南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

体系结构实验报告

---

cache 替换策略和数据预取

徐文斌

年级：2020 级

专业：计算机科学与技术

指导教师：李雨森

2023 年 1 月 7 日

摘要

在本实验中，我对 cache 数据预取和替换策略进行了一个大致的学习。使用 ChampSim 模拟器在 L2Cache 上实现了一个由多种预取机制组成的混合预取机制，以下简称为 GSDN。该机制一共由三种预取策略结合形成，分别是基于 GHB 的步长预取、基于 GHB 的 ip delta 预取以及 next line 预取。缓存替换算法方面，我在 LLC 上分别实现了 LFU 和 window-LFU 缓存替换算法。最终，经过测试，在实验提供的测试集上我所提出的混合预取机制结合 window-LFU 缓存替换策略相比于单纯使用 lru 缓存替换策略而不使用任何预取机制有着 1.9991 的 IPC 加速比。

关键字：ChampSim、预取、缓存替换、GHB、LFU

目录

一、 引言

1985 年至 2010 年间，CPU 的性能提升了数千倍，可是内存相关的性能只提升了不到 10 倍，CPU 和内存的执行速度出现了较大的差距。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。这就体现了预取的重要性，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，从而提升了 CPU 的性能。但是，如果预取做得不好，有可能导致性能的下降，由于 Cache 的大小有限，如果预取判断出错，预取的是无用的数据，反而把 Cache 中后续有可能还会用到的数据给驱逐了，那么会增加系统的功耗，减低性能。如果预取的不够及时，也会造成资源的浪费和性能的下降。因此，设计一个兼顾覆盖率、准确率和及时性的预取算法是一项比较有挑战性的、可以给计算机系统带来较大性能上的提升的工作。

此外，由于缓存大小有限，当缓存占满时，再次访问不存在缓存中的数据就需要将缓存中的缓存行给驱逐出缓存。一个好的缓存替换策略可以使得访问缓存的缺失率尽可能的小，从而提升 CPU 的整体性能。

本文中作者使用 ChampSim 模拟器，在 L2C 上实现了基于 GHB 的步长预取、基于 GHB 的 ip delta 预取以及 next line 预取相结合的混合预取机制，并在 LLC 上实现了 LFU、window-LFU 缓存替换策略。并对多种预取机制和缓存替换策略的组合进行了性能的测试和比较。

二、 相关工作

关于数据预取策略的设计已经有许多前人的非常优秀的研究。最基本的预取策略为预取顺序的缓存行，即预取紧跟当前正在访问的缓存行之后的地址的缓存行 [1]。但是这种预取机制在更加复杂的访问地址流上的性能可能并不是十分理想。更进一步的，有研究提出使用表来记录缓存的历史访问信息，从而便于对程序的访存行为进行更好的学习和预测 [2, 3]。其中，比较典型的预取机制有步长预取、相关性预取等。相比于简单的顺序预取，这些基于表的预取机制可以带来更好的性能提升。然而，这类基于表的预取机制也有其不足之处，比如对于长期没有使用到的表项，容易产生过时的数据，且每个表项能够存储的历史信息也较少。为了对这些缺点进行改进 Kyle 和 James 提出了基于 GHB(global history buffer) 的数据预取机制 [4]。GHB 机制将所有的历史信息存储在一个循环队列中，并使用一个索引表索引该队列中的条目。在一定程度上解决了过时历史数据以及单个表项容量有限的问题。此外，也有人提出了应对不规律的内存访问的预取机制 [5]。主要思想是引入一个额外的间接级别来创建一个新的结构地址空间，其中相关的物理地址被分配给连续的结构地址。一个访问地址流将在空间和时间上顺序地被映射到结构地址空间中，如此便可以将对无规律的内存访问流的预取转化为对结构地址空间中地址的顺序预取。更进一步，为了弥补单一预取机制的缺点，也有人提出了将多种预取策略组合成一个系统的预取机制 [6]。本文中提出的混合预取机制正是借鉴了这一思路。

