南開大學
Nankai University

计算机学院

计算机体系结构实验期末报告

Cache 预取和替换实验

姓名：彭钰钊

学号：2110756

专业：计算机科学与技术

2024 年 1 月 18 日

目录

1 研究动机

随着半导体工艺技术水平的不断提高和计算机组成的快速发展，处理器的频率得到了大幅度的提高，这给存储器带来了更高的要求，尽管存储器的容量在快速增大，但是其存取的速率提升却已然无法满足处理器的计算速度。因此，存储器和处理器之间的处理速度不匹配成为了制约计算机性能的一个关键节点，现代计算机体系结构中采用多级 Cache（高速缓冲存储器）来解决该问题。然而，虽然 Cache 相较于存储器的存取速率有很大提升，但是由于成本等原因，其容量相当有限。为了更好地利用 Cache 提高计算机的性能，我们需要设计并实现一种能够更加高效利用 Cache 的算法，本实验主要探索的是数据预取和 Cahce 替换策略这两种提高 Cache 命中率的手段。

2 相关工作

学术界对于 Cache 的研究一直是热点，影响 Cache 性能的因素有很多，其中预取（prefetching）和替换（replacement）算法是最重要的两个因素。尽管研究 Cache 预取和替换算法的工作有很多，但是已有工作存在两个问题：

（1）仅考虑 Cache 预取或者仅考虑 Cache 替换，没有考虑两者之间的关联性；

（2）通常只考虑某个层级（例如，L2 或者 L3）的预取或者替换策略，没有考虑多个层级预取和替换策略之间的关联关系。

本实验针对上述问题，研究多层级预取和替换联合优化策略，目标是最大化应用程序的性能。如图2.1是 Cache 替换算法的分类

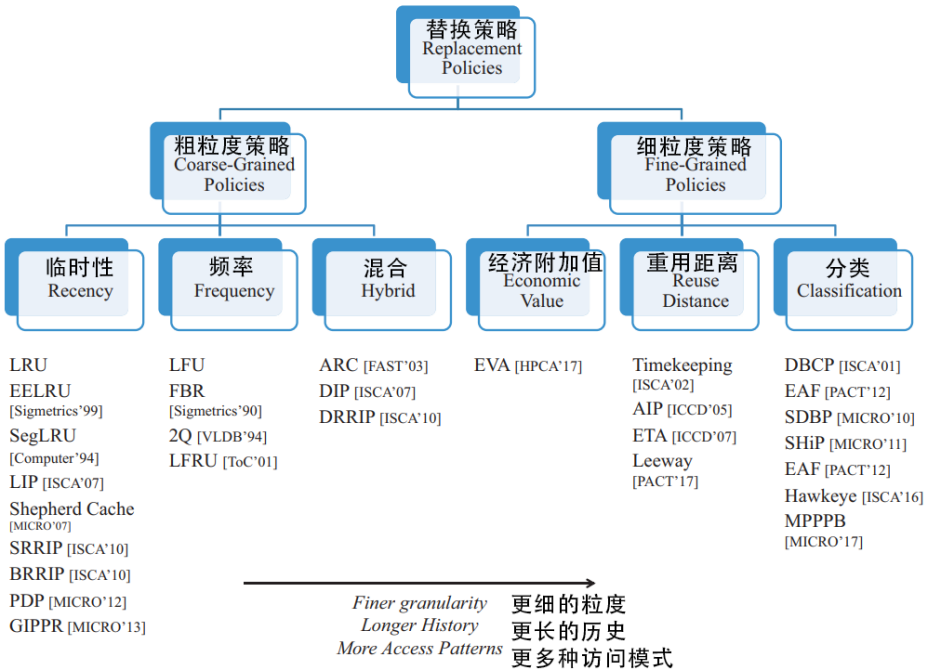

Figure 2.2: A taxonomy of cache replacement policies.
缓存替换策略的一种分类法。

图 2.1: cache 替换策略

3 方法描述

预取策略：

IP Stride 预取策略基于指令地址的步幅，预测程序访问指令的模式。通过监测指令地址的增量，尝试预测未来指令的位置，并在缓存中预取相关的指令块。这有助于提高指令访问的效率，减少指令缓存未命中。Next Line 预取策略是一种简单而有效的预取方法，通过在当前访问的行之后预取下一行数据。当一个缓存行被访问时，预取引擎会预取相邻的下一行，以期望利用局部性原理，提高数据的连续访问性能。

