体系结构期末报告

Cache 预取和替换

姓名：张弛　学号：2110729　专业：密码科学与技术

一、背景概述

研究动机

cache 替换和预取策略是影响缓存性能的两个重要因素。其中预取策略主要用于提前加载可能被访问的数据到缓存中，以减少等待时间和提高系统性能；替换策略则决定当缓存满时哪些数据应被替换，以优化缓存中数据的相关性和有效性，进一步提升访问效率。

由于业界当前的许多工作基本围绕分立的预取和替换算法，而在它们之间的关联性乃至不同层级间两者的联系方面，可以说仍没有实质性进展，故笔者在探究已有预取&替换算法组合性能优劣的基础上，进一步尝试了在 L2C 层级对特定预取&替换策略开发其联合优化算法。

相关工作

关于数据预取策略和缓存替换策略前人已有许多优秀的工作，此处由于篇幅限制，就不一一详细介绍了，仅对本次实验基础部分中复现的预取&替换算法列举如下：

$Prefetch$

- **IP-Stride** 预取算法: 由 Dahlgren 和 Stenstrom 提出，其核心思想是基于程序中指令地址的重复模式进行预取，即通过分析指令指针的步长模式来预测未来的数据访问。
- **GHB (Global History Buffer)** 预取算法: 由 Nesbit 和 Smith 开发，这种方法使用一个全局历史缓冲区来记录缓存访问的历史，用于识别复杂的访问模式并据此进行预取。
- **Markov** 预取算法: 由 Joseph 和 Grunwald 提出，基于马尔可夫链模型，通过分析最近的访问历史来预测下一个可能访问的数据块。
- **基于马尔科夫链的Pangloss** 预取算法: 由 Philippos 等人提出，它结合了马尔科夫预测模型和 Pangloss 虚拟缓存系统，在 Markov 预取的基础上对最佳路径的选取进行了概率优化。

$Replacement$

- **LRU (Least Recently Used)**: LRU 策略基于这样的假设：最长时间未被访问的数据在未来也最不可能被访问。因此，它优先淘汰那些最久未被使用的缓存项。
- **LFU (Least Frequently Used)**: LFU 算法侧重于识别和淘汰历史上访问频率最低的数据。它假设频繁访问的数据在未来也更可能被访问。
- **MRU (Most Recently Used)**: MRU 策略与 LRU 相反，它淘汰最近刚被访问过的数据项。这是基于一种假设，即最近访问过的数据短期内不太可能再次被访问。
- **SHIP++ (Signature-based Hit Predictor++)**: SHIP++ 由 Jinchun Kim 和 Michael D. Powell 提出。该算法使用历史访问模式的 `签名` 来预测缓存行的命中概率。它旨在识别那些未来可能不会再次被访问的缓存行，并将它们作为替换的候选对象。
- **RED (Reuse Detection)**: 由 Javier Díaz Maag 等人提出。RED 策略通过分析缓存块的访问模式来预测其重用概率，旨在识别那些未来可能不会被重用的缓存块，并优先替换它们。