在缓存替换策略方面，前人也进行了大量的研究。在 1966 年 Laszlo A.Belady 提出了缓存替换的最优算法 OPT [7]，此算法在已知未来所有访问记录的前提下，每次都替换未来最远被访问的现存数据。该算法是理论上的最优算法，但是因为需要已知未来所有访问记录，并不具备可实现性，通常用于衡量其它缓存替换算法的优劣。LRU(Least Recently Used) 缓存替换算法是一个广为人知的在实际中应用广泛的一个缓存替换算法 [8]。该算法认为最近一段时间没有被访问到的数据在将来被访问的可能性最小，每次替换最久未被访问的数据。此外，还有许多比较经典的缓存替换算法如 RAND 随机替换算法、FIFO(First In First Out) 先进先出算法、每次替换访问次数最小的数据的 LFU (Least Frequently Used) 算法以及 RRIP(Re-Reference Interval Prediction) 算法，RRIP 算法使用 M 位来存储每个数据的 RRPV 值，RRPV 值随着每个数据

被访问的频率动态变化，每次替换 RRPV 值等于 $2^M - 1$ 的数据 [9]。随着人工智能的快速发展，最近有人提出了基于人工智能的 PARROT 策略 [10]。PARROT 算法将缓存替换任务建模为强化学习任务，目标是找到能使长期累积奖赏最大化的缓存替换策略。

三、 L2C GSDN 混合预取机制实现

本文提出的混合预取机制 GSDN 结合了三种预取策略。该预取机制首先使用 GHB 步长预取进行预取。如果不满足 GHB 步长预取的预取条件，则将会交由 GHB ip delta 预取机制进行预取，如果也不满足 GHB ip delta 预取的预取条件，则最终简单使用 next line 预取机制预取顺序的下一个缓存行。下面将具体说明这样设计预取机制的原因和预取机制的实现。

（一） GHB 步长预取

考虑到大多数的程序中都会使用数组，并对数组进行有规律的遍历。尤其在科学运算、神经网络等方面，将会频繁的进行矩阵运算。而 GHB 步长预取对于这种模式的数组访问有着较好的性能提升。因此，我们使用 GHB 步长预取作为混合预取机制的第一层。

首先介绍一下传统的步长预取机制，传统的步幅预取使用一个表来存储关于单条 load 指令的与步幅相关的历史信息。因此，它使用 load 指令的 PC 值来索引历史表。每个表项记录对应的 load 指令的最近访存时发生缓存缺失的两个地址之间的步幅以及最近的一个发生缓存缺失的访存地址。当发生缓存缺失时，预取机制会使用当前的 load 指令的 PC 值来索引历史表，并将当前 load 指令的访存地址 addr 与表项中记录的最近一次的缓存缺失地址相减。得到的结果 s 如果与表项中记录的步幅一致，则这个时候预取器会感觉未来很有可能在 $addr + n * s$ 的地址处也发生缓存缺失，此时，它便会向更低一级的 cache 发出预取请求，请求的地址就是 $addr + s$、$addr + 2s$、...、$addr + (1 + d)s$，其中，d 就叫做预取度。

GHB 预取机制主要有两个部件组成，分别是索引表 (Index table) 和全局历史缓冲区 (Global history buffer)。其中，索引表主要用于根据预取键来索引 GHB 缓冲区中的具体条目。全局历史缓冲区，是一个循环队列。其中每项存储一个地址以及一个指针。通过指针按时间顺序将一个预取键对应的地址给链起来。

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

（二） GHB ip delta 预取

程序除了很大可能对基本类型的数组进行有规律的遍历之外，还有可能对类对象或者结构体数组进行有规律的遍历。在这种情况下，GHB 步长预取将不再适用。比如，考虑下面表1这一段地址访问序列。很显然，GHB 步长预取在该段序列上是不适用的，但是我们又明显的可以发现序列中是含有一定规律的。这种情况常常出现在遍历结构体数组，并访问结构体的元素中。因此，我们将 GHB ip delta 预取作为整个预取系统的第二层。当不满足 GHB 步长预取触发条件的时候我们将使用基于 GHB 的 ip delta 预取来处理上述情况。

address	0	1	3	6	72	73	75	78	144
delta		1	2	3	66	1	2	3	66

表 1: address stream

下面介绍一下 GHB ip delta 预取算法。这里我们使该算法与 GHB 步长预取共用一套数据结构，即 GHB 表项中记录的都是具体的地址而不是 delta，我们可以通过简单地将两个地址相减来得到它们之间的 delta。类似的，对于一个 ip，访问地址为 addr。我们首先计算它在 GHB 中历史地址访问序列的最后三个地址之间的两个 delta $\Delta 1$ 和 $\Delta 2$ ，这里并不要求两个 delta 相同。然后顺着地址链表继续向前寻找，在寻找过程中记录一下过程中每两个地址之间的 delta，直到找到与刚才计算出的两个 delta 相同的两个 delta，我们便可以从当前的位置根据预取度来进行预取，预取的地址即为 addr 加上当前位置之后出现的 delta。

