵的，如果预取判断出错，预取的是无用的数据，然后反而把 Cache 中后续有可能还会用到的数据给 Evict 了，那么会增加系统的功耗，减低性能。

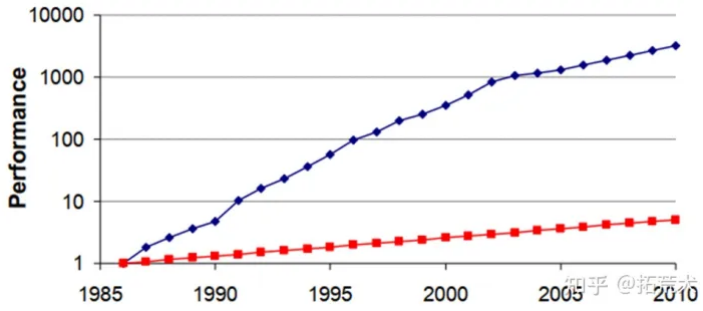

图 2.1: 为何要进行预取

2.2 硬件预取

2.2.1 考虑点

想要消除内存性能与 CPU 性能的巨大的 gap，相关的预取策略必须考虑以下三点：

1. 预取哪块数据：必须能准确地判断所需预取的数据，这个很好理解，无效数据对我们毫无用处，只会白白耗费电源，还有可能将缓存中有用的数据给踢出，从而导致性能下降。

2. 何时开始预取：如果预取不及时，甚至晚于需要相关数据的节点，那么这个策略毫无帮助，浪费硬件资源，预取过早亦同理。

3. 数据存放在哪：如果相关预取数据存放不合理，将会将后续需要用到的数据踢出 CPU，造成性能下降。

2.2.2 策略评估

在评价一个预取策略的好坏时，我们一般会从覆盖率和准确性两个维度来评判，好的预取策略必须同时在这两个指标上面有亮眼的成绩。

1. 覆盖率就是引入预取策略消除的 Cache Miss 次数与引入预取策略前的总的 Cache Miss 次数的比值。比如没有预取机制的时候会发生 100 次 Cache Miss，然后引入预取将 Cache Miss 减少到了 30 次，意味着该预取策略成功避免了 70 次 Cache Miss，即覆盖率为 70/100=70%。

2. 预取的正确率还需要考虑在这个过程中预取机制发出了几次预取操作，接着上面的例子，如果该预取机制总共发出了 140 次的预取操作，其中 70 次是有效的预取操作，那么正确率为 70/140=50%。

显然，覆盖率和准确性也是相互制约的。覆盖率的提高倾向于有更多的预取次数，而准确性的保证的限制了预取次数不能太高。我们应该在追求覆盖率的同时，尽量保证准确性。

3. 预取的及时性也是很需要考虑的一个指标。假设我们正确的判断出了要预取的数据，但是如果预取的时机不合适，也有可能适得其反。如果过早预取，则预取到的数据有可能在需要之前就被踢出缓存，还有可能占用缓存空间，造成 Cache Miss 的增加；如果过晚预取，则可能会造成较大的延时，CPU 需要等待预取后才能继续执行。

2.3 指令预取

2.3.1 next-line prefetcher

最简单的指令预取机制，维护了一个叫 stream buffer 的预取 buffer，每次 CPU 请求一个缓存行时，会从下一级 Cache 中将下一个缓存行读取到 stream buffer 中。只在指令顺序执行的时候表现比较好，当遇到分支或者函数调用时，预测效果可能会较差。

