计算机体系结构期末作业：cache 替换策略和数据预取

赵健坤　2010535
南开大学计算机学院, 天津 300350

摘要

本文探究不同 cache 替换策略和数据预取策略及其组合对 CPU 性能的影响。作者首先在 L1D 和 L2C cache 上实现了 GHB、SPP、Bouquet 等多种 cache 预取算法，在 LLC cache 上实现了 LFU 和改进的 SHiP 替换算法。为使预取器和替换策略在现实程序中的多种不同内存访问模式下都能取得较好效果，作者对这些策略进行了分别测试和性能比较，并探讨了其最佳组合策略。使用 ChampSim 模拟器在 SPEC CPU2006 trace 上进行的仿真实验表明，在 L1D 上采用 Bouquet 预取算法、在 L2C 上采用基于 IP-Stride 的 GHB 预取算法、在 LLC 上采用基于 SRRIP 的 SHiP++ 替换策略可获得最优性能。实验测得该组合策略平均 IPC 为 1.088，较无预取的 LRU 策略性能提升 105.9%。同时，该策略的总硬件开销仅为约 43.5kB。本工作源代码详见https://github.com/Ching-Yee-Chan/cache-prefetcher-and-replacement。

> 苟达变而识次，犹开流以纳泉；
> 如失机而后会，恒操末以续颠。
>
> ——《文赋》

1. 引言

从本质上讲，cache 替换策略和数据预取是提升 cache 命中率的两种不同手段：数据预取策略探讨如何将更有用的 cache line “引进来”，而 cache 替换策略探讨如何防止更有用的 cache line “走出去”。实现这一目标的关键，在于如何在有限的硬件开销下，利用历史 cache 访问记录，预测未来 cache 访问流。但突破这一瓶颈非常困难。程序的多样性和现代编程语言的灵活性意味着 cache 访问流的模式越来越难以学习。较准确的 cache 流预测需要更多的历史记录，但更多的历史记录意味着更大的硬件开销。另外，预取策略通常还需要权衡策略 (strategy) 和准确度 (accuracy)：过小的预取深度会导致遗漏潜在未来 cache line 而导致命中率降低，而过大的预取深度会导致 cache 污染或颠簸。更重要的是，不同访问流模式的最佳预取深度往往是不同的，不同类型 cache 访问的频率和模式亦不相同。理想的 cache 替换策略和数据预取策略必须 “达变而识次”，即不仅能准确识别和区分 cache 访问流中的各种模式，还具备处理复杂多变甚至无规律内存访问的鲁棒性。一旦 “失机而后会”，就会造成 cache 污染或颠簸，使得本末倒置，性能大幅降低。

因此，很多现代 cache 替换策略和数据预取策略采用分类思想：即根据不同类别的 cache 访问或历史访问流，采用不同的替换和预取策略。作者经实验发现，单一类别策略往往在一些 trace 上表现较好，而在另一些 trace 上表现较差。产生这种现象的可能原因是不同 trace 中程序的内存访问模式有很大差别，只有契合该模式的替换和预取策略能提升该 trace 的运行性能；而相同策略在其他拟合较差的 trace 上性能提升不明显，甚至可能因为不当的替换或预取导致 cache 污染或颠簸而降低总体性能。

本文实现了一种根据 cache 访问类型和历史访问记录动态调整的组合 cache 替换和预取策略。cache 预取方面，作者实现了 L2C cache 上的 GHB 算法 [6]作为性能评价基准。这种算法基于 IP-Stride 思想，将 IP 与对应的 cache line 访问历史记录解