二、实验过程

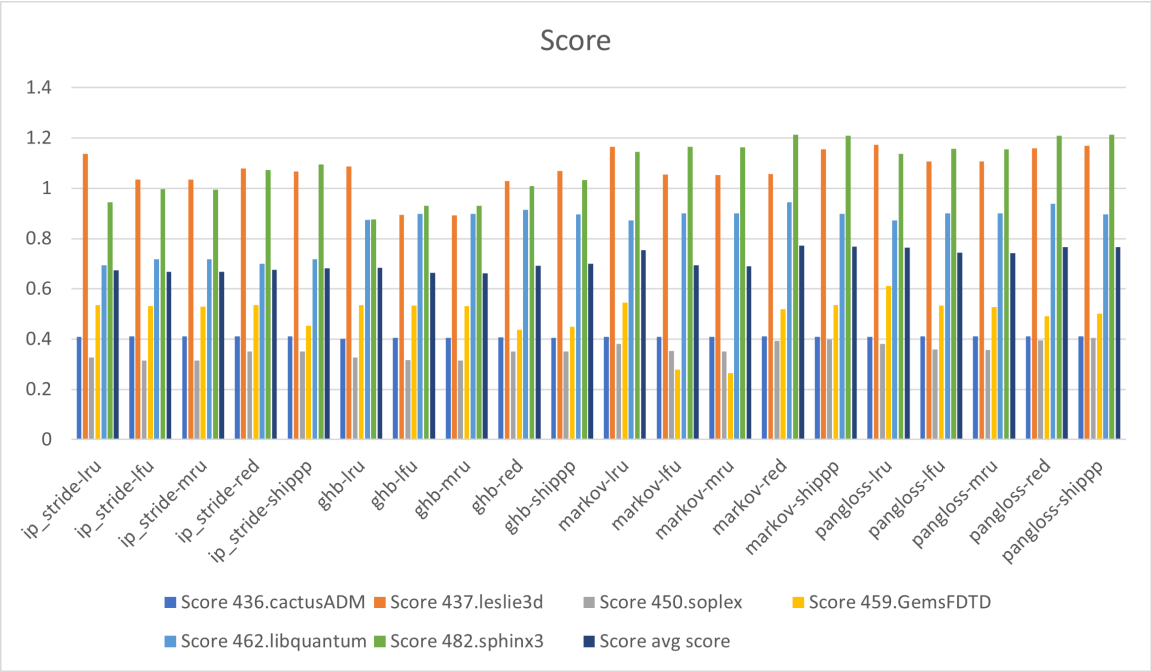
此处的实验过程是 ***研究方法与结果分析*** 两部分的糅合（在精简后），其中基础部分所复现的所有预取&替换算法的详细原理与实现过程都不再赘述，仅对进阶部分的联合优化算法作详细介绍，项目的所有 ***代码&数据&图表*** 见 Github 仓库 。

（一）基础部分

本实验基础部分中，笔者在复现上述 L2D 预取& LLC替换 策略的基础上，对不同的策略进行全排列组合并选定六个 trace（包括给定的 `462.libquantum` 和 `482.sphinx3`）进行了模拟测试，其中 L1D&LLC 均不进行预取，且 `warmup_instructions = 50M` & `simulation_instructions = 100M` 测得各算法组合的性能表现如下：

