南開大學
Nankai University

cache 替换策略和数据预取

计算机体系结构期末大作业报告

姓名：陈睿颖

学号：2013544

专业：计算机科学与技术

2023 年 1 月 3 日

目录

1 实验内容

本实验使用了 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。

现已有一些程序的 trace，以及 ChampSim 的源代码和对应接口，本实验的主要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 预取

预取是一种预测技术，用于预测未来的内存请求，并在处理器实际需要之前将其提取到缓存中。这种预测可以使程序运行更快。在具有三级缓存层次结构的处理器核中，可以在任何缓存层次上使用预取程序。现在有许多工作提出了预取算法，以准确预测程序未来的内存访问。

Cache 失效通常被分为三类：

- 强制性失效
- 冲突失效
- 容量失效

有很多技术关注的是减少冲突性失效，但是它们对于冲突失效和容量失效则不能解决。预取技术预测将会引起 Cache 失效的访存，利用存储器的空闲带宽，提前将其取回，从而隐藏由于访存延迟引起的处理器停顿。能够很好的解决强制性失效和容量失效。一般来说，CPU 并不直接与存储器交换数据，而是通过 Cache 间接进行。由平均访存时间公式和程序运行时间公式可以看出，Cache 失效对于系统的性能有着很大的影响。因此，为了改进系统的性能，首先必须要找出 Cache 失效的特点。通过对 NPB 访存行为的研究，发现在 NPB 测试集中线性访存模式是造成 Cache 失效的最主要的原因。统计结果表明，由线性访存模式直接引起的 Cache 失效占程序总的 Cache 失效次数的 68.6%；而若考虑由此引起的 Cache 污染对失效率的影响该比例上升至 78.2%。同时，线性访存模式具有模式简单，便于优化的优点。因此，在研究 Cache 行为、提高 Cache 利用率以及设计新的软件可管理的 Cache 结构时应该对这类访存模式给予考虑。对于线性访存模式所引起的大量失效，预取是个很好的选择。使用预取技术，能使数据和指令可在 CPU 使用之前到达与 CPU 更近的存储层次，因此在真正需要它们时能及时的拿到或者可以减少延时和阻塞的时间。

2.1.1 next_line 预取器

next_line 预取器实现原理为, 若预取 buffer 中有 CPU 请求的 cache_line, 则把该 cache_Line 送到 cache 中，并且把下个 cache_Line 的内容的读取请求发往下一级 cache.

2.1.2 table-based IP-stride

该预取器会检测来自相同 ip 的步长模式, 并且预取额外的 cache_line, 根据 L2 MSHR 的占用情况，预取被发布到 L2 或 LLC.

2.1.3 基于 GHB 的步长预取算法

该算法的目的是通过函数在 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。它由两个主要结构组成:

- 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。
- 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。

该算法的过程如下:

1. 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。
2. 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。

对于本次实验的设计，可以将 l 和 n 静态设置。并调整 IT 和 GHB 的大小，使其为 256 个条目。

2.2 cache

缓存（cache），原始意义是指访问速度比一般随机存取存储器（RAM）快的一种高速存储器，通常它不像系统主存那样使用 DRAM 技术，而使用昂贵但较快速的 SRAM 技术。缓存的设置是所有现代计算机系统发挥高性能的重要因素之一。

在计算机领域，缓存可以加速处理器对于内存的访问。内存请求在缓存中找到数据时比没找到时处理地要快。其中影响缓存性能的一个关键因素是缓存的替换策略，替换策略指定当一个新的缓存行插入到缓存中时，集合中的哪个缓存行被替换。以 LRU（Least Recently Used ）为例，LRU 使用了一个替换策略来替换最近使用最少的行。

2.2.1 cache 工作原理

缓存的工作原理是当 CPU 要读取一个数据时，首先从 CPU 缓存中查找，找到就立即读取并送给 CPU 处理；没有找到，就从速率相对较慢的内存中读取并送给 CPU 处理，同时把这个数据所在的数据块调入缓存中，可以使得以后对整块数据的读取都从缓存中进行，不必再调用内存。正是这样的读取机制使 CPU 读取缓存的命中率非常高（大多数 CPU 可达 90% 左右），也就是说 CPU 下一次要读取的数据 90% 都在 CPU 缓存中，只有大约 10% 需要从内存读取。这大大节省了 CPU 直接读取内存的时间，也使 CPU 读取数据时基本无需等待。总的来说，CPU 读取数据的顺序是先缓存后内存。