耦，使用一个 FIFO 队列（即 GHB）实现历史记录的自然迭代，而相同 IP 的访问记录则使用链表连接。更进一步地，作者在 L1D cache 上实现了四种基于 IP 的预取算法，分别处理 GS (Global Stream)、CS (Constant Stride)、CPLX (Complex Stride) 和 TIP (Temporal and Irregular IP) 四类访问模式。这四种策略组合形成了 Bouquet 预取算法（亦称 IPCP，即 Instruction Pointer Classifier-based Hardware Prefetching）[7]。该算法利用饱和计数器生成置信度，从而将不同 cache 访问流模式分为上述四类，并选择最优预取策略。作者选择在 L1D cache 上实现预取算法，是因为现代超标量 CPU 对较底层的 cache miss 的鲁棒性更高 [6]。而由于已经在 L1D cache 上实现了 Bouquet 预取策略，在 L2C 上就无需再实现复杂预取策略，否则容易造成 cache 污染。因此，作者将 L1D cache 上的 Bouquet 算法和 L2C cache 上的 GHB 算法结合，作为最后的组合 cache 预取策略。

cache 替换算法方面，作者首先在 LLC 上实现了经典的 LFU 算法作为性能基准。该算法逐出历史访问频率最低的 cache line。更进一步地，作者实现了 ShiP++ 算法 [9]，该算法从高置信度 cache 插入、SHCT (Signature History Counter Table) 更新策略、针对写回的 RRPV (Re-reference prediction values) 更新策略、针对预取的 SHCT 更新策略、针对预取的 RRPV 更新策略五个方面对 SHiP 算法进行了优化。

作者对不同预取和替换策略进行了对比实验，以探究不同策略对 IPC 和各级 cache 命中率等的影响。特别地，作者对 Bouquet 预取算法进行了消融实验，以验证组合策略的各个部分对整体性能的提升效果。最终的 Bouquet-GHB-SHiP++ 组合策略平均 IPC 为 1.088，较无预取的 LRU 策略性能提升一倍以上。

2. 相关工作

cache **预取**. 现代数据 cache 预取器一般分为四种 [1]：第一种是基于步长和流的预取器 (Stride and Stream Prefetchers)。这种策略可以看作固定步长和前向距离 (look-ahead) 的 Next-Line 预取策略，如示例代码中的 IP-stride 算法。第二种是地址相关预取器 (Address-Correlating Prefetchers)。这种策略旨在优化预取器在基于指针的数据结构（如链表）上的表现，但需要较大空间记录访问地址序列。典型的地址相关预取器有 Markov 预取器等。第三种是空间相关预取器 (Spatially Correlated Prefetching)。它基于 delta 概念，利用类和结构体的访问规律进行预取，例如 GHB PC/DC[6]、SPP[4]等。第四种是基于执行的预取器 (Execution-Based Prefetching)。它利用 CPU 中的阻塞周期和空闲的硬件资源，在指令执行之前提前把数据在 Cache 中准备好。

Bouquet 预取器 [7]是一种基于 IP 索引的组合策略预取器。它将访问模式分为全局流式访问 (GS)、固定步长访问 (CS)、复杂步长访问 (CPLX)、临时和无规律访问 (TIP) 四类，并分别采用 GHB 算法 [6]、IP-Stride 算法、SPP 算法 SPP[4]、Next-Line 算法进行预取。该算法分别建立 GHB、IP table、DPT (Delta Prediction Table) 以记录前三种算法的历史信息，并在其表项中使用饱和计数器生成每种类别的置信度，依据置信度对每次访问进行分类。

cache **替换**. 理论业已证明 FF 算法为最优 cache 替换算法 [5]。LRU 算法虽为 FF 的一种近似，但存在一个严重缺陷：LRU 假定刚刚插入的 cache line 最有可能在将来重新被访问，但某些 cache 访问（如写回）在相当长时间内极有可能只访问该地址一次。为支持更多样的内存访问模式和更新颖的预取策略，cache 替换策略正朝更细粒度、更长历史、更多访问模式的方向发展 [2]。从 LRU 及其简化版本 NRU 算法出发，Aamer Jaleel 等于 2010 年提出了 SRRIP、DRRIP 和 BRRIP，其中引入了 RRPV (Re-reference Prediction Values) 以估计当前 cache line 被换出的可能性，它事实上是扩展为多位的 NRU bit[3]。在此基础上，Carole-Jean Wu 等于 2011 年提出了 SHiP，其中利用 IP 签名索引的 SHCT (Signature History Counter Table) 实现细粒度的 RRPV 初始化 [8]。为实现更细粒度的 SHCT 和 RRPV 更新及初始化，Vinson Young 等从访问分类出发，提