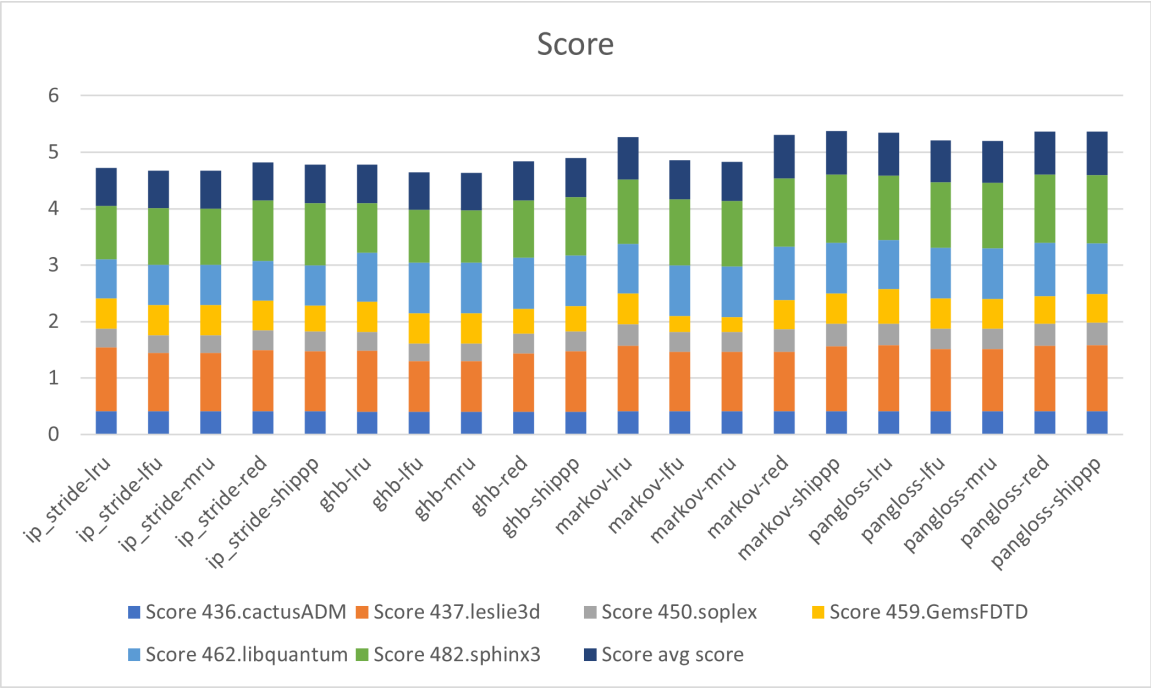
Benchmark	436.cactusADM	437.leslie3d	450.soplex	459.GemsFDTD	462.libquantum	482.sphinx3	avg score
ip_stride-lru	0.4097	1.1362	0.32733	0.53549	0.69346	0.94398	0.67436
ip_stride-lfu	0.41107	1.03565	0.3148	0.53121	0.71723	0.99736	0.667887
ip_stride-mru	0.41107	1.0344	0.31417	0.53009	0.71723	0.99534	0.66705
ip_stride-red	0.41186	1.07838	0.35045	0.53511	0.69883	1.07258	0.675927
ip_stride-shippp	0.41025	1.06733	0.35073	0.45272	0.7179	1.09436	0.682215
ghb-lru	0.40103	1.08604	0.32705	0.53566	0.87361	0.87656	0.683325
ghb-lfu	0.40429	0.89434	0.31599	0.53289	0.89804	0.93131	0.66281
ghb-mru	0.40429	0.8913	0.31539	0.53224	0.89804	0.92973	0.661832
ghb-red	0.40612	1.02876	0.34996	0.43725	0.91481	1.00921	0.691018
ghb-shippp	0.4044	1.06899	0.35113	0.44916	0.89615	1.03176	0.700023
markov-lru	0.40921	1.16559	0.38026	0.54549	0.87261	1.14552	0.753113
markov-lfu	0.40993	1.05488	0.35256	0.27847	0.90077	1.16482	0.693572
markov-mru	0.40993	1.05339	0.34989	0.26373	0.90077	1.16225	0.689993
markov-red	0.4106	1.05717	0.39294	0.51934	0.94378	1.21381	0.77193
markov-shippp	0.40973	1.15403	0.39958	0.53465	0.89849	1.20907	0.767592
pangloss-lru	0.40989	1.17348	0.38019	0.61145	0.87301	1.13785	0.764312
pangloss-lfu	0.4105	1.10739	0.35845	0.53262	0.90119	1.15775	0.74465
pangloss-mru	0.4105	1.10612	0.35611	0.52758	0.90119	1.15429	0.742632
pangloss-red	0.4109	1.15811	0.39464	0.49048	0.9379	1.20919	0.76687
pangloss-shippp	0.41003	1.16915	0.40534	0.50137	0.89703	1.21261	0.765922

将上述数据分布情况可视化展现如下：



就预取算法而言，`GHB` 步长预取相较于 `ip_stride` 来说有一定的优化效果，但作用并不明显。而 `Markov` 和 `Pangloss` 相较而言在性能上的提升较为显著，且 `Pangloss` 预取器在各数据集上的 score 分最高，平均达到了0.755左右。

