缓存替换策略
从基础到前沿

Cache Replacement Policies: From Fundamentals to State-of-the-Art

硬件与软件缓存替换算法全面解析

涵盖 LRU、LFU、ARC、S3-FIFO、SIEVE、DRRIP、Hawkeye、SHiP 等

2026年2月

目录

参考文献

第1章 缓存基础

1.1 缓存的基本概念

缓存（Cache）是计算机系统中一种关键的高速存储层次，用于弥合快速处理器与相对较慢的主存储器之间的速度差距。从CPU寄存器到磁盘缓存，从操作系统页缓存到分布式系统中的内容分发网络（CDN），缓存技术无处不在。

缓存的核心思想基于程序访问的局部性原理：

> **定义 1.1：局部性原理**
>
> **时间局部性（Temporal Locality）**：如果一个数据项被访问，那么它在不久的将来很可能再次被访问。
>
> **空间局部性（Spatial Locality）**：如果一个数据项被访问，那么与它相邻的数据项很可能在不久的将来被访问。

缓存系统通常由三个关键组件构成：

- **缓存存储器**：容量有限但访问速度快的高速存储区域
- **映射策略**：决定数据块如何放置到缓存中（直接映射、组相联、全相联）
- **替换策略**：当缓存满时，决定替换哪个数据块

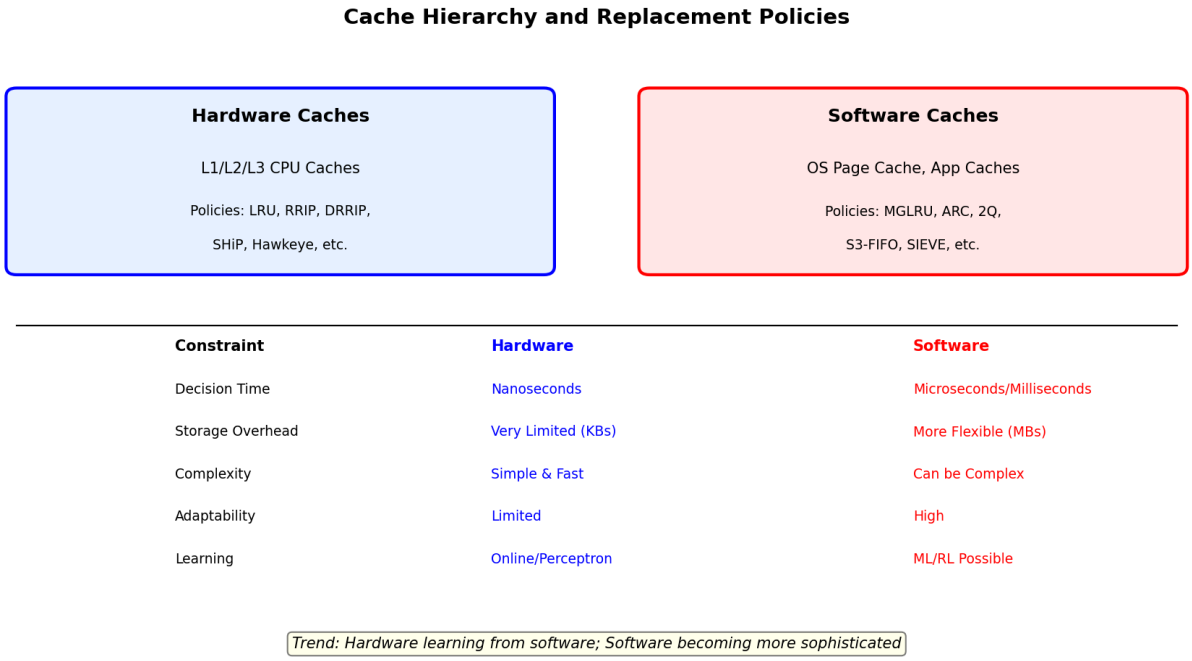

图 1-1 缓存层次结构与替换策略分类

1.2 缓存替换问题

缓存替换问题是缓存管理中最核心的决策问题。当缓存已满且需要加载新的数据块时，替换策略必须决定哪个现有数据块应该被驱逐（evict）。理想的替换策略应该最大化缓存命中率，即尽可能让后续访问在缓存中找到所需数据。

> **Belady最优算法（MIN）**
>
> Belady在1966年证明了最优的缓存替换策略：当需要替换时，选择**下一次访问距离最远**的数据块进行驱逐。这个算法被称为Belady's MIN或OPT算法。

然而，Belady最优算法在实际中是不可实现的，因为它需要预知未来的访问序列。因此，所有的实际替换策略都是基于启发式方法，试图近似最优算法的行为。

1.3 评估指标

评估缓存替换策略的主要指标包括：

表 1-1 缓存替换策略评估指标

指标	定义	说明
命中率（Hit Ratio）	命中次数 / 总访问次数	最重要的性能指标
缺失率（Miss Ratio）	1 - 命中率	需要访问慢速存储的比例
平均访问时间	命中时间 × 命中率 + 缺失时间 × 缺失率	综合性能指标
实现复杂度	时间复杂度和空间复杂度	影响实际部署
扫描抵抗性	抵抗顺序扫描的能力	防止一次性访问污染缓存

第2章 基本替换策略

本章介绍最基础的缓存替换策略，这些策略构成了理解更复杂算法的基础。尽管它们的设计简单，但在许多场景下仍然表现良好。

2.1 FIFO（先进先出）

FIFO（First-In-First-Out）是最简单的缓存替换策略。它按照数据块进入缓存的先后顺序进行管理，当需要替换时，选择最早进入缓存的数据块进行驱逐。

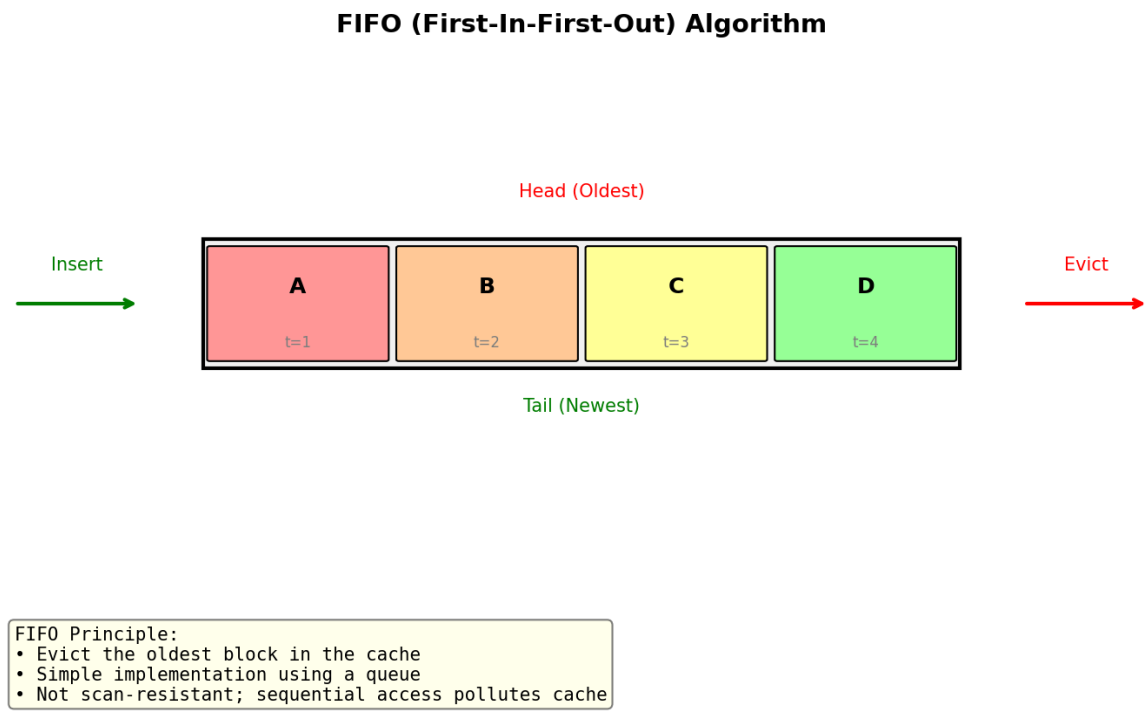

图 2-1 FIFO算法工作原理

算法 2.1：FIFO替换策略

```
当访问数据块 x 时:
    如果 x 在缓存中:
        返回命中
    否则:
        如果缓存已满:
            驱逐队列头部的数据块
        将 x 插入队列尾部
        返回缺失
```