缓存替换策略：

SHIP 替换策略结合了采样集合和历史引用计数，通过对采样集合进行抽样和动态调整引用计数，以灵活的方式进行缓存替换。它关注监测不同指令地址的访问模式，通过动态调整历史引用计数来适应不同工作负载，提高替换的准确性。SRIP 是一种集关联缓存替换策略，根据每个集中的计数器选择最佳替换目标。当缓存行被引用时，计数器递增；当需要替换缓存行时，选择计数器最低的行进行替换。这使得 SRIP 能够智能地选择适应不同数据访问频率的替换目标。DRRIP 替换策略基于动态引用间隔，动态调整替换策略。通过考虑引用同一缓存行的时间间隔，以动态地调整替换策略。当需要替换缓存行时，选择根据引用间隔调整后的替换策略，可能是选择较短引用间隔的缓存行进行替换，以提高对不同工作负载的适应性。LRU 是一种经典的替换策略，选择最近最少使用的缓存行进行替换。每次缓存访问都更新缓存行的访问时间，以保持对最近使用的缓存行的跟踪。LRU 旨在最大程度地利用局部性原理，但可能在实际实现中带来较高的计算成本。

4 实验结果

4.1 基础部分

- **Cache 预取策略与替换策略的不同组合性能分析**

在本次实验中我们依次进行了不同预取策略与替换策略的组合，经过在模拟器上运行测试得到如下实验数据：

- **不同 L1D 与 L2C 预取策略组合性能分析**

L1D	no	no	no	next_line	next_line	next_line
L2C	no	next_line	ip_stride	no	next_line	ip_stride
IPC.462	0.5015	0.54116	0.69895	0.54842	0.57188	0.66871
IPC.482	0.57324	0.89094	0.98026	0.90044	1.01462	1.12761
IPC.AVG	0.53737	0.71605	0.839605	0.72443	0.79325	0.89816

替换策略为 LRU，LLC 不使用预取策略，对比不同 L1D 与 L2C 预取策略组合的性能，其中性能最好的组合是 next_line[L1D]、ip_stride[L2C]。

- **不同 L1D 与 LLC 预取策略组合性能分析**

替换策略为 LRU，L2C 不使用预取策略，对比不同 L1D 与 LLC 预取策略组合的性能，其中性能最好的组合是 next_line[L1D]、next_line[L2C]。

- **不同 L2C 与 LLC 预取策略组合性能分析**

L1D	no	no	next_line	next_line
LLC	no	next_line	no	next_line
IPC.462	0.5015	0.52377	0.54842	0.90044
IPC.482	0.57324	0.85649	0.58239	1.02
IPC.AVG	0.53737	0.69013	0.565405	0.96022

L2C	no	no	next_line	next_line	ip_stride	ip_stride
LLC	no	next_line	no	next_line	no	next_line
IPC.462	0.5015	0.52377	0.54116	0.58452	0.69895	0.75466
IPC.482	0.57324	0.85649	0.89094	1.00163	0.98026	1.09569
IPC.AVG	0.53737	0.69013	0.71605	0.793075	0.839605	0.925175

替换策略为 LRU，L1D 不使用预取策略，对比不同 L2C 与 LLC 预取策略组合的性能，其中性能最好的组合是 next_line[L2C]、next_line[LLC]。

- **综合考虑 L1D、L2C 与 LLC 预取策略以及替换策略组合性能分析**

预取策略和替换策略如下：

* **L1D 预取**：无预取、next_line

* **L2C 预取**：无预取、next_line、ip_stride

* **LLC 预取**：无预取、next_line、ip_stride

* **替换策略**：LRU、SHIP、SRRIP、DRRIP

综合考虑，以上不同的预取策略和替换策略，一共 $2 \times 3 \times 3 \times 4 = 72$ 种组合方式，排除部分无法编译的情况，我们将剩下的全部数据可视化为下图，在综合考虑 L1D、L2C 与 LLC 预取策略以及替换策略组合时，我们得到 IPC 最高的组合是 next_line[L1D 预取策略]、ip_stride[L2C 预取策略]、next_line[LLC 预取策略]、drrip[Cache 替换策略]。