接着对比缓存替换算法可以发现，这些替换算法的性能表现各有千秋，在不同 trace 上差异较大，但总体来看，`ReD` 和 `Ship++` 表现要优于相对传统的 `LFU & MRU`；而 `LRU` 仍老当益壮，性能表现依旧能打，排在前两位之后。

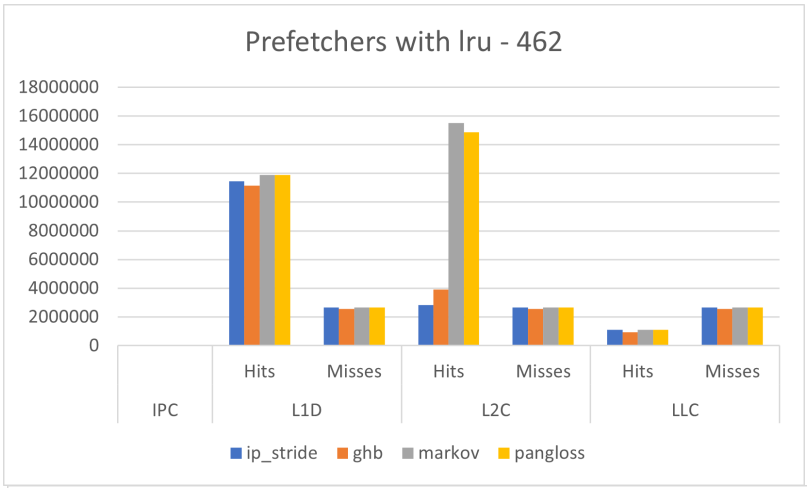


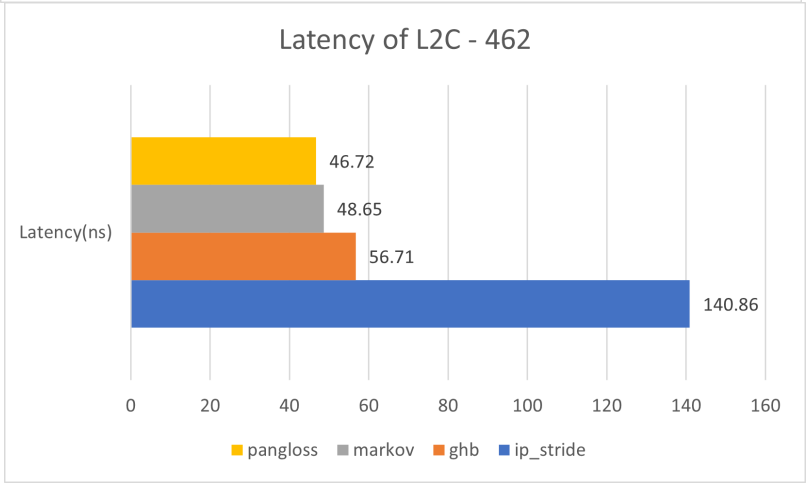
观察所有的算法组合中，不难发现其中的性能 top 是 `Markov+ReD`，其次是 `Pangloss+ReD` 。其中 `Pangloss` 和所有的缓存替换策略组合基本都有较好的表现，AvgScore 都能够稳定在 0.75 附近，这也说明了其本身的实力优越性。

对于数据集而言，这些组合在 `437.leslie3d-134B.champsimtrace.xz` 和 `482.sphinx3-1100B.champsimtrace.xz` 上都有较好的表现，而在 `450.soplex-247B.champsimtrace.xz` 和 `436.cactusADM-1804B.champsimtrace.xz` 上表现都相对一般，这明显与数据集本身的 ***访问模式复杂程度与可预测性*** 有关：不同数据集的访问模式（例如顺序、随机或局部性）会显著影响预取和替换算法组合的有效性。例如，具有强局部性的数据集可能更适合这些算法，因为一旦数据被预取，它们很可能很快被再次访问；同时如果数据集访问模式高度不可预测，预取算法可能难以准确预测下一次的数据访问，导致性能下降。