以表1为例，设置预取度为 4，如果程序继续访问地址 145，这个时候根据 GHB ip delta 算法，最后三个地址分别为 78、144 和 145，则两个 delta 为 66 和 1，我们继续向前寻找，在地址为 6、72、73 处再次找到了 delta 为 66 和 1。此时触发了预取条件，由于预取度为 4，我们会预取 145+2、145+3、145+66、145+1 处的数据。

（三） next line 预取机制

对于上述使用的 GHB 步长预取和 GHB ip delta 预取，不难发现它们主要适用于预测同一指令在时间上的访问数据地址流。但对于 CPU 的取指来说，要取不同的指令，可能优化不是很明显。因此，当 GHB 步长预取和 GHB ip delta 预取都没有触发预取条件的时候，这里我使用 next line 预取作为混合预取机制的最后一级，主要用于对 CPU 访问缓存的控制流进行预取。

next line 预取机制较为简单，每次 CPU 请求一个缓存行时，会从下一级 Cache 中将下一个缓存行预取到缓存中。思考该预取机制的适用范围，不难发现 next line 预取机制适合对于顺序执行的指令进行预取。对于顺序执行的指令来说，使用 next line 预取机制可以带来不错的性能提升。

（四） 混合预取机制具体实现

下面，具体介绍混合预取机制的实现方式。

1. 数据结构实现

首先说明程序中的宏定义，如下所示，PREFETCH_DEGREE 表示 GHB 步长预取的预取度；PREFETCH_DEGREE2 表示 GHB ip delta 预取的预取度；LOOK_HEAD 表示 GHB 步长预取的 look_ahead；GHB_SIZE 和 IT_SIZE 则分别为 GHB 和索引表的大小。

```
#define PREFETCH_DEGREE 10
#define PREFETCH_DEGREE2 10
#define LOOK_HEAD 1
#define GHB_SIZE 256
#define IT_SIZE 256
```

数据结构方面，我们需要实现索引表和全局历史缓冲区。这里我使用数组来对它们进行实现。

索引表数据结构如下，定义结构体 IT_ENTRY 表示一个索引表表项，其中存储了该项索引的 GHB 表项的下标。定义 IT_ENTRY 数组来模拟索引表，索引表使用直相联，一个 PC 值对应的表项为 it[PC % IT_SIZE]。

```
typedef struct
{
    unsigned int prev;
} IT_ENTRY;

IT_ENTRY it[IT_SIZE] = {0};
```

全局历史缓冲区定义如下，定义结构体 GHB_ENTRY 表示一个 GHB 表项，其中 cl_addr 为实际访问的缓存行地址，pc 为访问该缓存行的指令的 pc 值，prev 为 GHB 指针，指向 GHB 表中上一个该 pc 对应的表项。定义 GHB_ENTRY 数组来模拟全局历史缓冲区，下方代码的 curr_idx 为新的 GHB 表项要存入的 GHB 表的位置下标。

```
typedef struct
{
    unsigned long long pc;
    unsigned long long cl_addr;
    unsigned int prev;
} GHB_ENTRY;

GHB_ENTRY ghb[GHB_SIZE] = {0};
unsigned int curr_idx = 0;
```

2. 预取机制实现

接下来，具体讲述混合预取机制的实现，即 l2c_prefetcher_operate 函数的实现。当 L2C 接收到一个访问请求时，混合预取机制会进行以下几步操作：

1. 将请求的地址插入到 GHB 表中，并更新索引表。之后交由 GHB 步长预取机制来处理，转向下一步。
2. 得到 GHB 中该 pc 对应的地址链，并计算地址链上最后三个地址之间的步长 stride1、stride2。若计算得到的两个步长相等，则发出预取请求，根据规则预取若干缓存行到 L2C 中。若两个步长不相等，则交由 GHB ip delta 预取机制来处理，转到下一步。
3. 根据 GHB ip delta 机制，在地址链表中继续向前寻找，在寻找过程中记录每一步计算出的 delta。若最终成功找到了和 stride1、stride2 相等的 $\Delta 1$、$\Delta 2$，则根据记录的 delta 预取若干缓存行。如果始终没有找到对应的 delta 值，则将交由 next line 预取机制来处理，转到下一步。
4. next line 预取机制较为简单，只需要预取当前请求的缓存行的下一个缓存行即可。