优点：

- 实现极其简单，只需维护一个队列
- 时间复杂度为 O(1)
- 无需在每次访问时更新元数据

缺点：

- 完全不考虑访问频率或最近访问情况
- 对扫描型访问模式（sequential scan）没有抵抗力
- 可能驱逐仍然频繁使用的数据块

2.2 LRU（最近最少使用）

LRU（Least Recently Used）基于时间局部性原理，认为最近被访问的数据块在不久的将来很可能再次被访问。当需要替换时，LRU选择最长时间未被访问的数据块进行驱逐。

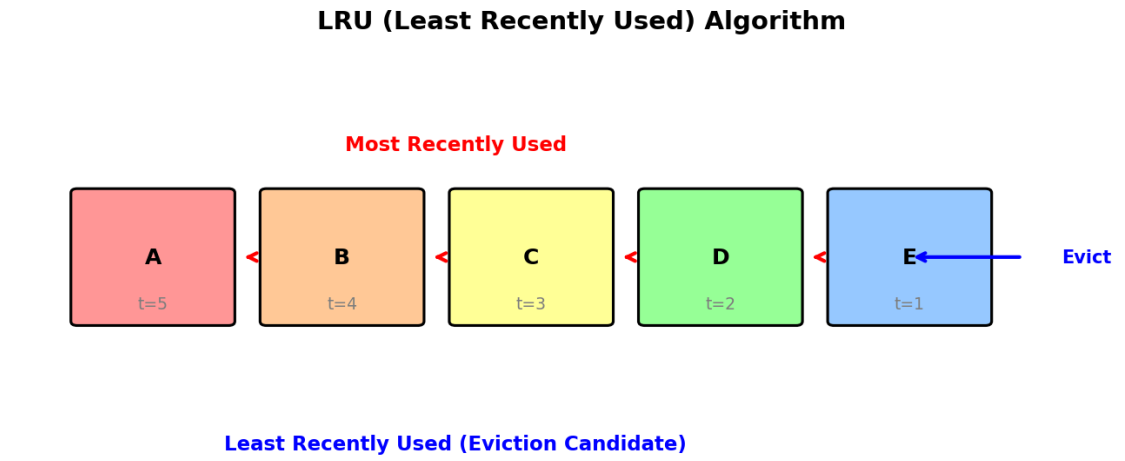

图 2-2 LRU算法工作原理

算法 2.2：LRU替换策略

```
当访问数据块 X 时:
    如果 X 在缓存中:
        将 X 移动到链表头部
        返回命中
    否则:
        如果缓存已满:
            驱逐链表尾部的数据块
        将 X 插入链表头部
        返回缺失
```

LRU通常使用双向链表和哈希表的组合来实现，保证 O(1) 的时间复杂度。然而，LRU也存在一些问题：

- **扫描抵抗性差**：顺序扫描会迅速填满缓存，并将原有的热数据驱逐出去

2.3 LFU（最少使用频率）

LFU（Least Frequently Used）基于访问频率来决定替换。它维护每个数据块的访问计数，当需要替换时，选择访问次数最少的数据块进行驱逐。

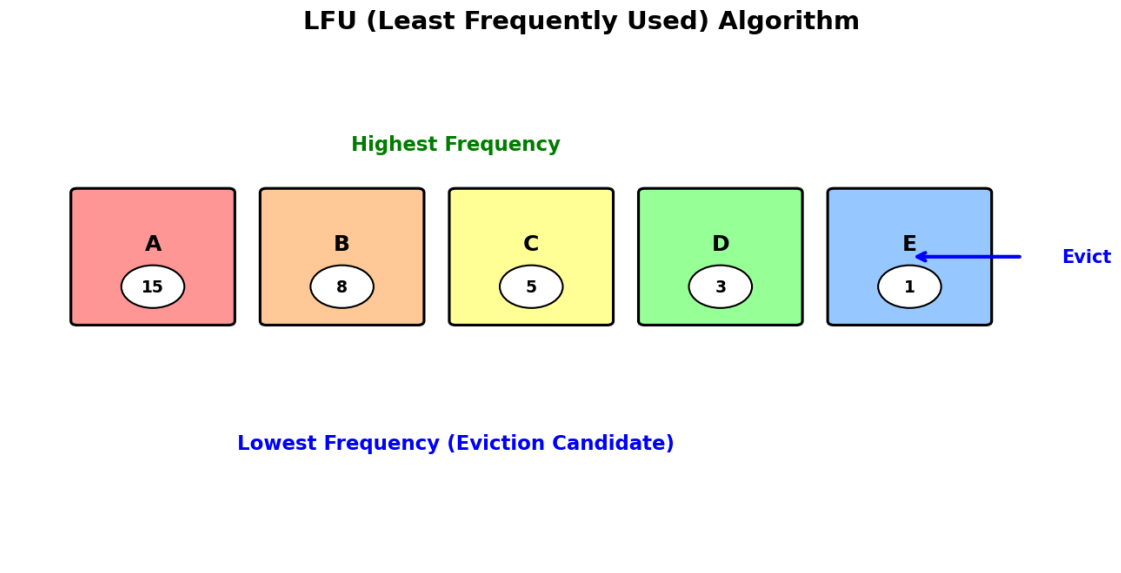

图 2-3 LFU算法工作原理

算法 2.3：LFU替换策略

```
当访问数据块 X 时:
    如果 X 在缓存中:
        X.count += 1
        返回命中
    否则:
        如果缓存已满:
            驱逐计数最小的数据块
        将 X 插入缓存，X.count = 1
        返回缺失
```

- **对新数据不友好**：新加入的数据块需要很长时间才能积累足够的计数
- **实现复杂度高**：需要维护有序结构来快速找到最小计数，通常为 O(log n)

为了解决这些问题，提出了多种LFU的改进版本，如LFU-Aging、LFU*等，通过定期衰减计数来适应访问模式的变化。

2.4 CLOCK算法

CLOCK算法（也称为Second Chance算法）是对LRU的一种近似实现，使用更少的内存开销。它将缓存块组织成一个环形缓冲区，每个块有一个引用位（reference bit）。

算法 2.4：CLOCK替换策略

```
初始化：所有引用位设为0，时钟指针指向任意位置

当访问数据块 X 时：
    如果 X 在缓存中：
        X.ref = 1
        返回命中
    否则：
        循环：
            如果时钟指向的块.ref == 0：
                驱逐该块，插入X，X.ref = 0
                时钟指针前移
                返回缺失
            否则：
                该块.ref = 0
                时钟指针前移
```

CLOCK算法的优势在于：

- 使用数组而非链表，内存开销更小
- 无需在每次命中时移动数据块
- 实现简单，适合硬件实现

研究表明，CLOCK在许多工作负载下不仅比LRU开销更低，而且命中率也更好。这启发了后续许多基于CLOCK的改进算法。

第3章 软件缓存替换策略

软件缓存（如操作系统页缓存、应用程序缓存、数据库缓冲池等）与硬件缓存相比，具有更大的灵活性。软件缓存可以采用更复杂的算法，使用更多的元数据，甚至可以利用机器学习技术。本章介绍几种最重要的软件缓存替换策略。