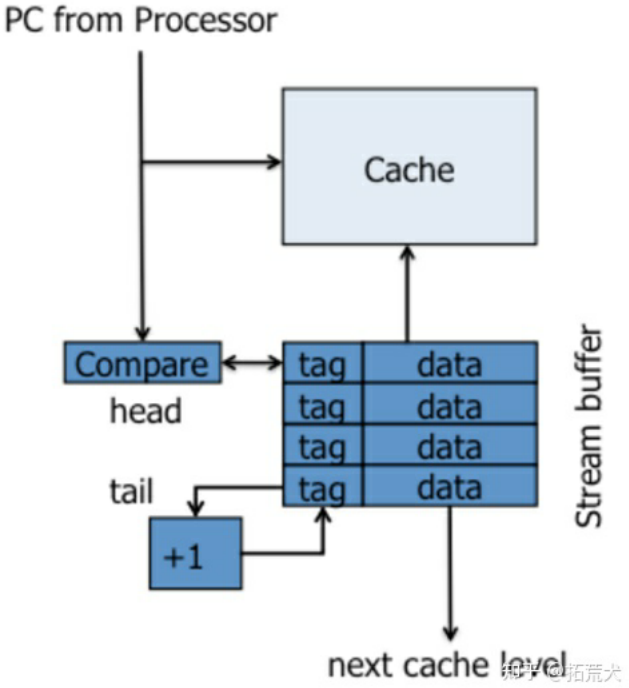

图 2.2: next-line prefetcher

2.3.2 FDIP(Fetch-directed instruction prefetching) prefetcher

这种预取机制针对分支指令进行了处理，大概思路就是预取指令分支预测的结果。该机制在分支预测模块和取指模块之间插入了一个 FTQ 的模块，然后会经过一个 filter 把一部分地址过滤掉（比如已经存在 ICache 中的地址），然后进入到 PIQ 中，这时就会将 PIQ 中的地址发往下一级 Cache，然后将取回的 Cache Line 放入一个全相联的 Cache 中。思考这种机制，它在分支预测之后预取，如果处理不好，预取的速度跟不上取指的速度，就会带来一定的延迟。

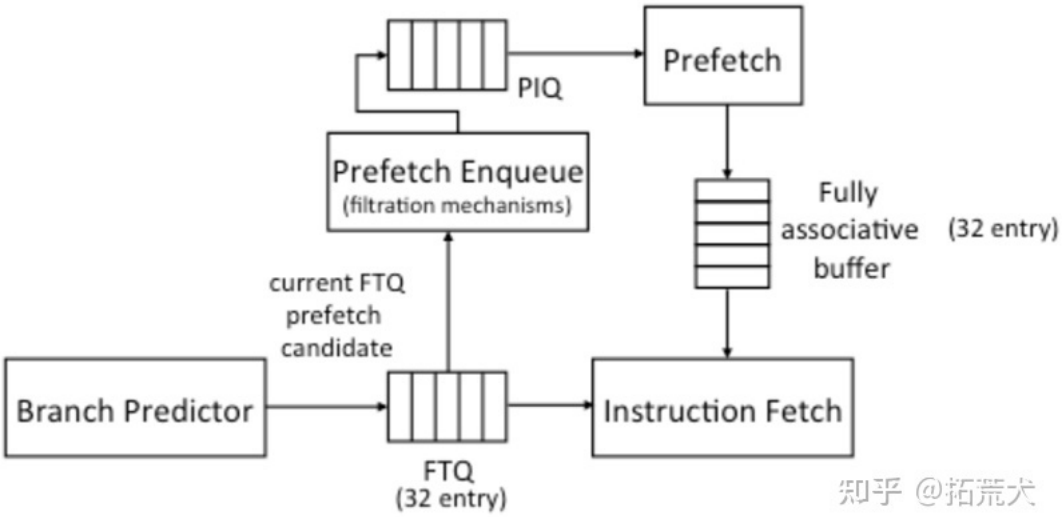

图 2.3: FDIP prefetcher

2.3.3 discontinuity prefetcher

不连续预取机制，这种机制就是把每次分支指令跳转的目标给记录一下，记录到一张表中。当下次再次访问到该分支指令时，在取指阶段查表，然后预取相应的缓存行就行，到分支目标指令取值时，指令已经被预取到缓存中。