下面展示一下具体实现代码。

更新 GHB 表和索引表的代码如下所示。首先，计算出请求的缓存行地址。然后根据 ip 来索引索引表，得到当前 ip 对应的 GHB 表项。若 GHB 表项中的 pc 值等于 ip，则表示该索引表项是有效的，将新插入的 GHB 表项的指针指向索引表项指向的 GHB 表项即可; 否则表示索引

表无效，即索引表项索引的并不是对应于当前 ip 的 GHB 表项，此时将新插入的 GHB 表项的指针指向新插入的 GHB 表项自身，从而表示链表的结尾。之后将当前缓存行地址和当前 ip 赋值到新插入的 GHB 表项的对应属性中即可。然后再更新索引表，将当前 ip 对应的索引表项指向新插入的 GHB 表项。

```
    uint64_t cl_addr = addr >> LOG2_BLOCK_SIZE;
    if (ghb[it[ip % IT_SIZE].prev].pc == ip)
    {
        ghb[curr_idx].prev = it[ip % IT_SIZE].prev;
    }
    else
    {
        ghb[curr_idx].prev = curr_idx;
    }
    ghb[curr_idx].cl_addr = cl_addr;
    ghb[curr_idx].pc = ip;
    it[ip % IT_SIZE].prev = curr_idx;
```

接下来介绍上述预取机制的第二三步，即 GHB 步长预取和 GHB ip delta 预取的实现部分。这里我以一种状态机的形式对 GHB 步长预取和 GHB ip delta 预取以及它们的交接进行实现。如下所示，定义枚举类型 STATE，表示当前状态机的状态，状态分别为：

1. STRIDE1：初始状态，GHB 步长预取计算第一个步长，执行完后转向 STRIDE2 状态。
2. STRIDE2：GHB 步长预取计算第二个步长，执行完后转向 COND 状态。
3. COND：判断两个步长是否相等，若相等则预取若干缓存行，并转到 OVER 状态。否则，转到 FINDS1 状态，交由 GHB ip delta 预取进行处理。
4. FINDS1：GHB ip delta 预取判断当前 delta 和 stride1 是否相等，若相等，则转向 FINDS2 状态。
5. FINDS2：GHB ip delta 预取判断当前 delta 和 stride2 是否相等，若相等，则根据规则预取若干缓存行，并转到 OVER 状态，否则转向 FINDS1。
6. OVER：GHB 步长预取和 GHB ip delta 预取结束，状态机结束。

```
typedef enum
{
    STRIDE1,
    STRIDE2,
    COND,
    FINDS1,
    FINDS2,
    OVER
} STATE;
```

状态机代码如下所示，其中 prefetch_addr 表示要预取的地址；stride1 和 stride2 存储 GHB 步长预取机制计算出的两个步长；delta 存储 GHB ip delta 预取当前遍历到的 delta 值；elem_idx 表示当前访问的地址链表元素在 GHB 表中的下标；state 用于记录状态机的状态；dels 用于记