L2C 预取策略对比

当然，为了进一步探究算法之间性能差异的原因，笔者尝试从更具体的数据出发进行分析。此处笔者对 L2C 实现的三种预取算法进行性能分析，分别选取了在各级缓存上的命中率以及在 L1D 上的 **Latency** 数据。其中，L1D和 LLC 上不进行数据预取，缓存替换策略选用 LRU，测得结果如下所示。



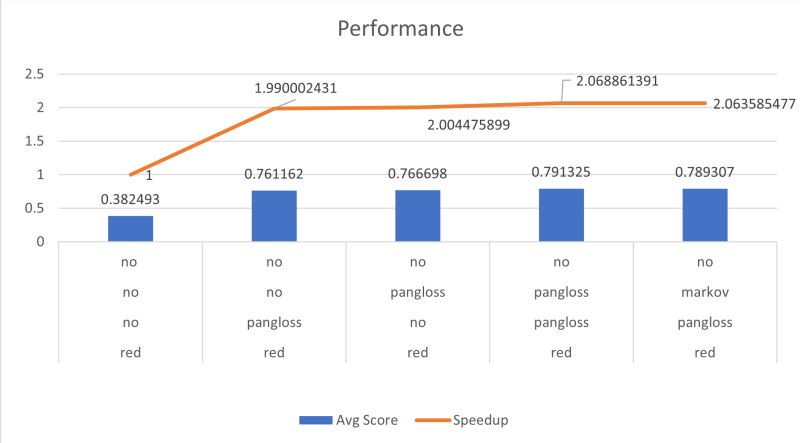


从图中数据可以看出，随着我们算法的提升，IPC 在逐步提高。GHB 策略某种程度上是对 ip_stride 的补充优化，能够根据历史信息来选择进行预取，由于补充了历史信息，因此这种预测相对于 ip_stride 会更加准确。pangloss（markov） 则是对 GHB 的大幅度优化改进，除了借鉴了 GHB 中有关历史信息的缓存，基于历史访问序列来预测，还引入了马尔科夫链模型来估计下一次访问的可能性。此外采用了访问序列间隔作为索引信息能够更好用有限的空间资源保存更多的历史信息。

因此，随着算法理论上的逐级优化，L2C 的 cache 命中率逐渐提升，且 L1D 由于 miss 产生的 latency 逐渐降低。这样的收益就是由 L2C 上合理的预取策略带来的。由于在 L2C 中进行了数据预取，因此能够在数据还没有被访问的时候提前取到 L2C 中，使得在时候的访问时原本会产生 miss 的数据能够在 cache 中命中，此外当 L1D 发生 miss 的时候，需要的 L2C 中查询，通过预取能够大大提升这类查询的命中率。从数据中也能够看出，性能提升主要是由 L1D 产生 miss 时的 latency 降低带来的。

L1D&L2C 组合预取策略对比

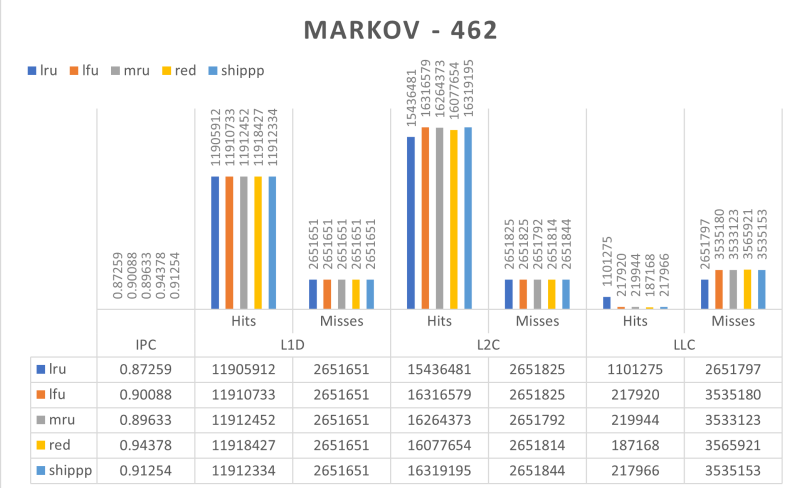
在本次实验中，笔者除了在 L2C 中实现了预取策略，为了更全面地测试不同层级算法组合的效果，在 L1D 中也增加了相应的预取策略（以 `Pangloss` 为例），在此实验设定中选用的替换策略均为 ReD，具体算法组合与性能表现如下所示：