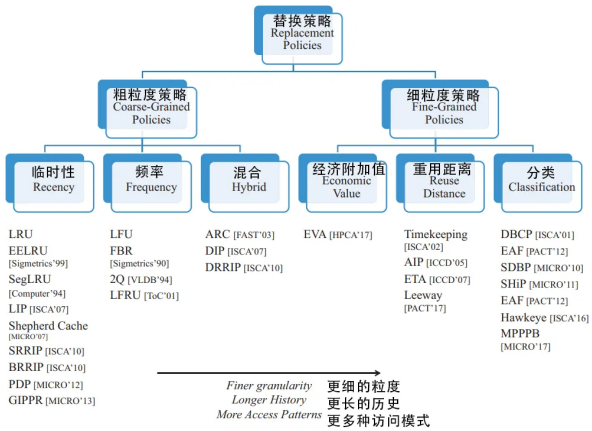


图 1. cache 替换策略的一种分类法（引自 [2]）

出了 SHiP 策略的五点改进方案，即 SHiP++ 策略 [9]。

3. cache 预取实现思路

除基于 IP-Stride 的 GHB 算法外，接下来介绍的面向 Bouquet 算法的 GHB 算法、IP-Stride 算法、SPP 算法和 Next-Line 算法均已集成在 IPCP.l1d_pref 和 IPCP.l2c_pref 文件中。第5章中的对比实验和消融实验是通过修改文件首部的宏定义，使能 Bouquet 中的选定模块完成的。

GHB 算法. GHB 算法的核心是实现索引与历史记录的解耦，利用 FIFO 队列逐出过时的“僵尸”表项，从而提高预取准确度。本实验首先实现基于 IP-Stride 的 GHB 算法。该算法的核心数据结构是 IT 和 GHB。IT 表根据 IP 哈希值（即 IP 后 8 位）进行索引，其中存储该 IP 最近一次访问的 cache-line 地址在 GHB 中对应的表项。同一 IP 访问的 cache 地址序列以链表形式存储在 FIFO 顺序队列 GHB 中，每个 GHB 表项由包含地址 cl_addr 和上一项指针 prev_ptr 两个域的结构体 ghb_entry 构成。本程序中所有空指针均以-1 表示。预取器初始化时，IT 表项和 GHB 表项中的指针域将被全部初始化为-1。在触发 L2 查找时，首先将地址插入 GHB，在这之前若队列已满，需要将队头表项出队。作者使用循环队列实现 GHB，并用全局变量 tail 标记队尾后的第一个空项（同时也可能是队头）。在每次出队时，需要遍历 IT 表以删除全部指向该项的索引并将 GHB 表中指向该项的指针置空，并对 tail 做取模自增。然后更新 IT 表项，并根据 IT 表中的索引获得上 3 次访问地址（若不足 3 条记录直接退出）。计算并比较两个 stride（此时需要注意无符号减法），若相同则启动预取。

Bouquet 算法中也使用了 GHB 算法以处理 GS 类访问模式，但其实现与上述方法有很大不同。Bouquet 中的 GHB 算法旨在捕捉 IP 无关的全局流式数据访问，即 cache line 对齐的连续地址访问。因此，该算法不需要索引表，只需要一个大小为 12 的 FIFO GHB，以及 trackers 表的三个域，即 str_valid、str_direction 和 str_length。在更新阶段，算法分别统计当前 GHB 中位于 [cl_addr-GHB_SIZE, cl_addr) 和 (cl_addr, cl_addr-GHB_SIZE] 间的表项数，若位于前一区间的表项数更多则判定当前 IP 处递增序列，令 str_direction=1，反之判定其属递减序列，令 str_direction=0。若该选定方向记录数量超过 GHB 容量的一半，则认为该 IP 属于临时 GS，并令 str_valid=1, str_strength=0；若未超过一半且不是永久 GS 则取消流认定并重置 str_valid=0；特别地，若超过 GHB 容量的 75%，则认为该 IP 属于永久 GS，并令 str_strength=1。然后添加非重复 GHB 项。在预取阶段，若 IP 被判定为 GS 类，则按 str_direction 指示方向，步长为 1 连续预取 6 项。