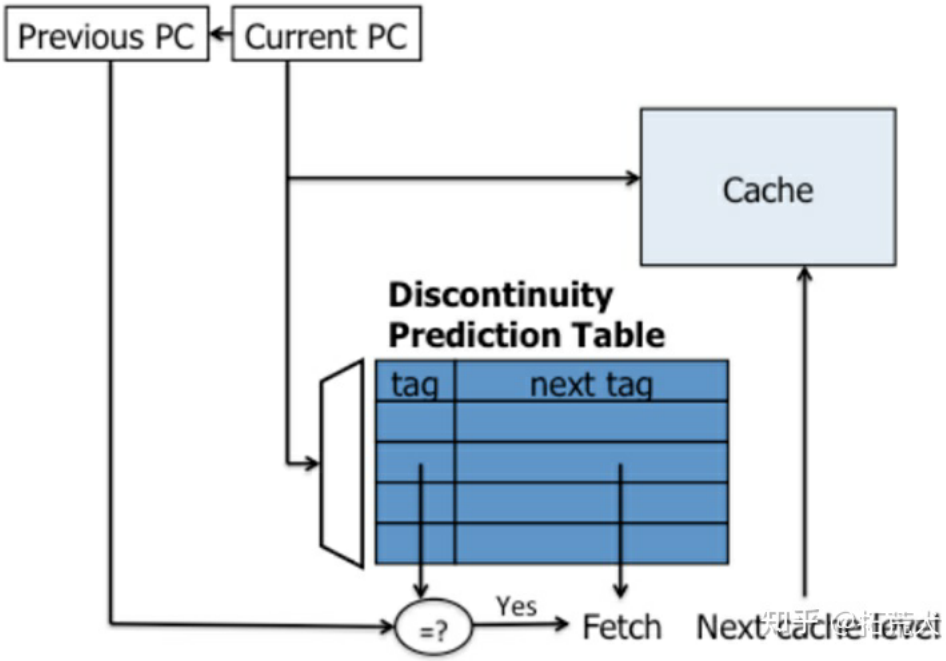

图 2.4: Caption

2.4 数据预取

2.4.1 传统基于表的预取

传统的基于表的预取机制通常使用一个表来记录历史信息，并根据表中记录的信息来指导预取的进行。表由键来索引，键常常是 PC 值或者访问数据的地址。这种预取机制分为两种方式，步幅预取 (stride prefetching) 和关联预取 (correlation prefetching)。

2.4.2 Stride Prefetching

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

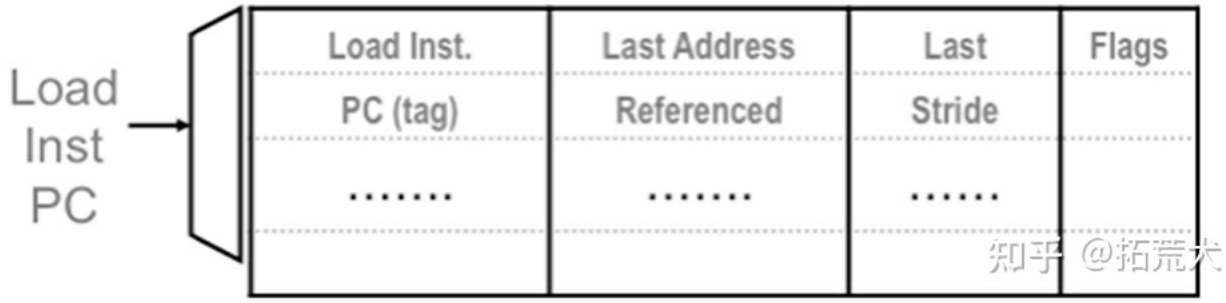

图 2.5: Stride Prefetching

2.4.3 Correlation Prefetching

比起步幅预取来说，关联预取可以处理更加复杂的地址访问模式。马尔可夫预取 (Markov Prefetch) 是一个使用历史表的关联预取算法。其中，预取键是导致缓存缺失的访问地址。历史表中的每个表项存储了历史中紧跟在对应预取键后的导致缓存缺失的访问地址的链表。当再次发生缓存缺失时，预取算法根据访存地址来索引历史表，然后从表项的链表中找出若干个置信度比较高的地址给预取了。