3.1 ARC（自适应替换缓存）

ARC（Adaptive Replacement Cache）由Megiddo和Modha于2003年提出，是一种自调优的缓存替换算法。ARC的核心思想是同时维护最近使用（recency）和频繁使用（frequency）两个列表，并根据工作负载动态调整它们之间的平衡。

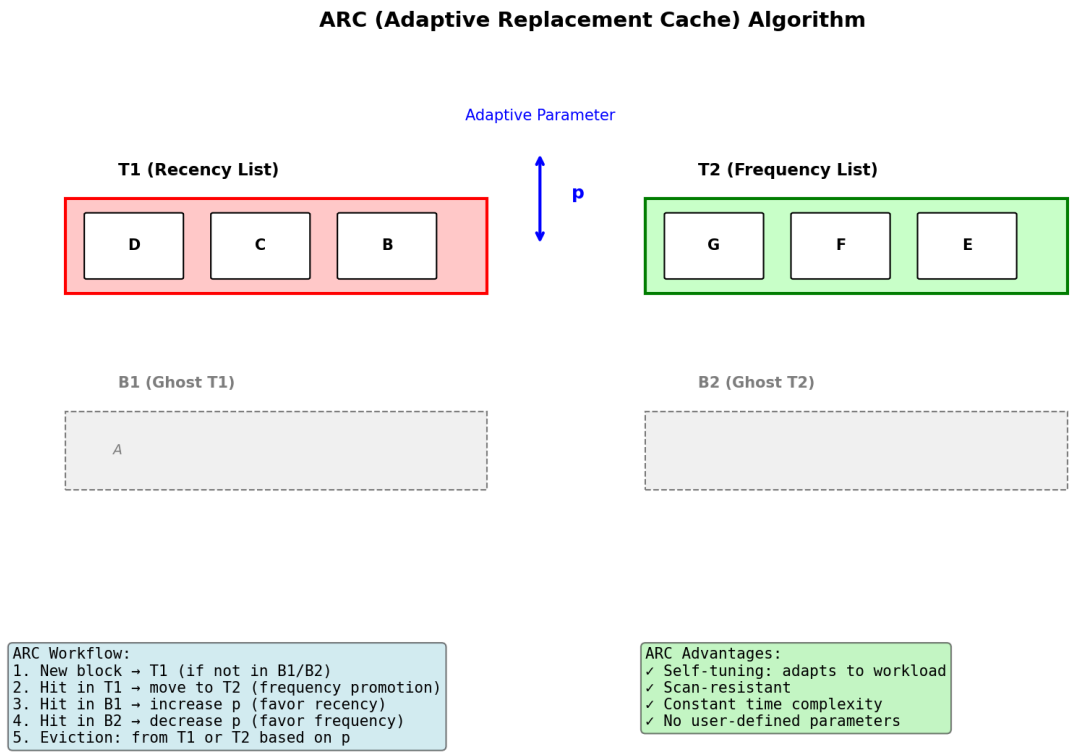

图 3-1 ARC算法结构示意图

3.1.1 ARC的结构

ARC维护四个列表：

- **T1**：最近访问一次的页面列表（LRU顺序）

3.1.2 工作流程

算法 3.1：ARC替换策略

```
当请求页面 X 时：
    情况1：X 在 T1 或 T2 中（命中）
        将 X 移动到 T2 的MRU位置

    情况2：X 在 B1 中（幽灵命中T1）
        // 说明最近访问的页面有价值
        增加 p：p = min(p + max(B2.length/B1.length, 1), cache_size)
        从 B1 移除 X，将 X 放入 T2 的MRU位置
        执行替换（见下文）

    情况3：X 在 B2 中（幽灵命中T2）
        // 说明频繁访问的页面有价值
        减少 p：p = max(p - max(B1.length/B2.length, 1), 0)
        从 B2 移除 X，将 X 放入 T2 的MRU位置
        执行替换

    情况4：X 不在任何列表中（冷缺失）
        如果 T1.length + B1.length == cache_size：
            如果 T1.length < cache_size：
                从 B1 驱逐LRU页面
                执行替换
            否则：
                从 T1 驱逐LRU页面
        否则如果 T1.length + B1.length < cache_size：
            如果 T1.length + T2.length + B1.length + B2.length >= cache_size：
                    如果 T1.length + T2.length + B1.length + B2.length ==
2*cache_size：
                    从 B2 驱逐LRU页面
                执行替换
        将 X 放入 T1 的MRU位置

替换过程：
    如果 T1.length > 0 且 (T1.length > p 或 (X 在 B2 中且 T1.length == p))：
        从 T1 驱逐LRU页面到 B1
    否则：
        从 T2 驱逐LRU页面到 B2
```

3.1.3 ARC的优势

ARC的关键特性

- **自适应性**：自动调整以适应工作负载的变化
- **扫描抵抗性**：一次性扫描不会污染缓存
- **常数时间复杂度**：每个请求的处理时间为 O(1)
- **无需调参**：不需要用户指定任何参数
- **性能保证**：在实践中表现至少与调优后的LRU-2相当

ARC已被广泛应用于各种系统中，包括ZFS文件系统、PostgreSQL数据库等。研究表明，在多种工作负载下，ARC的命中率可以比传统LRU提高10%以上。

3.2 2Q算法

2Q（Two Queue）算法由Johnson和Shasha于1994年提出，旨在解决LRU的扫描抵抗问题。2Q使用两个队列来区分短期访问和长期访问的数据块。

3.2.1 2Q的结构

2Q将缓存分为三个部分：

- **Am（Main LRU）**：主LRU队列，占缓存大小的 3/4
- **A1in（Small FIFO）**：小型FIFO队列，占缓存大小的 1/4
- **A1out（Ghost FIFO）**：幽灵FIFO队列，只存储元数据，大小约为缓存的一半

3.2.2 工作流程

算法 3.2：2Q替换策略

```
当访问数据块 X 时:
    如果 X 在 Am 中:
        将 X 移动到 Am 的MRU位置
        返回命中

    如果 X 在 A1in 中:
        // 不做任何操作，保持在A1in中
        返回命中

    如果 X 在 A1out 中:
        // 曾经被驱逐但又被访问，说明是热数据
        从 A1out 移除 X
        如果 Am 已满:
            从 Am 驱逐LRU页面到 A1out
        将 X 放入 Am 的MRU位置
        返回缺失

    // X 不在任何队列中（冷缺失）
    如果 A1in 已满:
        从 A1in 驱逐FIFO页面到 A1out
    将 X 放入 A1in 的尾部
    返回缺失
```

2Q的核心思想是使用A1in作为过滤器：只有那些在A1in中被驱逐后又被重新访问的数据块才能进入主缓存Am。这有效地防止了扫描型访问污染缓存。

3.2.3 2Q的局限性

- **高LRU开销**：Am使用LRU，需要维护链表，CPU和内存开销较高
- **A1in过大**：25%的缓存分配给A1in，在现代大缓存环境下可能过大
- **热数据额外缺失**：热数据块必须先经过A1in和A1out才能进入Am，造成额外的一次缺失

3.3 S3-FIFO

S3-FIFO（Simple and Scalable FIFO-based Cache）由Yang等人于2023年提出，是一种基于FIFO队列的新型缓存替换算法。S3-FIFO的设计目标是简单、可扩展且具有高命中率。