结合日志输出中各层级的 cache 命中率（由于篇幅限制就不作陈列了）分析可知，在 L1D 中增加了数据预取之后，能够极大提升 L1D 的 cache 命中率，虽然在部分数据集上 L2C 的命中率有所降低，但是 L1D 预取带来的性能提升已经较为显著，这样的下降可以说是瑕不掩瑜，无法掩盖最终的性能提升。

LLC 预取策略对比

此处，笔者进一步对比分析在 LLC 实现的五种替换策略的性能差异，其中预取策略固定为在 L2C 上选用 Markov，L1D&L2C 同样不作预取，测得结果如下所示：



	IPC	L1D Hits	L1D Misses	L2C Hits	L2C Misses	LLC Hits	LLC Misses
lru	0.87259	11905912	2651651	15436481	2651825	1101275	2651797
lfu	0.90088	11910733	2651651	16316579	2651825	217920	3535180
mru	0.89633	11912452	2651651	16264373	2651792	219944	3533123
red	0.94378	11918427	2651651	16077654	2651814	187168	3565921
shippp	0.91254	11912334	2651651	16319195	2651844	217966	3535153

从图中可以看出，相较于传统的缓存替换策略，SHiP++ 策略已经取得了一定的性能提升，原因是采用了更加细粒度的 cache 更新策略。而 ReD 性能略优于 SHiP++，其原因是 ReD 中在 LLC 上做了 bypass，将一些初次访问到的数据从 LLC 中 bypass，降低了 LLC 中被污染的概率。当然，可以设想到的是，如果进一步采用 `ReD plus SHiP++` 的混合策略可以带来更高的性能收益。

（二）提高部分

由于涉及到 ***预取&替换联合算法*** 的可参考工作较少（几乎没有表现良好的设计方案），故再三查阅资料发现没有找到适合复现的相关工作后，直接选择从简处理对 `L2C Pangloss- LLC ReD` 稍作改进后得到联合优化算法 `L2C PanglossC - LLC ReDC`(此处的命名较为随意，仅取 `Combined` 一词的首字母作为后缀加以区分)。

首先在 `cache.h` 头文件中，增加如下的 `CacheState` 结构体来跨越不同的缓存操作（即预取和替换）共享重要的状态信息，并在 `panglossC.l2c_pref` 和 `redC.llc_repl` 中进行相应的变量声明（修改&读取同一个`cache_state`）。

```
/* cache.h */
struct CacheState {
    uint64_t recent_prefetch_addr;
    uint32_t recent_replace_way;
};

/* panglossC.l2c_pref */
CacheState cache_state;
/* redC.llc_repl */
extern CacheState cache_state;
```

- `recent_prefetch_addr`（最近的预取地址）：
  - 这个字段存储了最近一次预取操作的地址。
  - 该信息可以被用于替换策略中，以避免刚被预取进缓存的数据被过早地替换掉。例如，如果预取的数据还没有被充分使用，那么替换算法可以选择保留这些数据，而替换其他的缓存行。
- `recent_replace_way`（最近的替换方式，~~在本次设计中其实没有具体涉及，有待开发~~）：
  - 这个字段可以存储最近一次替换操作中被替换缓存行的具体位置或其他相关信息。
  - 该信息可以用于预取策略中，以调整未来的预取决策。例如，如果某个特定的缓存行频繁被替换，可能意味着预取策略需要调整，以减少这种缓存行的预取或者改变预取的模式。