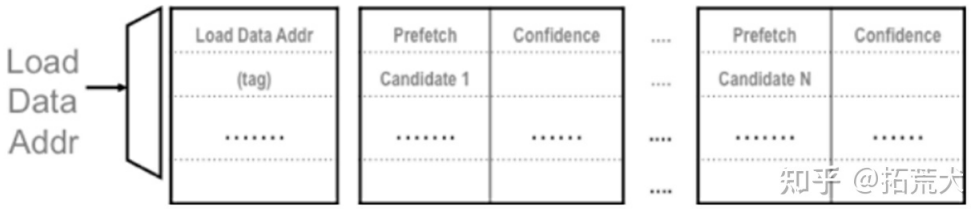

图 2.6: Correlation Prefetching

2.4.4 GHB 预取机制

传统的表预取有一些缺点，比如容易有过时的数据；预取键容易造成冲突；每个表项能存储的历史信息较少等不足。然后就有人发明出了基于 GHB 的预取机制，将预取键的匹配和历史信息的存储给解耦了。

传统的表预取有一些缺点，比如容易有过时的数据；预取键容易造成冲突；每个表项能存储的历史信息较少等不足。然后就有人发明出了基于 GHB 的预取机制，将预取键的匹配和历史信息的存储给解耦了。

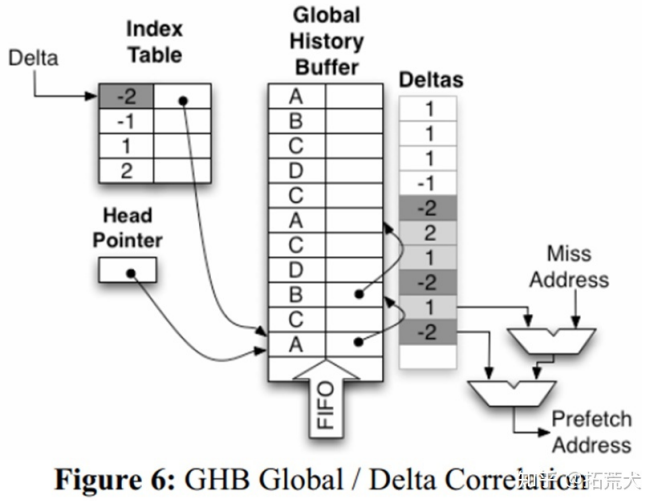

Figure 6: GHB Global / Delta Correlation

图 2.7: GHB 预取机制

GHB 预取机制的结构如上图所示，主要有两级组成：

1.Index table：索引表，这个表主要用于根据预取键来索引 GHB 缓冲区中的具体条目。这里预取键可以是 PC，也可以是访存地址。

2.Global history buffer：全局历史缓冲区，是一个循环队列。一共能存 n 项，其中每项存储一个地址以及一个指针。通过指针按时间顺序将一个预取键对应的地址给链起来。使用 GHB 也可以实现传统的步幅预取、马尔可夫预取、距离预取机制。

2.4.5 GHB 步幅预取

类似于传统的基于表的步幅预取，基于 GHB 的步幅预取的预取键也是 load 指令的 PC 值。预取器通过预取键来索引 GHB 中的条目，这里我们使每个预取键对应的 GHB 中的条目为预取键对应的 load 指令的访存地址。上边说过，这些地址形成了一个链，我们只需要取出链上的最后三个地址，计算出来对应的两个步幅，如果两个步幅相等，就可以像步幅预取那样预取新的缓存行了。

2.4.6 GHB 关联预取

GHB 关联预取也是类似的，它的预取键是实际的访存地址。预取键对应的 GHB 中的条目中存储了与预取键相同的地址。这样，当发生缓存缺失时，预取器根据缺失地址索引索引表，得到指向 GHB 条目的指针，然后访问得到的 GHB 条目链，链上每个元素在 GHB 中的直接后继元素即是预取候选。