2.2.2 SHiP

该算法的过程为: 首先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 会较少 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 cache 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 [1]。

2.2.3 srrip

首先，算法引入了要给概念叫 RRPV (Re-Reference Prediction Value)；RRPV 是一个多 bit 信号；RRPV 的更新策略和 LRU 其实类似，但是由于多 bit 增加带来的更大的状态空间，算法在 insertion 和 promotion 可以更新不同的值到 RRPV，由此，这个算法解决了 recency-friendly 和 scan 混合的场景；在 evict 时选择，RRPV 最大的 candidate；

2.2.4 常用的 cache 替换策略

1. **LRU**

   LRU（Least Recently Used，近期最少使用）算法是把 CPU 近期最少使用的块替换出去。这种替换方法需要随时记录 Cache 中各块的使用情况，以便确定哪个块是近期最少使用的块。每块也设置一个计数器，Cache 每命中一次，命中块计数器清零，其他各块计数器增 1。当需要替换时，将计数值最大的块换出。

   LRU 算法相对合理，但实现起来比较复杂，系统开销较大。这种算法保护了刚调入 Cache 的新数据块，具有较高的命中率。LRU 算法不能肯定调出去的块近期不会再被使用，所以这种替换算法不能算作最合理、最优秀的算法。但是研究表明，采用这种算法可使 Cache 的命中率达到 90% 左右。

2. **LFU**

   LFU（Least Frequently Used，最不经常使用）算法将一段时间内被访问次数最少的那个块替换出去。每块设置一个计数器，从 0 开始计数，每访问一次，被访块的计数器就增 1。当需要替换时，将计数值最小的块换出，同时将所有块的计数器都清零。

   这种算法将计数周期限定在对这些特定块两次替换之间的间隔时间内，不能严格反映近期访问情况，新调入的块很容易被替换出去。本次实验即实现了 LFU 的替换策略。

3. **随机替换**

   最简单的替换算法是随机替换。随机替换算法完全不管 Cache 的情况，简单地根据一个随机数选择一块替换出去。随机替换算法在硬件上容易实现，且速度也比前两种算法快。缺点则是降低了命中率和 Cache 工作效率。

2.3 实验环境

本次实验在 Linux 系统下进行。具体实验环境信息如下：

- 操作系统: Linux

- ubuntu 版本: WSL:Ubuntu-22.04
- 编译器版本: gcc version 11.3.0

3 实验步骤

3.1 实现一个预取器: 基于 GHB 的步长预取算法

在原理部分介绍了基于 GHB 的步长预取算法, 在这里进行具体的代码实现.
首先, 要根据原理部分定义两个结构, 索引表及 GHB:

```
struct GHB_entry
{
    uint64_t cache_line_addr;          // cache_line 地址
    unsigned int prev_ptr = 256;     // 链表指针
} ;

struct IT_entry
{
    unsigned int GHB_ENTRY_ptr = 256; //指向 GHB 条目的指针
};
```

为了方便理解, 还定义了一些常量:

```
/******* 定义常量 *********/
const int GHB_SIZE = 256;
const int IT_SIZE = 256;
const int look_head = 1;
const int d = 20;
```

接下来就可以定义以这些结构体构成的数组来表示 IT Table 以及 GHB Table:

```
struct GHB_entry GHB[GHB_SIZE];
struct IT_entry IT[IT_SIZE];
unsigned int k = 0; //记录当前的索引
```

初始化函数, 输出对应的信息:

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

然后就要进行预取器的具体实现. 根据上面的算法思路, 在代码中实现可以分为以下几步:

1. 进行 GHB 表项的设置: 若该表项没有使用, 则说其没有对应的 GHB 表项，该 GHB 表项是链表中的第一个项，前置项无，记为 256;

2. 更新 IT 表项;
3. 查找最近 3 次访问的 cache 的块号检查是否有 3 次 miss, 如果有 3 次 Miss 且计算的步长相同则预取;

代码实现如下:

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

    if (IT[ip % IT_SIZE].GHB_ENTRY_ptr != 256)
    {
        GHB[k].prev_ptr = IT[ip % IT_SIZE].GHB_ENTRY_ptr;
    }
    else
    {
        GHB[k].prev_ptr = 256;

    }
    uint64_t cache_line_addr = addr >> LOG2_BLOCK_SIZE;
    GHB[k].cache_line_addr = cache_line_addr;

    IT[ip % IT_SIZE].GHB_ENTRY_ptr = k;
    unsigned int block1=k;
    unsigned int block2=GHB[block1].prev_ptr;
    unsigned int block3=0;
    bool triple_miss=false;
    if(block2 != 256){
        block3=GHB[block2].prev_ptr;
        if(block3 != 256){
            triple_miss=true;
        }
    }
    if(triple_miss){
        uint64_t stride1 = GHB[block1].cache_line_addr - GHB[block2].cache_line_addr;
        uint64_t stride2 = GHB[block2].cache_line_addr - GHB[block3].cache_line_addr;
        if(stride1==stride2){
            for (int i = look_head; i <= look_head + d; i++)
                {
                    uint64_t pf_addr = (cache_line_addr + i * stride1) << LOG2_BLOCK_SIZE;
                    prefetch_line(ip, addr, pf_addr, FILL_L2, 0);
                }
        }
    }
    k++;
    k %= 256;

    return metadata_in;
}
```

最后填补一下最后统计数据的函数即可:

```
void CACHE::l2c_prefetcher_final_stats()
{
    cout << "CPU " << cpu << " L2C GHB Stride prefetcher final stats" << endl;
}
```

3.2 实现一个 cache 替换策略:LFU

LFU 策略在原理部分中也已介绍, 其核心思想就是要寻找访问次数最少的块. 在代码实现中, 可以声明一个二维数组记录访问次数:

```
uint32_t counts[LLC_SET][LLC_WAY]; //记录访问次数的二维数组
```

初始化时, 将该二维数组全部置为 0:

```
// initialize replacement state
void CACHE::llc_initialize_replacement()
{
    cout << "Initialize LFU state" << endl;
    for (int i = 0; i < LLC_SET; i++) {
        for (int j = 0; j < LLC_WAY; j++) {
            counts[i][j] = 0;
        }
    }
    miss_count = 0;
    hit_count = 0;
}

```

接下来进行遍历, 寻找访问次数最少的 way 以及该 way 的访问次数:

```
// 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_way = -1; //访问次数最小的 way
    uint64_t min_count = -1; //访问最小的次数

    //寻找访问次数最少的 way, 进行遍历两两比较, 找到并返回
    for (uint32_t i = 0; i < LLC_WAY; i++) {
        if (counts[set][i] < min_count) {
            min_count = counts[set][i];
            min_way = i;
        }
    }

    return min_way;
```

```

}
```

在每次 cache 命中或者填充时, 调用更新状态的函数, 其工作就是若发生命中, 我们用于记录访问次数的二维数组对应元素数值 +1, 若没有命中, 则重新置 1.

```
// 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";

    if ((type == WRITEBACK) && ip)
        assert(0);

    // uncomment this line to see the LLC accesses
    // cout << "CPU: " << cpu << "  LLC " << setw(9) << TYPE_NAME << " set: " << setw(5) <<
    // cout << hex << " paddr: " << setw(12) << paddr << " ip: " << setw(8) << ip << " victi

    if (type != WRITEBACK || (type == WRITEBACK && !hit)) {
        lru_update(set, way);
    }
```