IP-Stride 算法. Bouquet 算法和 GHB 算法中 IP_Stride 算法的原理非常相似，下面仅阐述实现中的差别。由于引入了置信度，Bouquet 中的 IP_Stride 能够识别“僵尸”表项，故仍采用基于单表而非 GHB 的实现以降低硬件开销并利用更长历史。trackers 表中设置了 IP_tag、IP_valid、Page_no、Page_offset、Last_stride、CS_conf 六个域供 IP-Stride 算法使用，其中 CS_conf 为置信度，Page_no 和 Page_offset 供页边距学习使用。置信度计算方法为：若步长相同，则置信度自增（最高不超过 3），否则自减（最低不小于 0）。仅当置信度为 0 时，更新步长。这种 2 位饱和计数器能够使算法对分支造成的单次步长抖动鲁棒。页边距学习能够使置信度

估计更准确。当历史记录中 Page_no 与当前地址的 Page_no 不相同时，认为跨页。若此时根据 offset 计算出的 stride 为负，则加 64（一页中的 cache line 数量），否则减 64。当 $CS\_conf > 1$，且 stride 不为 0 时，触发 3 次等步长预取。

SPP **算法**. SPP (Signature Path Prefetcher) 算法是一种利用步长匹配进行游走的预取策略，在 Bouquet 中用于处理 CPLX 类访问模式。SPP 是基于 delta 索引的空间相关预取器，其维护记录预测步长和置信度的 DPT 表。DPT 表共 4096 项，由 trackers 表项中的 12 位签名索引。更新阶段首先根据 DPT 步长预测结果更新置信度和 stride，该步步骤与 IP-Stride 中完全一致，此处不再赘述。然后按下式更新 trackers 表中的 sig:

$$\text{sig} = (\text{last\_sig} \ll 1) \hat{} \ \text{stride}$$

SPP 的预取策略非常精妙：SPP 将从当前 trackers 表项中存储的签名开始，按照签名对应的 DPT 表项中存储的 stride 计算新的预取地址。如果该表项置信度大于 1，则按该地址进行预取。然后根据当前 stride 更新签名。重复以上步骤直至遇到置信度为 0 或 stride 为 0 的 DPT 表项，跨页，或重复次数超过预取深度 3。

Next-Line **算法**. 在 Bouquet 算法中，若当前 IP 在 trackers 表中没有对应的表项（即 tag 不匹配），或不属于 GS、CS、CPLX 中的任何一类，则会触发有条件的 next-line 预取。由于此时剩余的 IP 大多是临时或无规律访存，无条件的 next-line 很可能造成 cache 污染。事实上，只有在 cache 占用率不高的情况下启用 next-line 才能提升性能。因此，作者在 l1d_prefetcher_operate 函数前部加入了计算 MPKC (miss per kilo cycles，每千周期缺失数) 的模块。MPKC 每 256 次缺失更新一次，若 MPKC 大于 15，则关闭 next-line 以避免污染。

Bouquet **算法**. Bouquet 算法是上述四种算法的组合。该算法根据置信度将访问模式分为 GS、CS、CPLX 和 TIP 四类，并分别使用 GHB、IP-Stride、SPP、Next-Line 进行预取。Bouquet 基于统一的，IP 哈希索引的表 trackers 存储这四类访问的历史信息（如图2）。Bouquet 的核心在于使用置信度进行分类：每次 cache 访问都将首先更新当前 IP 对应的 GS、CS 和 CPLX 的置信度，然后按照 GS、CS、CPLX、TIP 的优先级，逐一检查该 IP 分为各类的置信度，选取当前 IP 对应的一类，并执行该类对应的预取算法。