2.5 cache 替换策略

Cache 的工作原理要求它尽量保存最新数据或者最近历史访问数据，由于 Cache 的容量限制和各种映射机制存在不足，在发生缺失时必然会产生替换。直接映射机制只将特定数据行换出；全相连和组相连映射从具有访问行数据存储权的若干特定行中按照替换机制选出一行进行换出。常见的替换算法有：随机替换算法，先进先出替换算法（FIFO，First In First Out）、最不经常使用替换算法（LFU，Least Frequency Used）、近期最少使用替换算法（LRU，Least Recently Used）

1. 随机替换算法

Cache 在发生替换时随机的选取一行换出。这种替换策略速度快、硬件实现简单，但是有可能替换最近访问行而降低 Cache 的命中率。

2. 先进先出替换策略（FIFO）

将替换规则与访问地址的历史顺序相关联，每次都把最先放入的一行数据块被替换出，但是该设计方法与 Cache 设计原理的局部性原理相悖，并不能反应数据块的使用频率。

3. 最不经常使用替换算法（LFU）

LFU 是替换最近一段时间内长期未被访问且访问频率最低的 Cache 行数据，这种策略是通过设置计数器来完成计数每个数据块的使用频率。每行设置一个从 0 开始的计数器，计数器在每次被访问时自增 1。当由于发生 Cache 缺失时，需要替换计数值最小行，同时清零这些行的计数器。这种替换算法只计数到两次特定时间内，不能严格反映出近期访问情况，与局部性原理不符。

4. 最近最少使用替换算法（LRU）

与 LFU 不同的是，LRU 算法是将最近一段时间内很久未访问过的 Cache 行换出，也是通过设置计数器来完成计数每个数据块的使用频率。每行含有一个计数器，Cache 在发生命中时对该命中行计数器清零，其它相关行的计数器自增加 1。如果 Cache 缺失，则替换计数值最大的行。LFU 可有效减小冲突缺失，保护最新数据行，提高 Cache 命中率。

3 实验代码

GHB 预取

```
#include "cache.h"
#include<stdio.h>

#define GHB_SIZE 256
#define IT_SIZE 256
unsigned int current_index = 0;

#define look_ahead 1     // 预取look_ahead
#define degree 10        // 度

struct GHB_ENTRY
{
    uint64_t cl_addr; // cache_line地址
    unsigned int num = 256;    // 编号最大255，记256为未使用
} GHB[GHB_SIZE];

struct IT_ENTRY
{
    unsigned int ghb_entry = 256;
} IT[IT_SIZE];

void CACHE::l2c_prefetcher_initialize()
{
    cout << "CPU " << cpu << " L2C ghb prefetcher" << endl;
}

uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t cache_hit,
    uint8_t type, uint32_t metadata_in)
```

```
{

    int num0=ip%IT_SIZE;
    GHB[current_index].num=IT[num0].ghb_entry;
    GHB[current_index].cl_addr=addr>>LOG2_BLOCK_SIZE;
    IT[num0].ghb_entry=current_index;

    //查找最近3次访问的cache的块号
    unsigned int most_1st_recent=current_index;
    unsigned int most_2nd_recent=GHB[most_1st_recent].num;
    unsigned int most_3rd_recent=GHB[most_2nd_recent].num;

    // 有3次Miss，步长相同，则预取
    if(most_2nd_recent!=256&&most_3rd_recent!=256)
    {
        uint64_t stride1=GHB[most_1st_recent].cl_addr-GHB[most_2nd_recent].cl_addr;
        uint64_t stride2=GHB[most_2nd_recent].cl_addr-GHB[most_3rd_recent].cl_addr;
        if(stride1==stride2)
        {
            for(int i=look_ahead;i<=look_ahead+degree;i++)
            {
                uint64_t addr0=((addr>>LOG2_BLOCK_SIZE)+i*stride1)<<LOG2_BLOCK_SIZE;
                prefetch_line(ip, addr, addr0, FILL_L2, 0);
            }
        }
    }
    current_index=(current_index+1)%GHB_SIZE;
    return metadata_in;
}

uint32_t CACHE::l2c_prefetcher_cache_fill(uint64_t addr, uint32_t set, uint32_t way,
    uint8_t prefetch, uint64_t evicted_addr, uint32_t metadata_in)
{
    return metadata_in;
}

void CACHE::l2c_prefetcher_final_stats()
{
   cout << "CPU " << cpu << " L2C ghb prefetcherfinal stats" << endl;
}
```