然后需要在 `panglossC.l2c_pref` 预取成功时对该全局变量进行维护，即代码中的 ***Optimisation*** 注释处。

```
uint32_t CACHE::l2c_prefetcher_operate(uint64_t addr, uint64_t ip, uint8_t
cache_hit, uint8_t type, uint32_t metadata_in)
{
   ...
    // 进行预取
   int next_delta = new_delta;
   uint64_t next_addr = addr;
   int l2c_prefetch_degree = (MSHR.SIZE - MSHR.occupancy) * 2 / 3;
   if((type == PREFETCH) && (cache_hit == 0)) {
      l2c_prefetch_degree /= 2;
   }
  for(int i = 0, prefetch_count = 0; i < l2c_prefetch_degree && prefetch_count <
l2c_prefetch_degree; i++) {
  // 获取当前 next_delta 对应的最好的预取策略
  int best_delta = get_l2c_next_best_transition(next_delta);
  // 如果找不到合适的 delta 则退出
  if(best_delta == -1) {
    break;
  }
```

```
  // 预取在 delta cache 构成的 Markov 图中概率超过 1/3 的节点
  else {
    // 由于概率超过 1/3 的节点不可能超过2个
    int candidate_way[2];
    int max_LFU[2] = {0};
    // 计算在当前 set 中 LFU_count 的总和 进而计算概率
    int set_LFU_sum = 0;
    for(int j = 0; j < L2C_DELTA_CACHE_WAYS; j++) {
        int lfu_count = L2C_Delta_Cache[next_delta][j].LFU_count;
        set_LFU_sum += lfu_count;
        if(lfu_count > max_LFU[0]) {
            max_LFU[0] = lfu_count;
            candidate_way[0] = j;
        }
        else if(lfu_count > max_LFU[1]) {
            max_LFU[1] = lfu_count;
            candidate_way[1] = j;
        }
        else ;
    }
    // 接下来判断前两个候选是否满足要求
    for(int j = 0; j < 2; j++) {
        // 如果满足预取条件 则进行预取
        if(max_LFU[j] * 3 > set_LFU_sum && prefetch_count < l2c_prefetch_degree)
{
            // 计算预取地址
            uint64_t pref_addr = ((next_addr >> LOG2_BLOCK_SIZE)
                    + (L2C_Delta_Cache[next_delta][candidate_way[j]].next_delta
- L2C_DELTA_CACHE_SETS / 2))
                    << LOG2_BLOCK_SIZE;
            uint64_t pref_page = pref_addr >> PAGE_SIZE_OFFSET;
            // 判断预取的 block 是否在当前 page 中 并且确实需要进行预取
            if((page == pref_page)) {
                prefetch_line(ip, addr, pref_addr, FILL_L2, 0);
                prefetch_count++;
                                /* Optimisation */
                //更新共享状态
                cache_state.recent_prefetch_addr = pref_addr;
            }
        }
    }
  }

  // 向前走一步
  next_delta = best_delta;
  uint64_t pref_addr = ((next_addr >> LOG2_BLOCK_SIZE)
        + (best_delta - L2C_DELTA_CACHE_SETS / 2))
        << LOG2_BLOCK_SIZE;
  next_addr = pref_addr;
  }
   ...
}
```

接着便是对 `redC.l2c_repl` 的修改，由于笔者重点考虑的是 ***预取对替换的影响***，因次这部分的修改也是理论上提高性能表现的关键，具体优化的角度如下：