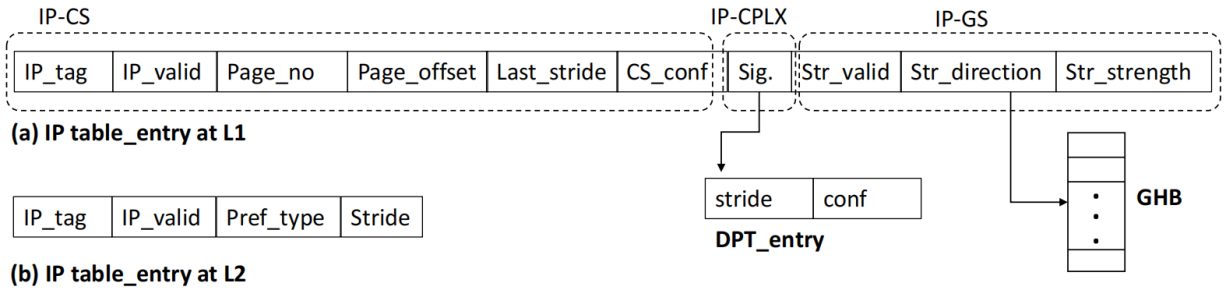
在 L2C 上实现的 Bouquet 算法较 L1D 有很大不同：在 L1 向 L2 传递的元数据中，作者自高位到低位依次编码了 SPEC_NL (1 位)、Class_Type (3 位) 和 Stride (8 位) 三个值。因此，L2C 上的 Bouquet 算法中直接记录 IP 对应的 Pref_type，而不再使用置信度分类。此外，为防止对 L1 预取的二次预取造成 L2 cache 污染，L2 cache 不对 CPLX 类访问进行预取，且预取深度更小（一般为 4）。这样，L2 上的 Bouquet 就退化为基于步长和流的预取器，其 trackers 表也仅用于储存其类别和步长信息，如图2。

4. cache **替换实现思路**

LFU **算法**. 为便于实现，作者将访问频率保存在 BLOCK 类的 lru 域中。除写回命中外，LFU 算法在每次 cache 访问时均会将频率加 1。在换入时，LFU 首先填充空闲 cache line。若没有空闲 cache line，则遍历每组的所有 cache line 并选择访问频率最低的一个换出。

SHiP **算法**. 受到 RRIP[3]的启发，ShiP 算法用两位计数器 RRPV 代替了 LRU 算法中记录访问次序的 lru 字段。RRPV 越大，代表该 cache line 的访问间隔越大，越优先被逐出。SHiP RRPV 的状态机如图3。在选择换出的 cache line 时，SHiP 选择第一个 RRPV=3 的 cache line 换出。若没有 cache line 的 RRPV=3，则所有 cache line 的 RRPV 自增 1。重复以上操作，直到找到 RRPV=3 的 cache line。

SHiP 算法的核心是 SHCT 表。该表记录了由 IP 哈希值索引的三位饱和计数器。其初始值为 1，命中时自增，缺失时自减，用以反映新插入 IP 所访



Size of an IP Table entry at L1: 77 bits
IP_tag (6), IP_valid (1), Page_no (52), Page_offset (6), Last_stride (7), CS_conf (2), str_valid, direction, and strength (1 bit each)
Size of an IP Table entry at L2: 17 bits
IP_tag (6), IP_valid (1), Last_stride (7), Pref_type(3)
An L1 GHB entry: 58 (64-6) bits for cache block aligned address An L1 DPT entry: 9 bits (stride (7) + conf (2))

图 2. Bouquet 算法表项示意图（引自 [7]）

问地址未来被重复访问 (Re-reference) 的概率。与 SRRIP 将所有初次插入的 cache line 的 RRPV 设置为 2 (far) 不同，SHiP 将 SHCT 表项值围为 0 的新插入 cache line 的 RRPV 初始化为 3 (distant)。这样，一些访问间隔较大的 cache 在换入后很快就能换出，从而有效避免了颠簸的发生。

受到 DRRIP 的启发，SHiP 仅采样了一些 cache line 录入 SHCT，这样能有效防止 hash 碰撞，并通过引入随机性提高性能。这也是事例程序在 update_sampler 函数中完成 SHCT 表项更新的原因。