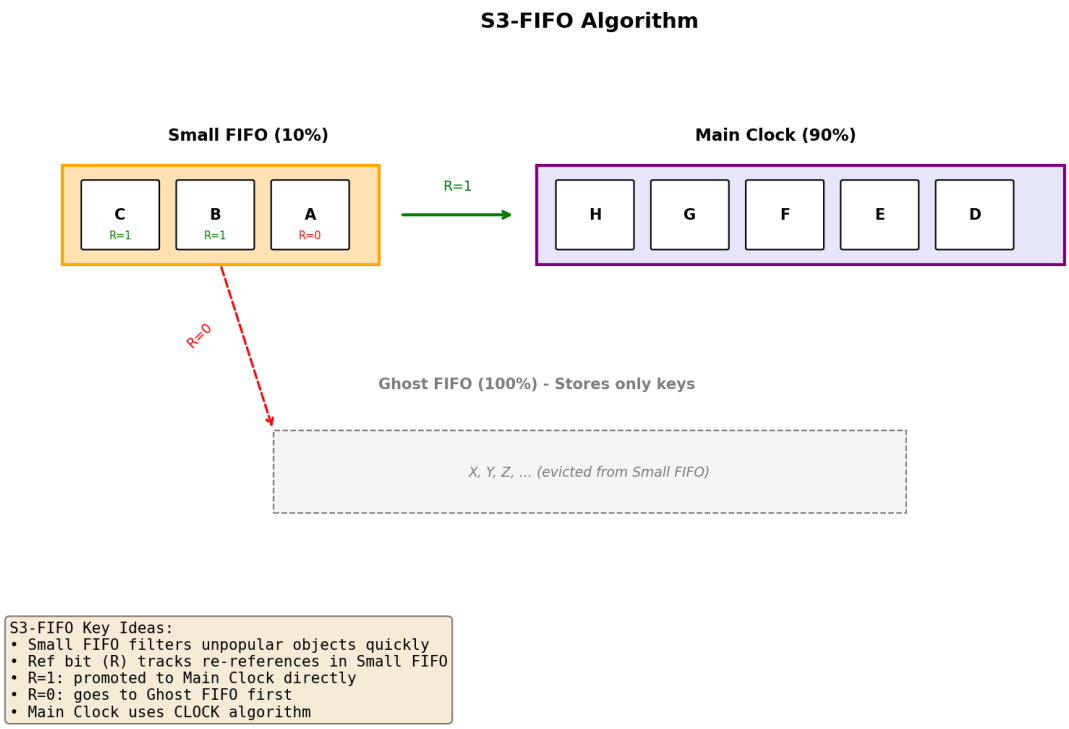

图 3-2 S3-FIFO算法结构示意图

3.3.1 S3-FIFO的结构

S3-FIFO使用三个队列：

- **Small FIFO**：占缓存大小的10%，用于快速过滤不热门的数据块
- **Main Clock**：占缓存大小的90%，使用CLOCK算法管理
- **Ghost FIFO**：大小等于缓存大小，只存储被驱逐数据块的元数据

每个在Small FIFO中的数据块有一个引用位（Ref bit），用于跟踪是否被重新访问。

3.3.2 工作流程

算法 3.3：S3-FIFO替换策略

```
当访问数据块 X 时:
    如果 X 在 Main Clock 中:
        X.ref = 1
        返回命中

    如果 X 在 Small FIFO 中:
        X.ref = 1
        返回命中

    如果 X 在 Ghost FIFO 中:
        从 Ghost FIFO 移除 X
        如果 Main Clock 已满:
            使用CLOCK算法驱逐一个页面
        将 X 插入 Main Clock
        返回缺失

    // X 不在任何队列中（冷缺失）
    如果 Small FIFO 已满:
        Y = Small FIFO 的头部页面
        从 Small FIFO 移除 Y
        如果 Y.ref == 1:
            // 被重新访问过，是热数据
            如果 Main Clock 已满:
                使用CLOCK算法驱逐一个页面
            将 Y 插入 Main Clock
        否则:
            // 冷数据，放入Ghost FIFO
            将 Y 插入 Ghost FIFO

    将 X 插入 Small FIFO 的尾部，X.ref = 0
    返回缺失
```

3.3.3 S3-FIFO的优势

S3-FIFO解决了2Q的几个关键问题：

- **低CPU开销**：使用FIFO和CLOCK，避免了LRU的链表操作
- **合理的队列大小**：Small FIFO只占10%，Main Clock占90%
- **无额外热数据缺失**：通过Ref位，热数据可以直接从Small FIFO提升到Main Clock

研究表明，S3-FIFO在多种工作负载下的命中率优于ARC和LRU，同时具有更低的CPU开销。

3.4 SIEVE

SIEVE是2024年提出的一种极其简单的缓存替换算法，其设计理念是"比LRU更简单，比LRU更高效"。SIEVE基于CLOCK算法，但采用了不同的指针移动策略。

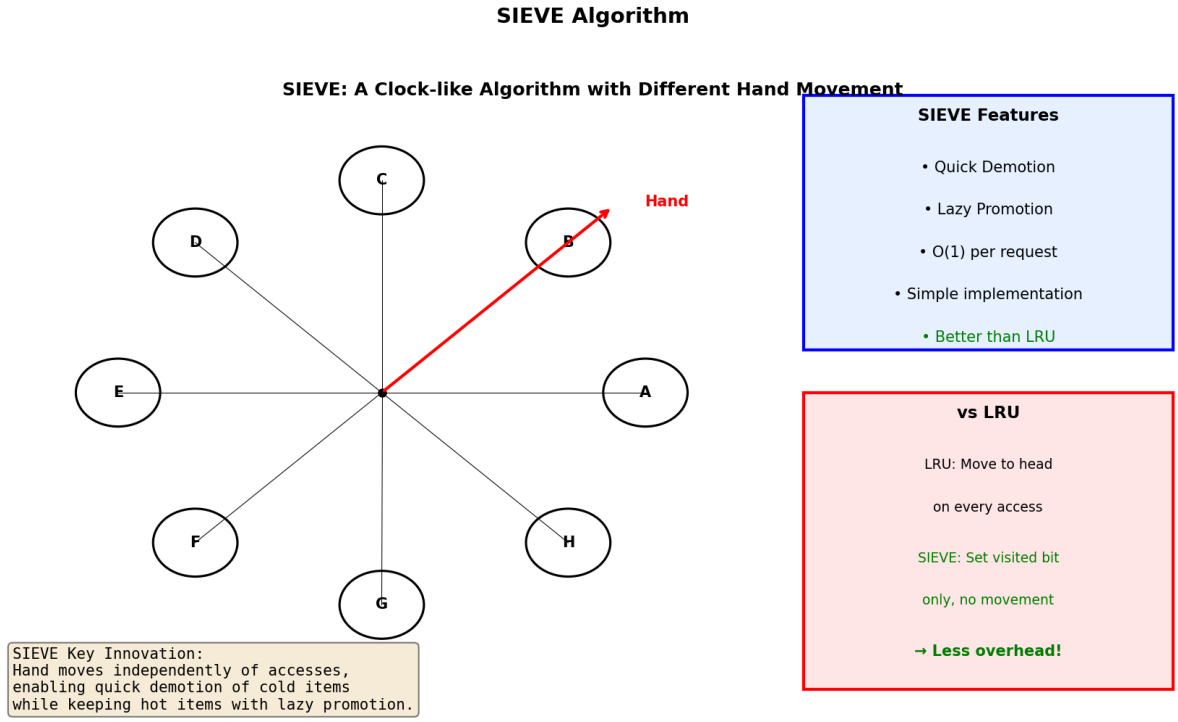

图 3-3 SIEVE算法工作原理

3.4.1 SIEVE的核心思想

SIEVE引入了两种关键机制：

- **Quick Demotion（快速降级）**：时钟指针独立移动，可以快速识别并驱逐冷数据
- **Lazy Promotion（延迟提升）**：只在驱逐检查时更新访问状态，而非每次访问都更新