- ***提高预取数据的留存率***：通过降低匹配预取地址的缓存行的替换优先级（RRPV值），使这些预取数据在缓存中的存留时间更长，降低了预取数据被替换的可能性，从而增加了预取操作的有效性和性能效益。

```
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) {
  // 判断是否需要 bypass
  if(ReD.bypass(full_addr, ip, type)) {
      return LLC_WAY;
  }

  // 如果是 WB 操作
  // 或者访问的地址在 ART 中命中
  // 或者访问的地址在 PCRT 中具有较高的 reused
  // 则不需要进行 bypass 正常按照 SRRIP 进行处理即可

  /* Optimisation */
  // 检查预取的地址是否在即将被替换的缓存行中
  for (int i = 0; i < LLC_WAY; i++) {
      if (current_set[i].address == cache_state.recent_prefetch_addr) {
          // 如果预取地址即将被替换，适当降低其RRPV值 -> 提高优先级
          if (rrpv[set][i] > 0) {
              rrpv[set][i]--;
          }
      }
  }

  // 标准 SRRIP 替换算法
  ...
}
```

- ***区别对待预取命中和常规命中的缓存行***:
  - 对于预取命中，适度提高 RRPV 值以减少预取数据的替换频率；
  - 对于常规加载命中，将 RRPV 值置为0，优先保留这些数据；
  - 对于未命中情况，则将 RRPV 值设为较高但不是最大值，以平衡预取与常规加载。

```
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) {
  ...
// 如果是由预取命中的缓存行，可能需要特殊处理
  if (type == PREFETCH && hit) {
      // 针对预取命中的情况，可以调整其在替换策略中的优先级
      // 例如，可以选择不立即将其RRPV值置为0，以避免频繁替换
      rrpv[set][way] = std::min(rrpv[set][way] + 1, static_cast<uint32_t>
(MAX_RRPV));
  }
  else if (hit) {
      // 对于常规加载命中，将RRPV值置为0
      rrpv[set][way] = 0;
  }
  else {
      // 对于未命中，将RRPV值置为一个较高的值，但略低于MAX_RRPV
```

```
        // 这样做是为了在预取和常规加载之间保持平衡
        rrpv[set][way] = MAX_RRPV - 1;
    }
}
```

至此便完成了基于 **简单的预取&替换策略联动** 对算法实现的优化，然后对 `L2C PanglossC - LLC ReDC` 同样在六个 trace 上进行模拟测试，其中 L1D&LLC 同样均不进行预取，且热身和测试指令数保持不变，具体性能表现见下表：

	436.cactusADM	437.leslie3d	450.soplex	459.GemsFDTD	462.libquantum	482.sphinx3	avg score
pangloss-red	0.4109	1.15811	0.39464	0.49048	`0.9379`	`1.20919`	`0.76687`
panglossC-redC	0.41086	1.15221	0.38944	0.48692	`0.94056`	`1.21726`	`0.766208`

观察数据可知，优化后的联合预取&替换算法虽然在整体 AvgScore 性能表现上相较原算法组合基本保持不变，但是在部分数据集的 IPC 上有一定程度的提升。这也能够从理论实现与实际应用的角度进行剖析：

- ***数据访问模式***：不同数据集有不同的访问模式，而预取和替换策略的有效性很大程度上取决于这些模式。或者说上述联动方法由于从理论上就不够 **泛化** ，其实际效果是高度依赖于具体的应用场景和数据特性的。例如，如果数据访问模式高度可预测且具有强烈的局部性，预取可能非常有效。然而，在随机或不规则的访问模式下，预取可能引入更多不必要的数据，导致有价值的缓存行被替换。
- ***预取精度***：还要考虑到是否改进前 `pangloss` 预取算法从理论or实现的角度精度本身就不是很高，这可能会导致大量不必要的数据被加载到缓存中，降低缓存的有效利用率。在这种情况下，即使按照上述联动方法优化了预取数据的保留策略，也因为本身预取的不准确性而导致性能下降

当然，这也可能与 ***观测指标的不全面*** 以及 ***测试 trace 数量*** 的相对匮乏有关，之后可以结合对结构体中 `recent_replace_way` 的应用（也就是替换对预取的影响角度），进一步探索性能表现更佳的预取&替换联合优化算法，同样也可以不只局限于 L2C 层级的预取和 LLC 层级的替换，进行更复杂的排列组合与不同层级间的联合优化。