SHiP 算法并不是本工作的重点，以上对 SHiP 的介绍只是为了让读者便于理解接下来的 SHiP++ 算法。SHiP 代码框架来源于 ChampSim 提供的事例，作者对朴素 SHiP 代码的原创性没有贡献。

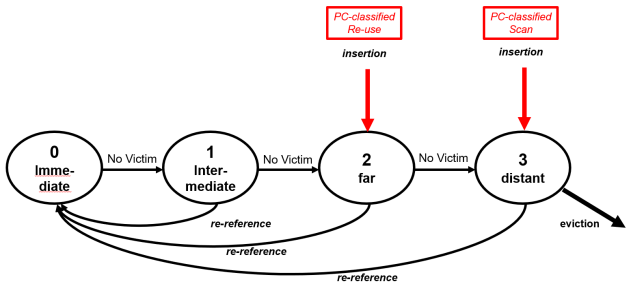


图 3. SHiP 算法状态机（引自 [9]）

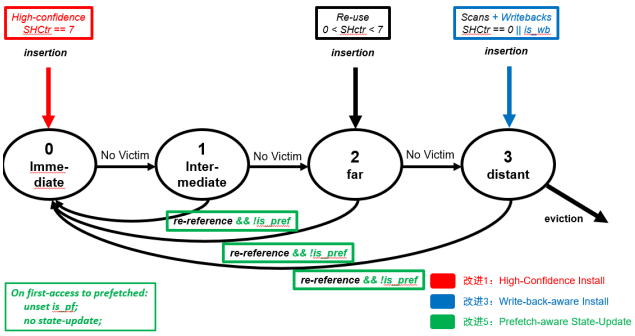


图 4. SHiP++ 算法状态机（引自 [9]）

SHiP++ 算法. 如图4所示，Ship++ 算法对 Ship 进行了 5 个方面的改进，其中后 2 项特别针对有预取情况。第 1、3、5 项旨在实现 RRPV 更细粒度的初始化和更新，第 2、4 项旨在实现 SHCT 更细粒度的更新 (即 training)。以下将分别介绍。

1. High-Confidence Install

   SHiP++ 认为，SHCT 表项为最大值 7 时，可以认为新插入的 cache line 极可能将被重新访问，因此可以将其 RRPV 直接初始化为 0。当然，如果这是一条预取访问，则其重复命中可能是预取策略而非程序本身所致，故初始化 RRPV 为 1。

2. Balanced SHCT Training

   朴素 SHiP 中，SHCT 表项在每次 cache 命中时均自增，在每次 cache 缺失时均自减。然而这样可能导致计数不均。SHiP++ 中，将其修改为仅首次命中时计数器增加，而实现方式为：在

SAMPLER_class 中加入 used 字段，跟踪换入的 cache line 是否被重复访问。

另外，与 SHiP 原论文相反，ChampSim 提供的 SHiP 实现中，SHCT 表项更新规则为命中自减，缺失自增。作者按照原论文对其进行了修改。

3. Write-back-aware Install

   写回 cache 访问的重复访问间隔一般很长。因此换入时，若访问类型为写回，无论其是否命中，都将其 RRPV 置为最大。

4. Prefetch-aware Training

   预取类型的 cache 访问模式一般取决于预取策略而非程序本身。因此，SHiP++ 将 IP 签名最低位设置为 is_prefetch。这样，同一 IP 发出的正常访问和预取将被分别记录在 SHCT 表中，并获得不同的 RRPV 初始值。

5. Prefetch-aware State-Update

   预取策略可能在短时间内对同一 cache line 发出多次预取请求，致使替换策略误认为该 cache line 很可能被程序重复访问，但事实上程序可能仅访问一次，甚至根本不会访问该 cache line。因此，SHiP++ 为每个 cache line 额外维护一个标志位 is_prefetched，以判断该块是否为预取进的块。若该块被非预取指令命中，则认定为流式访问，RRPV 置为最大值，并将 is_prefetched 复位；若被预取指令命中，则保持 RRPV 不变。