算法 3.4：SIEVE替换策略

```
初始化：时钟指针指向缓存的任意位置

当访问数据块 X 时：
    如果 X 在缓存中：
        X.visited = true
        返回命中
    否则：
        如果缓存已满：
            循环直到找到驱逐候选：
                如果当前指针位置的块.visited == false：
                    驱逐该块
                    在该位置插入 X
                    X.visited = false
                    指针前移
                    跳出循环
                否则：
                    当前块.visited = false
                    指针前移
        否则：
            在任意空闲位置插入 X
            X.visited = false
        返回缺失
```

3.4.2 SIEVE的优势

SIEVE的关键特性

- **极简实现**：只需修改几行代码即可将LRU替换为SIEVE
- **O(1)复杂度**：每个请求的处理时间为常数
- **高命中率**：在超过45%的测试 traces 上优于所有现有算法
- **低开销**：无需链表操作，只需设置一个标志位

大规模评估表明，SIEVE可以将FIFO的缺失率降低超过42%，相比ARC平均降低1.5%，最高可降低63.2%。

3.5 MGLRU（多代LRU）

MGLRU（Multi-Generation LRU）是Google为Linux内核开发的新型页替换算法，旨在解决传统LRU在复杂工作负载下的性能问题。MGLRU已被合并到Linux内核6.1版本。

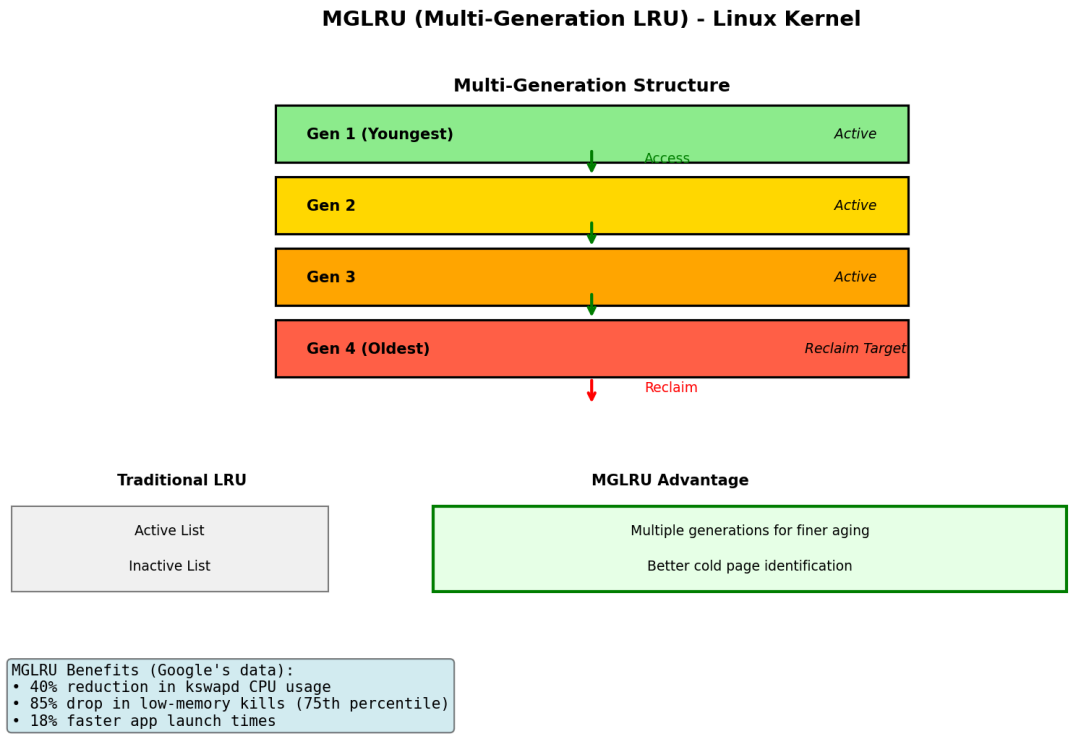

图 3-4 MGLRU多代结构示意图

3.5.1 MGLRU的设计动机

传统Linux页替换使用两个LRU列表（active/inactive），存在以下问题：

- 高CPU使用率（kswapd进程消耗大量CPU）
- 不必要的页面扫描
- 在内存受限设备上容易出现OOM（Out of Memory）杀死进程

3.5.2 MGLRU的结构

MGLRU使用多个世代（generation）来跟踪页面的使用情况：

- **Gen 1（最年轻）**：最近访问的页面

页面在不同世代之间移动基于访问情况。当页面被访问时，它被提升到最年轻的世代。当需要回收内存时，从最老的世代开始扫描。

3.5.3 MGLRU的优势

Google在其Chrome OS和Android设备上的测试表明：

- **kswapd CPU使用率降低40%**
- **低内存杀死事件减少85%**（第75百分位）
- **应用启动时间提升18%**（第50百分位）

MGLRU的多代设计使得系统能够更准确地识别真正的冷页面，减少不必要的扫描，从而提高整体性能。

第4章 硬件缓存替换策略

硬件缓存（如CPU的L1/L2/L3缓存）与软件缓存面临不同的约束和挑战。硬件缓存需要在纳秒级别做出替换决策，因此算法必须极其简单且易于硬件实现。本章介绍几种最先进的硬件缓存替换策略。

4.1 RRIP与DRRIP

RRIP（Re-Reference Interval Prediction）是由Jaleel等人于2010年提出的一类硬件缓存替换策略。RRIP通过预测数据块的重新引用间隔来做出替换决策。

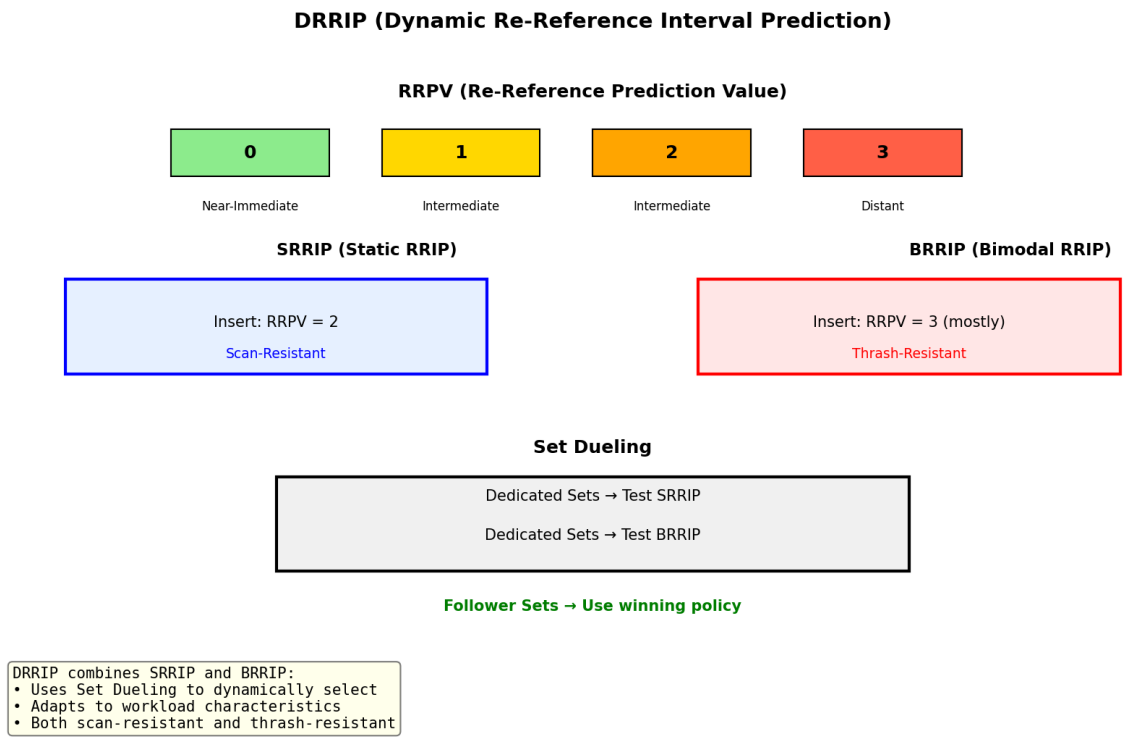

图 4-1 DRRIP算法与RRPV值说明

4.1.1 RRPV（重新引用预测值）

RRIP为每个缓存块维护一个2位的RRPV（Re-Reference Prediction Value），表示预测的重新引用间隔：

表 4-1 RRPV值含义

RRPV	含义	描述
0	Near-Immediate	预计很快会被重新访问
1	Intermediate	预计中等时间后被访问
2	Intermediate	预计较长时间后被访问
3	Distant	预计很久之后或不会被访问