录访问地址链过程中所遍历到的 delta 值，用于 GHB ip delta 的预取；need_next 则表示最后是否需要启用 next line 预取。状态机的代码较易理解，我们只需要根据上述描述的状态，在各个状态进行相应的计算和预取即可，这里不做过多说明。

```
    uint64_t prefetch_addr;
    long long int stride1 = 0, stride2 = 0;
    long long int delta = 0;
    unsigned int elem_idx = curr_idx;
    stride1 = ghb[curr_idx].cl_addr;
    STATE state = STRIDE1;
    std::vector<long long int> dels;
    int need_next = 1;

    while (ghb[elem_idx].prev != elem_idx && ghb[elem_idx].pc == ip && state
        != OVER)
    {
        unsigned int prev_idx = ghb[elem_idx].prev;
        switch (state)
        {
        case STRIDE1:
            stride2 = ghb[prev_idx].cl_addr;
            stride1 = stride1 - stride2;
            dels.insert(dels.begin(), stride1);
            state = STRIDE2;
            elem_idx = prev_idx;
            break;
        case STRIDE2:
            delta = ghb[prev_idx].cl_addr;
            stride2 = stride2 - delta;
            dels.insert(dels.begin(), stride2);
            state = COND;
            elem_idx = prev_idx;
            break;
        case COND:
            if (stride1 == stride2)
            {
                need_next = 0;
                for (int i = LOOK_HEAD; i <= LOOK_HEAD + PREFETCH_DEGREE; i
                    ++)
                {
                    prefetch_addr = (cl_addr + i * stride1) <<
                        LOG2_BLOCK_SIZE;
                    prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
                }
                state = OVER;
            }
            else
            {
                state = FINDS1;
```

```
            }
            break;
        case FINDS1:
            delta = delta - ghb[prev_idx].cl_addr;
            dels.insert(dels.begin(), delta);
            if (delta == stride1)
            {
                state = FINDS2;
            }
            delta = ghb[prev_idx].cl_addr;
            elem_idx = prev_idx;
            break;
        case FINDS2:
            delta = delta - ghb[prev_idx].cl_addr;
            dels.insert(dels.begin(), delta);
            if (delta == stride2)
            {
                need_next = 0;
                for (int i = 2; i - 2 < PREFETCH_DEGREE2 && i < dels.size();
                    i++)
                {
                    prefetch_addr = (cl_addr + dels[i]) << LOG2_BLOCK_SIZE;
                    prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
                }
                state = OVER;
            }
            else
            {
                state = FINDS1;
            }
            delta = ghb[prev_idx].cl_addr;
            elem_idx = prev_idx;
            break;
        default:
            state = OVER;
            break;
        }
    }
```

完成了 GHB 预取和 GHB ip delta 预取的部分后，我们最终需要判断是否需要使用 next line 预取来对控制流进行预取。根据上述提到的 need_next 变量，如果 need_next 为 0，则表示已经触发了 GHB 步长预取或者 GHB ip delta 预取的预取条件，此时不再需要使用 next line 预取；如果 need_next 为 1，则我们这里需要使用 next line 预取预取下一个缓存行。具体代码如下所示。

```
    if (need_next)
    {
        prefetch_addr = (cl_addr + 1) << LOG2_BLOCK_SIZE;
        prefetch_line(ip, addr, prefetch_addr, FILL_L2, 0);
```

```
    }
```

四、 LLC 缓存替换策略实现

缓存替换策略方面，这里我们主要实现了 LFU 和 window-LFU 缓存替换策略。同时，我们对实验原始提供的 LRU、SHiP、SRRIP 缓存替换算法代码也进行了阅读和学习，下面介绍一下以上缓存策略的原理和实现方式。

（一） LFU 缓存替换策略

1. 算法原理

LFU 缓存算法维护一个关于缓存中缓存行访问次数的计数器。它将删除访问频率最低的项，以便添加新项。每次进行缓存读写时，缓存行中的访问次数状态都要有所变化。

2. 算法实现

考虑在 LLC 上实现 LFU 缓存替换算法，该算法的实现较为简单，数据结构方面，我们只需要维护一个记录每个缓存行历史访问次数的二维数组 freq[LLC_SET][LLC_WAY]。在初始时，将数组清零。每当一个缓存行被访问到时，将对应的数组元素加一。当需要从某一组中驱逐出一个缓存行时，我们只需要遍历整个组中所有缓存行对应的 freq 数组元素，并选出数组元素最小的缓存行，将其替换掉即可。

具体选择替换的缓存行的代码如下所示。

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

cache 命中以及 cache 填充时的代码如下所示。如果访问命中，我们把对应的数组元素加一。如果未命中且此时类型为预取，则考虑到预取时其实并没有访问该缓存行，这里我们将访问次数设置为零。否则访问次数设置为 1。

```
    if (hit)
    {
        freq[set][way] += 1;
    }
```

```
    else if (type == PREFETCH)
    {
        freq[set][way] = 0;
    }
    else
    {
        freq[set][way] = 1;
    }
```