1. **Cache 替换策略选择 drrip：**

   DRRIP 是一种基于动态访问历史的替换策略，能够动态地调整缓存块的访问优先级。这种策略可能更适应于具有不同访问模式的工作负载，提高了对多样性访问模式的适应能力。

2. **L1D 预取策略选择 next_line：**

   L1D 缓存使用 next_line 预取策略，可以在处理器访问某一行数据时，预先加载相邻的数据行。这种预取策略有助于提高局部性，减少对主内存的访问需求，特别是在具有顺序访问模式的情况下。

3. **L2C 预取策略选择 ip_stride：**

   使用 ip_stride 预取策略可能表明系统能够充分利用指令计数器（IP）的信息，提前加载可能会被执行的指令。这对于指令的预取能力有所帮助，提高了指令缓存的效率。

4. **LLC 预取策略选择 next_line：**

   在 LLC 层面使用 next_line 预取策略可能表明系统更注重提高 LLC 的命中率。通过预取相邻的数据行，可以更好地利用数据的空间局部性，减少 LLC 未命中，从而减少对主内存的访问。

总的来说，这个组合可能在多个缓存层面充分利用了数据和指令的局部性，并通过动态的替换策略适应了不同的访问模式。

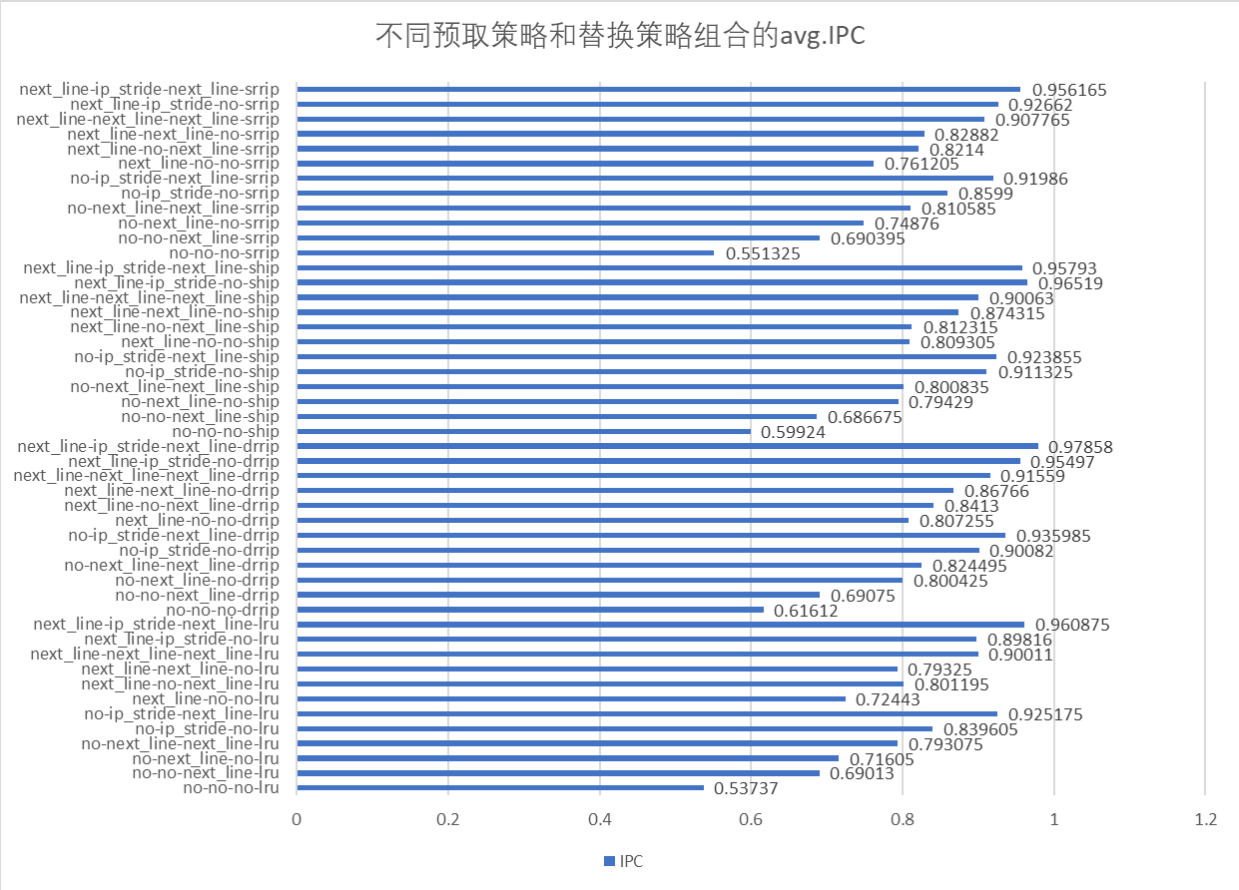

图 4.2: 综合性能分析

4.2 提高部分

在本部分中我们尝试实现基于 GHB 的步长预取算法和 LFU 替换策略，GHB PC/CS 预取策略中的 GHB（全局历史缓冲区）是一种循环队列，存储着 L2 缓存观察到的程序执行历史，每个 GHB 条目包含程序计数器（PC）和对应的缓存行地址（CS）。通过链接指针（prev），GHB 建立了相同索引的地址的时间顺序序列。此外，预取策略还使用 Index Table（IT）通过程序计数器的低位将其映射到 GHB 数组的索引，以便快速查找和更新 GHB 条目。预取操作根据历史访问模式计算步长，当步长相等时执行预取操作，提前预测未来可能的访问模式，从而尝试增加缓存命中率。这种策略动态捕捉了程序执行过程中的局部性和访问模式，利用历史信息优化缓存访问。其数据结构如下：

```
// 结构体定义，表示全局历史缓冲区的条目
struct GHB {