4.1.2 SRRIP（静态RRIP）

算法 4.1：SRRIP替换策略

```
初始化：所有RRPV设为最大值（3）

当访问缓存块 X 时：
    如果 X 在缓存中（命中）：
        X.RRPV = 0  // 预测很快会被再次访问
        返回命中
    否则（缺失）：
        循环直到找到驱逐候选：
            在当前集合中查找RRPV == 3的块
            如果找到：
                驱逐该块
                插入X，X.RRPV = 2  // 静态插入为Intermediate
                返回缺失
            否则：
                将集合中所有块的RRPV加1（饱和到3）
```

SRRIP具有扫描抵抗性，因为新插入的块RRPV为2，需要经过几次扫描才会被驱逐。但如果工作负载具有突发性（thrashing），SRRIP表现不佳。

4.1.3 BRRIP（双模态RRIP）

BRRIP（Bimodal RRIP）针对thrashing工作负载进行了优化。它大多数时间将新块的RRPV设为3（Distant），偶尔设为2：

算法 4.2：BRRIP替换策略

```
参数：ε = 1/32（小概率）

当插入新块 X 时：
    以概率 1-ε：X.RRPV = 3（Distant）
    以概率 ε：X.RRPV = 2（Intermediate）
```

BRRIP的thrash抵抗机制：在thrashing情况下，大多数新块会被赋予RRPV=3，只有少量"幸存者"能获得RRPV=2，从而保护这些幸存者不被快速驱逐。

4.1.4 DRRIP（动态RRIP）

DRRIP（Dynamic RRIP）使用Set Dueling机制在SRRIP和BRRIP之间动态选择。Set Dueling的核心思想是：

- 将部分缓存集合（sets）专门用于测试SRRIP
- 将部分缓存集合专门用于测试BRRIP
- 其余的"跟随者"集合（follower sets）使用表现更好的策略

算法 4.3：DRRIP的Set Dueling机制

```
初始化：
    选择32个集合作为SRRIP专用集合
    选择32个集合作为BRRIP专用集合
    其余为跟随者集合
    policy_selector = 0（偏向SRRIP）

当发生缓存缺失时：
    如果当前集合是SRRIP专用：
        使用SRRIP
        如果是 thrash-resistant miss：
            policy_selector -= 1（偏向BRRIP）
    否则如果当前集合是BRRIP专用：
        使用BRRIP
        如果是 scan-resistant miss：
            policy_selector += 1（偏向SRRIP）
    否则（跟随者集合）：
        如果 policy_selector >= 0：
            使用SRRIP
        否则：
            使用BRRIP
```

DRRIP结合了SRRIP的扫描抵抗性和BRRIP的thrash抵抗性，能够适应不同的工作负载特性。

4.2 SHiP

SHiP（Signature-based Hit Predictor）由Wu等人于2011年提出，是一种基于签名的命中预测器。SHiP在RRIP的基础上增加了基于历史行为的预测机制。

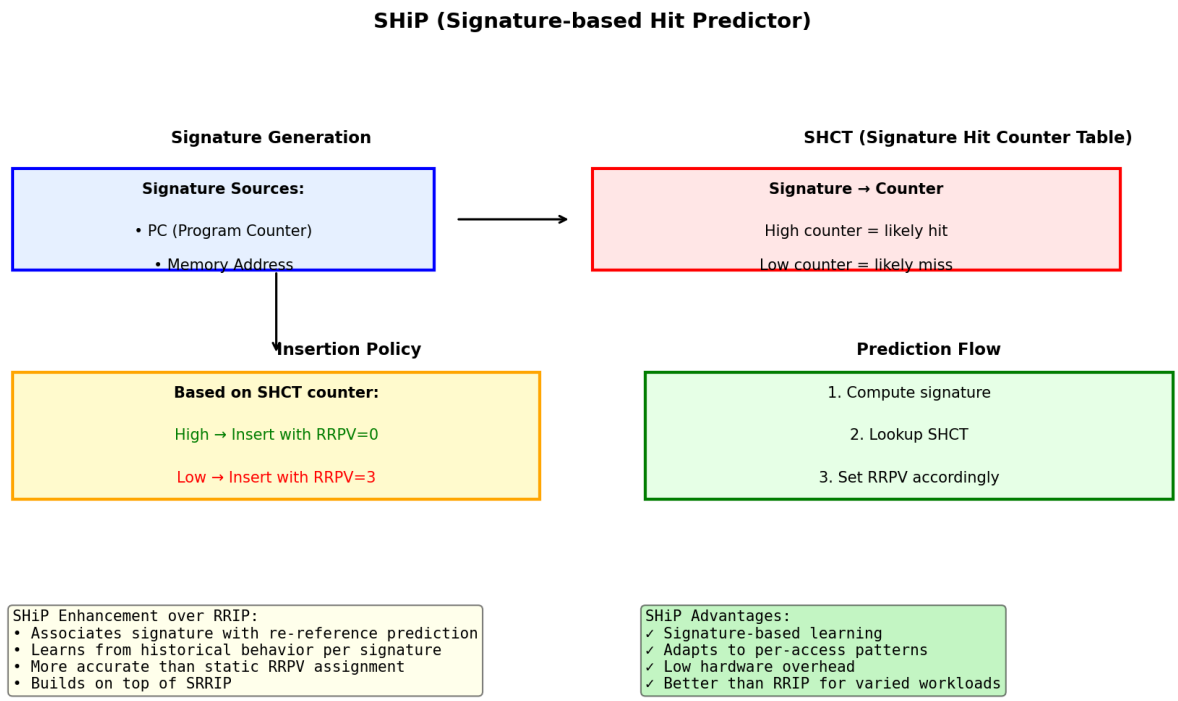

图 4-2 SHiP算法结构示意图

4.2.1 SHiP的核心思想

SHiP的核心洞察是：不同内存访问模式（由程序计数器PC或内存地址标识）具有不同的重新引用特性。SHiP通过学习这些模式来做出更准确的插入决策。

4.2.2 SHiP的结构

SHiP包含两个主要组件：

- **签名生成器**：基于PC或内存地址生成签名
- **SHCT（Signature Hit Counter Table）**：记录每个签名的历史命中情况

算法 4.4：SHiP替换策略

```
初始化：SHCT所有计数器设为0

当访问缓存块 X（由签名 S 访问）时：
    如果 X 在缓存中（命中）：
        X.RRPV = 0
        如果 SHCT[S] < 最大值：
            SHCT[S] += 1  // 增加该签名的命中计数
        返回命中
    否则（缺失）：
        使用RRIP找到驱逐候选 Y
        驱逐 Y

        // 根据SHCT决定插入RRPV
        如果 SHCT[S] == 0：
            X.RRPV = 3  // 从未命中过，设为Distant
        否则：
            X.RRPV = 2  // 有过命中历史，设为Intermediate
            如果 SHCT[S] > 0：
                SHCT[S] -= 1  // 衰减计数器

        插入 X
        返回缺失
```