（二） window-LFU 缓存替换策略

由于 LFU 记录的是缓存行在历史上的访问频率，如果一条缓存行曾经被访问了很多次，但是如果服务的逻辑发生了改变，这条缓存行未来将很少甚至不再会被访问。然而，这条缓存行由于其历史访问次数很高，依然不会被淘汰。这就导致了缓存污染的问题。window-LFU 算法在一定程度上解决了这个问题。

1. 算法原理

与 LFU 记录每个缓存行全部的历史访问次数不同，window-LFU 算法维护了一个大小有限的窗口，窗口中记录了在过去一段时间内的缓存访问请求。当窗口占满时，每到来一个缓存访问请求，算法均会把窗口中最老的一条访问请求记录给清除掉，并把新的访问请求记录添加到窗口中，保证算法记录的请求次数不超过窗口的大小。

window-LFU 的缓存淘汰方式和 LFU 是一致的。window-LFU 首先会计算窗口中存储的记录中各缓存行的访问次数，然后根据访问次数排序，淘汰访问次数最少的缓存行。

2. 算法实现

这里，我使用一种比较简易的方式来实现 window-LFU 算法。具体为，为每一组缓存行创建一个队列，用来模拟该组缓存行的窗口，当该组缓存行有新的访问请求时，将新的记录插入到队尾，同时如果队列长度超过窗口大小，就把队首的记录给清除掉。当要从某一组缓存行中驱逐出一条缓存行时，遍历该组缓存行的窗口，从而计算出过去一段时间内该组缓存行中每个缓存行被访问的次数，从中选出访问次数最少的缓存行将其驱逐即可。

具体代码如下所示，首先使用一个 vector 来模拟各组缓存行的窗口。

```
std::vector<vector<int>> wnds;
```

具体选择替换的缓存行的代码如下所示。首先，遍历对应 set 的窗口，计算出每个缓存行在窗口中的历史访问次数，存储到 freq 数组中。之后的逻辑和 lfu 算法相同，即找出访问次数最少的缓存行并将其返回。

```
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)
{
    vector<int> list = wnds[set];
    int freq[LLC_WAY] = {0};
    for (int i = 0; i < list.size(); i++)
    {
```

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

cache 命中以及 cache 填充时的代码如下所示。如果窗口满了，我们就将窗口队列中队首的访问记录删除，然后将新加入的访问记录插入到队尾即可。这里和 lfu 类似的，由于预取时并不算是真正的访问，我们这里并不把预取类型的 cache fill 记录插入到窗口队列中。

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

（三） LRU 缓存替换策略

LRU(Least recently used) 算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

由于实验已经提供了 LRU 算法的实现，这里对于其实现不做过多说明。

（四） SHiP 缓存替换策略

SHiP(Signature-based Hit Predictor) 算法利用署名信息来预测 cache 中插入的数据将会在未来的哪里被再次访问。算法使用一个饱和计数器组成的 SHCT 表来学习签名的重引用行为。当缓存行命中时，SHiP 算法会增加 SHCT 表中该缓存行对应的签名的值。当一个缓存行要被替换出去并且在插入之后没有被重引用过，SHiP 会减少 SHCT 中对应的值。SHCT 表中的值代表着签名的重引用行为。如果值为 0，则说明这个缓存行很有可能不会被使用。换句话说，与签名相关联的引用的重引用间隔很大。另一种情况，如果 SHCT 中计数器的值是正的，说明相应的签名很有可能被命中。由于 SHCT 值记录一个给定的签名是否被重引用而不是时间，所以 SHCT 无法得到准确的重引用间隔。SHiP 的根本目的是为 LLC 的替换策略提供一个参考，算法可以提供对于每一个插入的缓存行给出一个重引用间隔。在算法执行过程中，如果发生缓存缺失，通过要插入的缓存行的签名在 SHCP 表中找到相应的值，如果这个值为零则表示该要插入的缓存行的重引用间隔很大，否则就认为重引用间隔较小，将会被访问。利用这些信息，替换策略可以选择是否要替换该行。