    ull pc;              // 程序计数器
    ull cacheline_addr;  // 缓存行地址
    ui prev;             // 前一个条目的索引
};

// 全局变量和数据结构
static double cache_ac = 0, cache_miss = 0;
```

```
10  static ui cur_idx = 0;
11  static GHB my_GHB[GHB_SIZE];
12  static std::map<int, ui> it;  // 索引表，将程序计数器的低位映射到 GHB 数组的索引
```

LFU（Least Frequently Used）替换策略的算法设计思路如下：

在缓存初始化阶段，不需要进行特殊的 LFU 计数的初始化工作。当需要替换缓存中的数据块时，首先尝试找到一个无效的块（即未被使用的缓存行）。如果存在无效行，选择其中一个用于新的数据块。如果所有行都已被使用，从已被使用的行中选择 LFU 计数最小的行进行替换。LFU 计数表示数据块被访问的次数，计数越小表示该块访问频率越低。在每次缓存命中时，更新 LFU 计数。只有在缓存命中的情况下才更新计数，因为 LFU 替换策略主要关注于选择访问频率最低的块进行替换。更新 LFU 计数的方式可以是简单的自增，表示该块被访问了一次。可以在相关函数（如 llc_replacement_final_stats）中收集统计信息，但 LFU 一般不需要特殊的统计信息，因为其核心思想是选择访问频率最低的块。

经过测试，实验数据如下，从测得的数据来看，自行实现的预取和替换策略较现有的方法性能更好，但在本次实验中还有一些值得进一步探索的问题，比如部分组合测试时无法得到数据的原因等。

L1D	no	no	next_line	next_line
LLC	no	next_line	no	next_line
IPC.462	0.94028	0.93783	0.94469	0.94882
IPC.482	1.15022	1.15022	-	-
IPC.AVG	1.04525	1.044025	0.94469	0.94882

本次的实验代码已上传至 Github 仓库，链接为 https://github.com/Yuzhao-P/C-A