4.2.3 SHiP的优势

- **签名级学习**：通过学习访问模式而非单个块的行为，提高了泛化能力
- **低开销**：SHCT通常只需要64KB存储
- **兼容性强**：可以构建在任何基于RRIP的策略之上

4.3 Hawkeye

Hawkeye由Jain和Lin于2016年提出，是一种从Belady最优算法学习的缓存替换策略。Hawkeye使用监督学习来预测缓存块是否应该被保留。

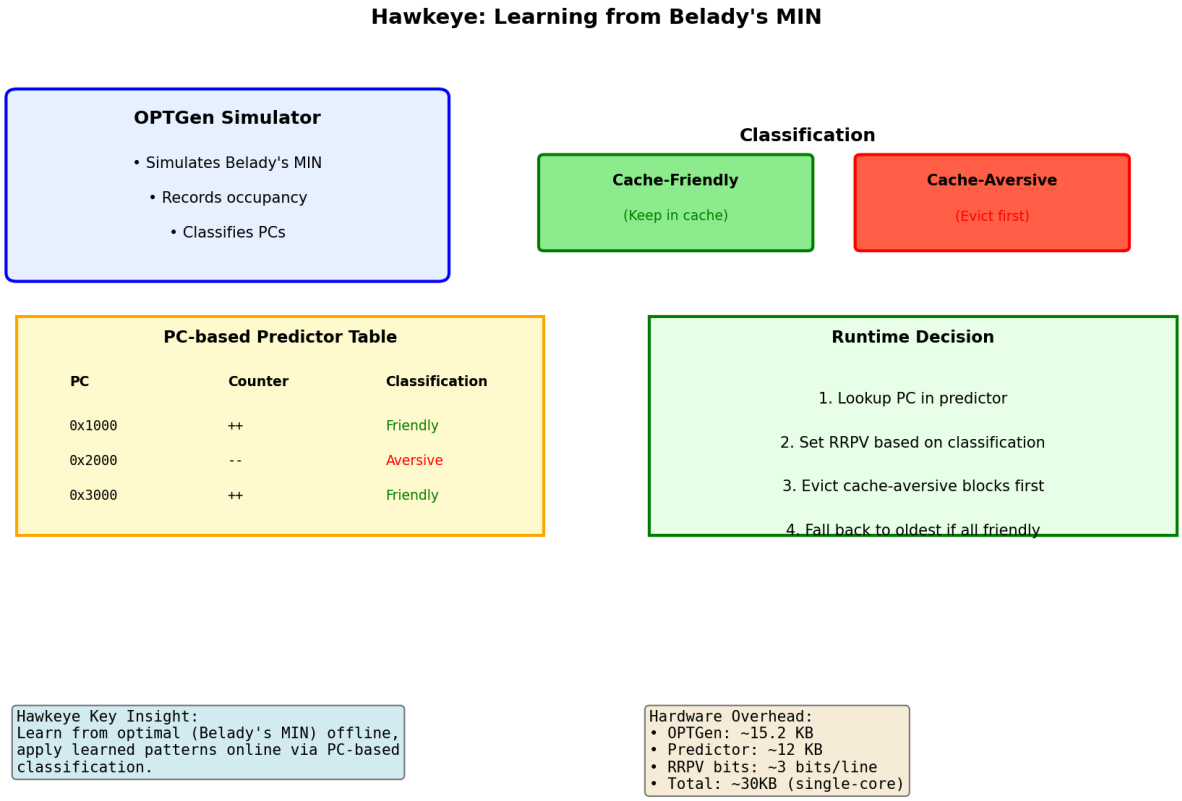

图 4-3 Hawkeye算法工作原理

4.3.1 Hawkeye的核心思想

Hawkeye的核心洞察是：虽然Belady最优算法需要预知未来，但我们可以从历史访问中学习，识别哪些访问模式倾向于产生缓存友好的块，哪些产生缓存不友好的块。

Hawkeye将加载指令（由PC标识）分为两类：

- **Cache-Friendly（缓存友好）**：该PC加载的块应该保留在缓存中
- **Cache-Aversive（缓存不友好）**：该PC加载的块应该被优先驱逐

4.3.2 OPTGen模拟器

Hawkeye使用OPTGen（Optimal Generator）来离线模拟Belady最优算法，并生成训练标签：

算法 4.5：OPTGen工作原理

```
OPTGen模拟Belady最优算法：
    对于每个访问，记录缓存集合的占用情况（occupancy）
    如果当前访问在Belady最优下是命中：
        标记该PC为Cache-Friendly
    否则：
        标记该PC为Cache-Aversive

占用情况计算：
    对于访问序列中的每个地址，计算其重用间隔内
    最多有多少个不同的地址被访问（最大重叠数）
    如果最大重叠数 >= 缓存相联度，则为缺失
```

4.3.3 Hawkeye的预测器

Hawkeye使用PC作为特征，构建一个二进制分类器。在实际硬件中，使用一个简单的计数器表来实现：

算法 4.6：Hawkeye运行时策略

```
预测器表：PC → 计数器（正数表示Friendly，负数表示Aversive）

当由PC P访问缓存块 X 时：
    如果 X 在缓存中：
        返回命中
    否则：
        查找预测器表中P的计数器
        如果计数器 > 0：
            // Cache-Friendly
            插入X，RRPV = 0（Near-Immediate）
        否则：
            // Cache-Aversive
            插入X，RRPV = 3（Distant）

        // 驱逐时优先驱逐Cache-Aversive的块
        如果集合中有RRPV == 3的块：
            驱逐其中一个
        否则：
            驱逐最老的块
```

4.3.4 Hawkeye的硬件开销

表 4-2 Hawkeye硬件开销

组件	单核	四核
OPTGen	15.2 KB	~45 KB
预测器表	12 KB	~36 KB
RRPV位	3 bits/line	3 bits/line
总计	**~30 KB**	**~90 KB**

Hawkeye代表了硬件缓存替换策略的一个重要方向：从最优算法中学习，并将学到的知识蒸馏到轻量级预测器中。后续工作如Glider进一步使用深度学习来改进预测准确性。

第5章 算法比较与总结

本章对前面介绍的各种缓存替换策略进行综合比较，并提供算法选择的指导建议。

5.1 硬件与软件缓存的区别

硬件缓存和软件缓存在设计约束和优化目标上存在显著差异：

Cache Replacement Algorithm Comparison

Algorithm	Type	Scan-Resistant	Complexity	Key Feature
FIFO	Basic	✗	O(1)	Simple queue
LRU	Basic	✗	O(1)	Temporal locality
LFU	Basic	✗	O(log n)	Frequency-based
CLOCK	Basic	✗	O(1)	LRU approximation
ARC	Software	✓	O(1)	Adaptive LRU/LFU
2Q	Software	✓	O(1)	Two-queue filter
S3-FIFO	Software	✓	O(1)	Small FIFO + Clock
SIEVE	Software	✓	O(1)	Quick demotion
MGLRU	Software	✓	O(1)	Multi-generation
RRIP	Hardware	✓	O(1)	Re-reference pred.
DRRIP	Hardware	✓	O(1)	Dynamic RRIP
SHiP	Hardware	✓	O(1)	Signature-based
Hawkeye	Hardware	✓	O(1)	Learn from MIN

Software: Red | Hardware: Blue | Basic: Gray

图 5-1 缓存替换算法综合比较

表 5-1 硬件缓存与软件缓存对比

特性	硬件缓存（CPU L1/L2/L3）	软件缓存（OS/应用）
决策延迟	纳秒级（<1ns）	微秒到毫秒级
存储开销	极有限（KB级）	相对灵活（MB级）
算法复杂度	必须简单（O(1)）	可以较复杂
元数据	每位都很珍贵	可以更宽松
学习能力	在线学习/感知机	可以使用ML/RL
部署灵活性	硬件固定，难以修改	软件可动态调整