由于实验已经提供了 SHiP 算法的实现，这里对于其实现不做过多说明。

（五） SRRIP 缓存替换策略

SRRIP(Static RRIP) 替换策略是对 NRU 的扩展，其将 NRU bit 扩展成 M 位，称为该缓存行的 RRPV 值。每当一个 cache hit，该缓存行的 RRPV 值被设置为 0，表示在最近的将来，该 cache block 很有可能再被访问到；每当一个 cache miss，替换算法会从左至右扫描 RRPV 值为 $2^M - 1$ 的块，如果找到则替换出该缓存行，并将新插入的缓存行的 RRPV 值置为 $2^M - 2$，如果没有找到，那么将所有缓存行的 RRPV 值加 1，重新从左至右扫描。新插入的缓存行设置为 $2^M - 2$，主要是为了防止那些很久才能被再次使用到的缓存行长期占用 cache 空间。

由于实验已经提供了 SRRIP 算法的实现，这里对于其实现不做过多说明。

五、 性能评估

（一） 评估方式

这里我们将多种预取机制，如本文提出的 GSDN 混合预取机制、GHB 步长预取机制、next line 预取机制、不使用预取机制和多种缓存替换策略如 LFU、window-LFU、LRU、SHiP、SRRIP 缓存替换策略相结合，在实验提供的数据集上测试它们的平均 IPC。这里我们不对传统的不使用 GHB 的 ip stride 预取机制进行性能测试，因为我们认为 GHB 步长预取机制在绝大多数情况下性能都是要比传统的 ip stride 预取机制要好的。

（二） 评估结果

得到的平均 IPC 如表2所示。

预取机制\替换策略	LFU	window-LFU	SHiP	SRRIP	LRU
NO	0.597475	0.542465	0.596555	0.546580	0.531730
next line	0.795285	0.759100	0.775485	0.732320	0.702960
GHB ip stride	1.044320	1.043030	0.988835	0.967850	0.938240
GSDN	1.060480	1.062980	1.028880	1.016725	0.986200

表 2: 不同预取机制与不同缓存替换策略结合在测试数据集上的平均 IPC 得分

下面我们根据得到的数据进行相关算法性能的分析。我们首先分析不同预取机制对整体性能的影响。绘制柱状图如图1所示。可以看到，在本实验中无论使用何种 LLC 缓存替换策略，使用各个预取机制得到的性能相对大小是类似的，不使用预取机制的性能最差，其次是使用 next line 预取机制，然后是单纯的 GHB 步长预取机制，我们提出的混合预取机制 GSDN 在各个缓存替换算法下均取得了最好的性能。这也验证了我们对于该 GSDN 混合机制的设计是合理的。

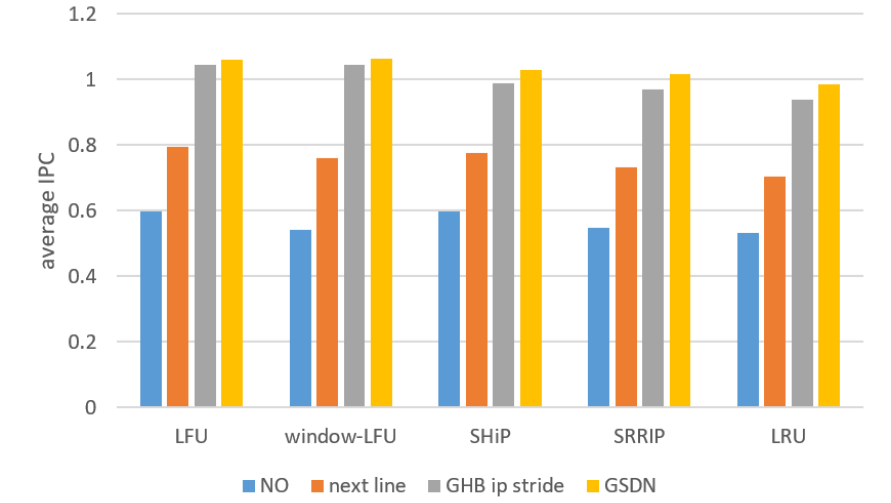

图 1: 预取机制对比

下面我们计算各个预取机制的预取准确率。这里，我们定义的预取准确率为有用的预取占全部预取的比例。显然，预取准确率越大，预取机制对于计算机资源的浪费越少。即使预取机制有较好的加速效果，但是如果有过小的预取准确率，将会造成大量的资源的浪费。表3为使用 lfu 缓存替换策略下三种不同的预取机制在两个数据集上的预取准确率。可以看到，next line 的预取准确率最高，其次为 GHB 步长预取机制，而我们提出的 GSDN 准确率最低，这可能是由于我们 GSDN 预取机制过于激进造成的。根据上述对于 GSDN 机制的描述，不难发现我们对于每一次缓存访问均会进行数据的预取，即使只是预取下一个缓存行。这是一点在未来可以进行改进的方面。