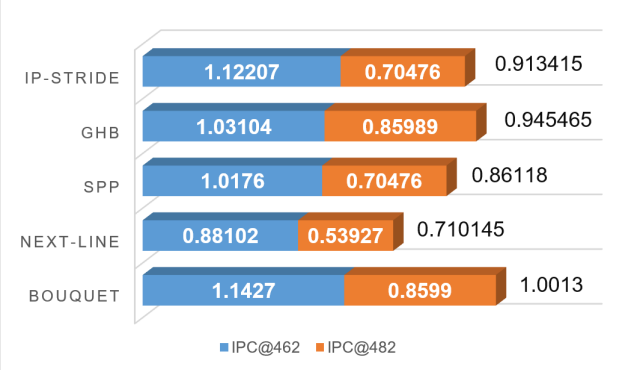


图 5. bouquet 算法中各部分在不同 trace 上的 IPC（白）及平均 IPC（黑）

5. 实验结果

作者使用 ChampSim 模拟器在 SPEC CPU2006 中的 462.libquantum（以下简称 462）、482.sphinx3（以下简称 482）两个 trace 上进行了性能测试。

不同预取器的比较. 使用 LRU 替换策略，仅在 L1D 上部署不同预取器，于两个 trace 中分别测得的 IPC、L1D 命中率、有效预取率及预取延迟如表1。由图5可见，IP-Stride 在前一 trace 上性能表现较好，GHB 在后一 trace 上性能表现较好，而 Bouquet 通过分类及策略组合使得其在两个 trace 上的性能均较好。值得注意的是，表1中 IP-Stride 与 SPP 的各项址表均比较接近，这是因为 SPP 与 IP-Stride 一样，也能学习到固定步长的访问模式。但 IP-Stride 在拟合固定步长访问模式时历史保留时间更长，鲁棒性更高，而 SPP 主要面向复杂访问模式，因此当 trace 中固定步长访问模式占较大比例时，就会产生类似 462 号 trace 中 IP-Stride 性能高于 SPP 的情况。

不同预取器的组合. 作者使用 LRU 替换策略，在 L1D 和 L2C 上尝试了 bouquet 和 GHB 的三种组合，其性能比较如表2。由表可见，bouquet+GHB 混合策略的总体性能较好，这种性能增益主要来源于 482 号 trace 上 L2 命中率的提升。这可能是因为 482 号 trace 中有较多流式访问模式，而相较于组合策略 bouquet，在 L2 cache 上部署 GHB 能以更激进的预取提升流式访问在 L2 cache 上的性能。

不同替换算法的比较. 作者在 L1D 和 L2C 上分别部署 bouquet 和 GHB 预取器，测得不同 cache 替换算法的 LLC cache 命中率及 IPC 如表3。基于访问频率的策略使得 LFU 和 SHiP 较 LRU 取得显著性能提升，而分类思想使得 SHiP++ 在 SHiP 的基准上再次取得显著性能提升。

硬件开销. 为实现 cache 访问流的预测，大多数 cache 预取和替换算法都需要一个相当大的表存储历史访问信息。bouquet 中开销较大的数据结构有

	462				482			
	IPC	命中率	有效预取率	预取延迟	IPC	命中率	有效预取率	预取延迟
IP-Stride	1.12207	0.90614	0.1562012	60562	0.70476	0.83849771	0.180911975	1646581
GHB	1.03104	0.893071	0.08392435	15884	0.85989	0.87481669	0.147335255	479685
SPP	1.0176	0.916601	0.18156121	46802	0.70476	0.83849767	0.180808476	1654783
Next-Line	0.88102	0.931875	0.04205356	411657	0.53927	0.87882465	0.057786003	2076262
bouquet	1.1427	0.910543	0.15933389	67304	0.8599	0.87481668	0.147350964	479923