```
38
39      if (hit) {
40          counts[set][way] += 1;
41
42      }
43      else {
44          counts[set][way] = 1;
45
46      }
47      hit_count++;
48      if (hit_count == 50) {
49          miss_count = 0;
50          hit_count = 0;
51      }
52  }
```

4 性能测试与分析

4.1 策略的组合

根据已完成的预取器和 cache 替换策略的实现, 可以组合出 12 种不同的策略组合, 分别为:
预取器:

- 无
- next_line 预取器
- table-based IP-stride 预取器
- GHB 预取器

cache 替换策略:

- LRU
- LFU
- SHiP
- srrip

4.2 结果分析

经过组合后在 436 数据集上测定每一次运行的 IPC 结果, 如下图所示:

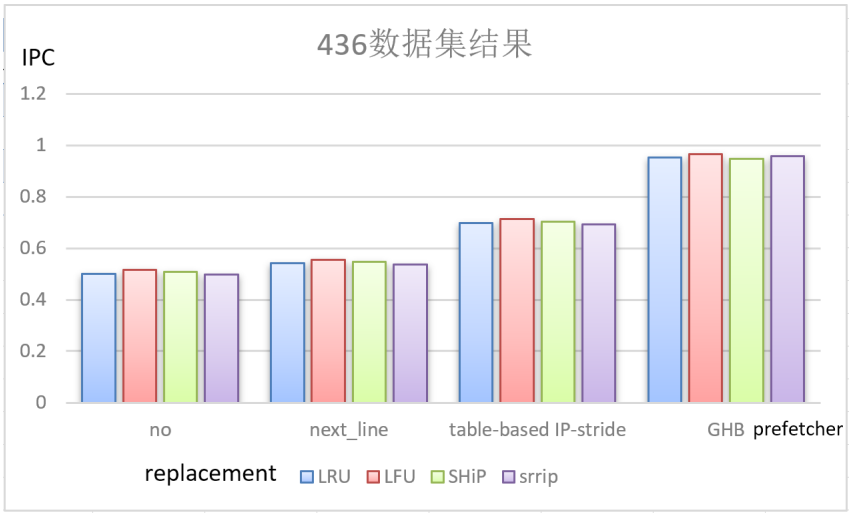

图 4.1: 436 Result

再运行 score 脚本, 每一次运行的得分结果如下图所示:

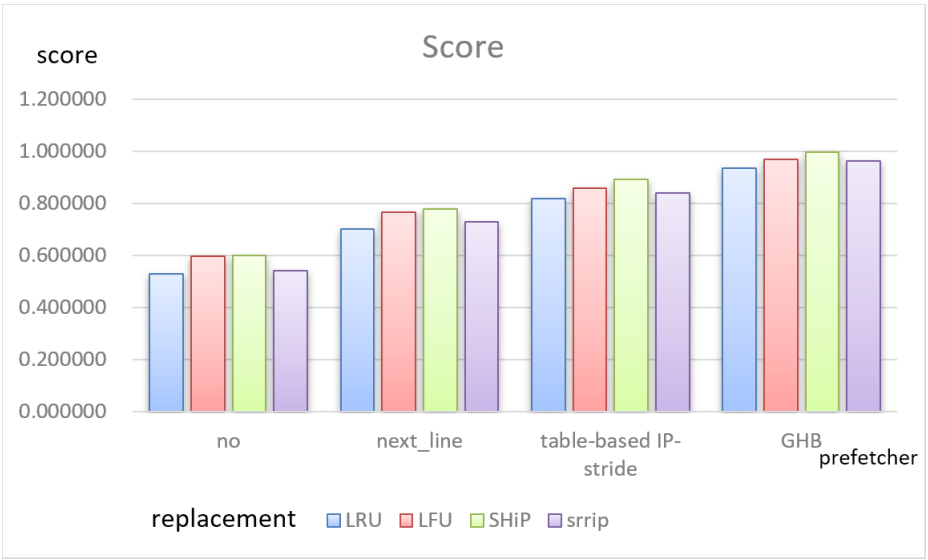

图 4.2: score

可以看出, 在 436 数据集上, 本实验实现的 GHB 预取器无论结合哪种 cache 替换策略, 其 IPC 指数均高于其他预取器. 其中, 本实验实现的 LFU 算法的 cache 替换策略又略高于其他 cache 替换策略; 在得分方面, 仍然是 GHB 预取器得分最高, 而各种 cache 替换策略中, SHiP replacement 得分略高出于其他, 其原因是 SHiP 算法所占用的存储空间更大, 其记录的历史行为有助于其进行进行缓存替换, 带来了更高的 IPC 性能. 这可以类比于将空间换时间的说法. LFU 所使用的存储空间为 $set \times way \times 32\text{bits}$, 这会小于 SHiP 算法所使用的存储空间. 但数据上看两者差异并不大, 可以认为本次实验实现的 GHB-LFU 的预取器-cache 替换策略达到了良好的性能.

参考文献

[1] Calvin Lin Akanksha Jain. *Cache Replacement Policies*. Springer Cham, 2019.