LFU cache 替换策略

```
#include "cache.h"

uint32_t counts[LLC_SET][LLC_WAY];

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

```
{
    for(int i=0;i<LLC_SET;i++)
    {
        for(int j=0;j<LLC_WAY;j++)
        {
            counts[i][j]=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 min=0;
    for (int i = 0; i < LLC_WAY; i++)
    {
        if (counts[set][i] < counts[set][min])
        {
            min = i;
        }
    }
    if(min<LLC_WAY)
    {
        return min;
    }
    return 0;
}

// called on every cache hit and cache fill
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)
{
    string TYPE_NAME;
    if (type == LOAD)
        TYPE_NAME = "LOAD";
    else if (type == RFO)
        TYPE_NAME = "RFO";
    else if (type == PREFETCH)
        TYPE_NAME = "PF";
    else if (type == WRITEBACK)
        TYPE_NAME = "WB";
    else
        assert(0);

    if (hit)
        TYPE_NAME += "_HIT";
    else
        TYPE_NAME += "_MISS";
```

traces/482	next$_{l}ine$	ip$_{s}tride$	ghb
lfu	0.979670	0.998600	1.039050
lru	0.869000	0.943780	1.000420
drrip	1.011670	1.072290	1.086190
srrip	0.933560	0.990670	1.042990

表 1: traces 482

traces/462	next$_{l}ine$	ip$_{s}tride$	ghb
lfu	0.552630	0.717140	0.902220
lru	0.532450	0.693400	0.876440
drrip	0.524300	0.662340	0.887090
srrip	0.525820	0.687360	0.891690

表 2: traces 462

```
54
55      if ((type == WRITEBACK) && ip)
56          assert(0);
57
58      // uncomment this line to see the LLC accesses
59      // cout << "CPU: " << cpu << "  LLC " << setw(9) << TYPE_NAME << " set: " <<
            setw(5) << set << " way: " << setw(2) << way;
60      // cout << hex << " paddr: " << setw(12) << paddr << " ip: " << setw(8) << ip <<
            " victim_addr: " << victim_addr << dec << endl;
61
62      // baseline LRU
63
64      if(hit)
65      {
66          counts[set][way]++;
67      }
68      else
69      {
70          counts[set][way]=1;
71      }
72  }
73
74  void CACHE::llc_replacement_final_stats()
75  {
76
77  }
```

4 实验结果

通过以上实验数据可以看出，对比三种预取器，可以发现 GHB 预取策略无论在哪个数据集上都有最高的分数和 IPC，而 next_line 性能最差，在 trace 462 上三个预取策略的差距尤为明显。

再对比不同的 cache 替换策略时，可以看到 lfu 替换策略在 trace 462 上是最优的 cache 替换策

avg score	next$_l ine$	ip$_s tride$	ghb
lfu	0.766150	0.857870	0.970635
lru	0.700725	0.818590	0.938430
drrip	0.767985	0.867315	0.986640
srrip	0.729690	0.839015	0.967340

表 3: avg score

略，但到了 trace 482 上 drrip 替换策略是最优的 cache 替换策略。可能原因可能是不同数据集中不同类型的指令占比不同，导致在 trace 482 上 lfu 并不能很好的提升性能，仅相较于 lru 策略有明显的性能提升。

由数据可以看出，GHB 预取器达到了很不错的性能优化效果，相较于 next_line 和 ip_stride 预取器都有很大的领先，很好的解决了 ip_stride 这些传统的表预取数据容易有过、预取键容易造成冲突、存储的历史信息较少等缺陷。而 cache 替换策略方面 lfu 替换策略并不能在所有的情况都能获得最好的性能，可能是因为数据集指令类型占比不同的原因。