表 1. bouquet 算法中各部分在不同 trace 上的性能比较

	462			482			avg
	IPC	L1 命中率	L2 命中率	IPC	L1 命中率	L2 命中率	IPC
GHB+GHB	0.87374	0.942066	0.67629	0.85916	0.8807	0.740365	0.86645
bouquet+bouquet	1.18467	0.915127	0.700763	0.87743	0.882686	0.308139	1.03105
bouquet+GHB	1.18076	0.914809	0.68682	0.92987	0.899733	0.6302	1.055315

表 2. 不同 L1D 与 L2C 预取策略组合分析

	LLC hit@462	LLC hit@482	IPC
LRU	0.142554	0.296197	1.055315
LFU	0.319163	0.061458	1.07086
SHiP	0.21852	0.041987	1.07486
SHiP++	0.19823	0.059661	1.08808

表 3. 不同 cache 替换算法性能比较

L1D pref.	L2C pref.	LLC repl.	avg. IPC
no	no	LRU	0.528585
no	GHB	LRU	0.874755
bouquet	no	LRU	1.0013
bouquet	GHB	LRU	1.055315
bouquet	GHB	LFU	1.07086
bouquet	GHB	SHiP	1.07486
bouquet	GHB	SHiP++	1.08808

表 4. cache 预取策略与替换策略的不同组合及整体性能

trackers 表、DPT 表和 GHB（只有 12 项，相对较小）；GHB 中有 GHB 表和 IT 表；SHiP++ 中有 SHCT 表、sampler 表及 rand_sets 表、RRPV 及 is_prefetched 标志位。各个模块的硬件开销如表5。相较于 ChampSim 中 32.8kB 的 L1D cache 和 524kB 的 L2C cache，组合策略的开销确实是不容忽视的。但考虑到其带来的超过一倍的性能增益，这种折中又是值得的。

模块	存储开销/kB
bouquet	11.7
GHB	11.8
SHiP++	20
合计	43.5

表 5. bouquet-GHB-SHiP++ 策略硬件存储开销 [7, 9]

在经过不同预取器的比较、组合，以及不同替换算法的比较后，作者在表4中展示了预取器与替换算法不同组合的整体性能。实验表明，bouquet-GHB-SHiP++ 组合为最佳策略，其性能较无预取 +LRU 替换提升 105.9%。

6. 总结

本文通过在 ChampSim 模拟器和 SPEC CPU2006 trace 上进行的仿真实验，探究了不同 cache 替换策略和数据预取策略及其组合对 CPU 性

能的影响。作者首先实现了 GHB 预取算法和 LFU 替换算法作为性能基准。在 Bouquet 预取算法中，作者将访问模式分为 GS、CS、CPLX 和 TIP 四类，并根据其特点分别采用 GHB、IP-Stride、SPP、Next-Line 预取器完成预取；在 SHiP++ 替换算法中，为实现更细粒度的 RRPV 及 SHCT 管理，作者在 ChampSim 提供的 SHiP 框架下进行了 5 项改进。本工作的核心在于通过分类提高对未来 cache 行为预测的准确性，以期“达变而识次”的效果。实验结果表明，bouquet-GHB-SHiP++ 组合为最佳策略，其平均 IPC 可达 1.08 以上。

参考文献

[1] B. Falsafi and T. F. Wenisch. A primer on hardware prefetching. Synthesis Lectures on Computer Architecture, 9(1):1–67, 2014. 2

[2] A. Jain and C. Lin. Cache replacement policies. Synthesis Lectures on Computer Architecture, 14(1):1–87, 2019. 2, 3

[3] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). ACM SIGARCH computer architecture news, 38(3):60–71, 2010. 2, 4

[4] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12. IEEE, 2016. 2

[5] J. Kleinberg and E. Tardos. Algorithm design, 2005. 2

[6] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In 10th International Symposium on High Performance Computer Architecture (HPCA'04), pages 96–96. IEEE, 2004. 1, 2

[7] S. Pakalapati and B. Panda. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 118–131. IEEE, 2020. 2, 5, 7

[8] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pages 430–441, 2011. 2

[9] V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi. Ship++: Enhancing signature-based hit predictor for improved cache performance. In The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017), 2017. 2, 3, 5, 7