5.2 算法选择指南

选择合适的缓存替换策略需要考虑工作负载特性、系统约束和性能目标。以下是针对不同场景的建议：

5.2.1 软件缓存选择

表 5-2 软件缓存算法选择指南

场景	推荐算法	理由
通用应用缓存	SIEVE 或 S3-FIFO	简单、高效、高命中率
数据库缓冲池	ARC 或 2Q	自适应、扫描抵抗
操作系统页缓存	MGLRU	低CPU开销、高扩展性
CDN/ Web缓存	SIEVE	简单部署、性能优异
文件系统缓存	ARC	ZFS等已验证

5.2.2 硬件缓存选择

表 5-3 硬件缓存算法选择指南

场景	推荐算法	理由
通用处理器LLC	DRRIP 或 SHiP	自适应、低开销
内存密集型负载	Hawkeye	学习最优算法
扫描型负载	SRRIP	扫描抵抗
Thrashing负载	BRRIP	Thrashing抵抗

5.3 未来发展趋势

缓存替换策略的研究正在向以下几个方向发展：

5.3.1 机器学习驱动的替换策略

越来越多的研究将机器学习技术应用于缓存替换：

- **LeCaR**：使用强化学习在LRU和LFU之间动态选择
- **Cacheus**：学习SR-LRU和CR-LFU之间的平衡
- **GL-Cache**：使用机器学习对对象组进行排序
- **3L-Cache**：低开销的精确学习策略

5.3.2 应用感知缓存

利用应用程序的语义信息来指导缓存决策：

- 预取提示（prefetch hints）
- 访问模式标注
- 优先级区分

5.3.3 异构存储层次

随着存储技术的发展，缓存需要适应新的存储层次：

- NVM（非易失性内存）缓存
- CXL内存扩展
- 计算存储（Computational Storage）

5.3.4 专用工作负载优化

针对特定应用领域的优化：

- 机器学习推理缓存（KV Cache管理）
- 图数据库缓存
- 时序数据库缓存

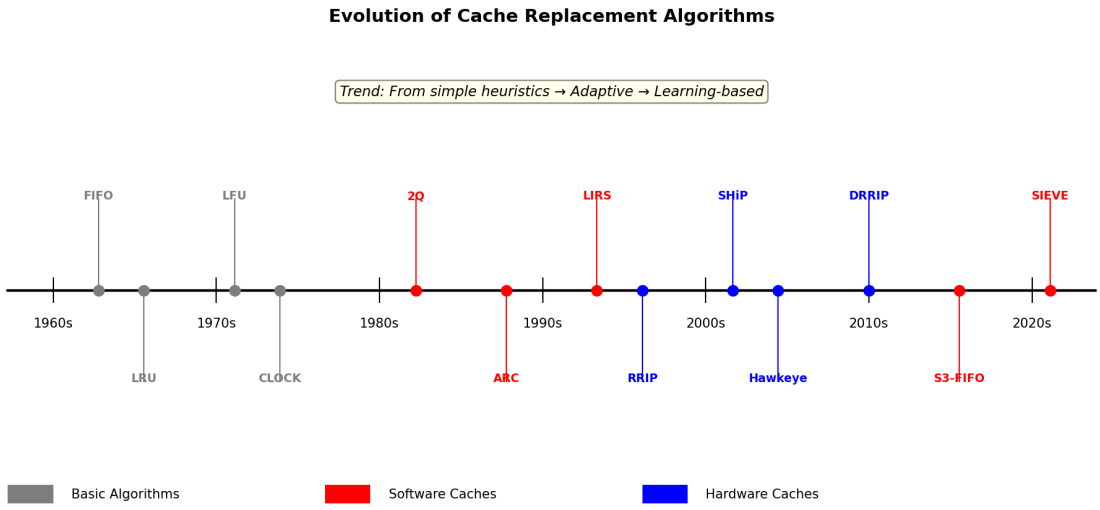

图 5-2 缓存替换算法演进时间线

5.3.5 总结

缓存替换策略从最初的简单启发式（FIFO、LRU、LFU）发展到复杂的自适应算法（ARC、DRRIP），再到如今的机器学习驱动方法（Hawkeye、LeCaR）。这一演进反映了计算机系统对工作负载复杂性的不断适应。

尽管新技术层出不穷，但一些基本原则始终不变：

- **简单性**：简单的算法更容易实现、调试和验证
- **自适应性**：好的算法应该能适应工作负载的变化
- **低开销**：缓存管理本身不应该成为性能瓶颈
- **扫描抵抗**：防止一次性访问污染缓存

未来的缓存替换策略将继续在简单性和复杂性之间寻找平衡，在硬件约束和性能目标之间做出权衡。无论是软件缓存还是硬件缓存，理解这些基本原理都将帮助系统设计者做出更好的选择。

参考文献

1. Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 78-101.

2. Effelsberg, W., & Haerder, T. (1984). Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4), 560-595.

3. Corbato, F. J. (1969). A paging experiment with the Multics system. *MIT Project MAC Report*.

4. Johnson, T., & Shasha, D. (1994). 2Q: A low overhead high performance buffer management replacement algorithm. *Proceedings of the 20th VLDB Conference*, 439-450.

5. Megiddo, N., & Modha, D. S. (2003). ARC: A self-tuning, low overhead replacement cache. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 115-130.

6. Jiang, S., & Zhang, X. (2002). LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1), 31-42.

7. Jaleel, A., Theobald, K. B., Steely Jr, S. C., & Emer, J. (2010). High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 38(3), 60-71.

8. Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely Jr, S. C., & Emer, J. (2007). Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2), 381-391.

9. Wu, C. J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely Jr, S. C., & Emer, J. (2011). SHiP: Signature-based hit predictor for high performance caching. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 430-441.

10. Jain, A., & Lin, C. (2016). Linearizing irregular memory accesses for improved correlated prefetching. *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 247-259.

11. Yang, J., Yue, Y., & Rashmi, K. V. (2023). FIFO queues are all you need for cache eviction. *Proceedings of the 29th Symposium on Operating Systems Principles*, 130-149.

12. Zhang, Y., Yang, J., Yue, Y., Vigfusson, Y., & Rashmi, K. V. (2024). SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. *21st USENIX Symposium on Networked Systems Design and Implementation*, 1229-1246.

13. Zhao, Y. (2022). Multi-Generational LRU: The Background. *Linux Kernel Mailing List*.

14. Vietri, G., Lora, M., Salkhordeh, M., Ichkov, V., & Tozun, P. (2018). Driving cache replacement with ML-based LeCaR. *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.

15. Rodriguez, A., & Kandemir, M. (2021). Cacheus: A cache replacement policy framework using online learning. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

16. Einziger, G., Friedman, R., & Manes, B. (2017). TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4), 1-31.

17. Beckmann, N., & Sanchez, D. (2018). LHD: Improving cache hit rate by maximizing hit density. *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, 389-403.

18. Shi, X., Li, Q., Jiang, X., & Zhang, C. (2019). Glider: A GPU-based learned index for high-dimensional data. *Proceedings of the 2019 International Conference on Management of Data*.

19. Song, J., Zhang, Y., Qian, Z., & Chen, M. (2020). LRB: Learning-based resource allocation and buffer management. *Proceedings of the 2020 ACM SIGCOMM Conference*.

20. Blankstein, A., Sen, S., & Freedman, M. J. (2017). Hyperbolic caching: Flexible caching for web applications. *Proceedings of the 2017 USENIX Annual Technical Conference*, 499-511.