预取机制\数据集标号	462	482	average accuracy
next line	99.998	88.446	94.222
GHB ip stride	99.997	82.719	91.358
GSDN	99.997	79.580	89.789

表 3: LLC 缓存替换策略为 LFU，三种预取机制准确率对比（单位：%）

最后，我们对于各个缓存替换策略的性能进行比较。从图2中不难看出，在实验中的各个预取机制下，LFU 替换策略表现出的性能要好于 SHiP 替换策略，SHiP 替换策略好于 SRRIP 替换策略，SRRIP 替换策略好于 LRU 替换策略。这应该是由于测试数据集中的程序不频繁访问相同的内容造成的，如果程序频繁访问相同的内容，显然应该是 LRU 替换策略的性能最好，这里应该是因为测试集中的程序更多的进行数组等的遍历操作，这类操作的时间局部性较差，从而导致 LRU 没有取得很好的性能。

此外，从图2中我们还可以注意到，window-LFU 的算法性能并不是十分稳定。虽然 window-LFU 算法是对 LFU 算法的改进，但是在没有预取机制以及 next line 预取机制下，window-LFU 算法的性能并不是十分优秀。猜测造成该现象的原因和上述分析是一致的，即测试集中的程序不会频繁的访问相同的内容，这就使得 LFU 算法中不容易出现脏数据。而 window-LFU 算法由于限制了窗口中记录的历史访问记录条目的个数，会导致驱逐缓存行时依据的信息产生误差，从而反而导致 window-LFU 算法的性能不如 LFU 算法。

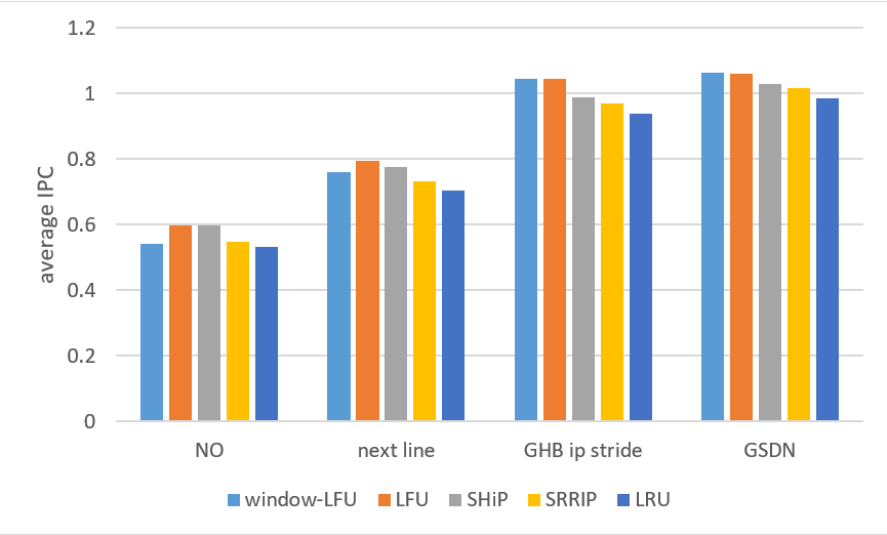

图 2: 缓存替换策略对比

最后，LLC 使用 window-LFU 替换策略，计算出本文提出的 GSDN 混合预取机制相对于其他预取机制在测试数据集合上的平均 IPC 加速比如表4所示。

GHB ip stride	next line	NO
1.01912697	1.4003162	1.9595366

表 4: GSDN 相对于其余预取机制 IPC 加速比

参考文献

[1] A. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[2] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[3] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[4] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004, pp. 96–96.

[5] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 247–259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[6] M. Chaudhuri and N. Deshmukh, "Sangam: A multi-component core cache prefetcher," *3rd Data Prefetching Championship*, 2019.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[8] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *SIGMOD Rec.*, vol. 22, no. 2, p. 297–306, jun 1993. [Online]. Available: https://doi.org/10.1145/170036.170081

[9] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815971

[10] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 6237–6247. [Online]. Available: https://proceedings.mlr.press/v119/liu20f.html