Cache 考核实验报告

王世鑫

2024 年 9 月 10 日

摘要

本实验分别实现了传统 LRU 淘汰算法在单线程以及单把锁的多线程版本，用于展示 lock contention 对性能的影响。另外实现了基于无锁队列和智能指针的近似 LRU 算法、类似于 Redis 方案的采样 LRU 算法。本实验分别对不同线程数量、不同缓存容量以及不同缓存访问行为三个角度出发对上述三种缓存进行性能测试，报告还介绍了本实验的三种缓存的部分实现细节以及其对性能的影响. 对考核和拓展阅读文章的学习总结，最后还有本人对系统方向优化工作的感想。

关键字： LRU　采样 LRU　lock-free　无锁队列　并发编程

1　实验报告

1.1　实验目的

本次实验主要通过实现、讨论两种无锁 LRU 方案和传统单线程/一把锁的 LRU 方案，从算法和并发两个角度上学习 LRU 缓存理论，训练单机单 CPU 并发编程能力。通过性能测试找出各 LRU 实现特点，深化原子操作等并发编程技巧，提高业务基础能力。向读者大致展示本人的学习和实践能力。

1.2　实验平台

本实验设计中开发、性能测试均在具备以下软硬件的系统上进行：

- Intel Core I7 8750H 2.20GHz ∼ 4.10GHz
- 16GB DDR4 2667MHz
- 192 KB L1d, 192 KB L1i, 1.5M L2, 9M L3
- Linux 6.6.47-gentoo-dist x86_64 GNU/Linux
- gcc version 14.2.1 20240817 (Gentoo 14.2.1_p20240817 p4)

实验采用 xmake 作为构建系统，使用的与本实验主题相关的开源软件在参考文献中列出。使用的其他软件请参见`xmake.lua`文件。报告本身使用 LaTeX 编写。编译运行实验代码、编译本报告可通过以下命令完成：

```
git submodule update --recursive --remote --init
xmake f -m release; xmake -j 12; xmake r
```

1.3 实现细节

所有实现中使用的 Hash 算法均为 MurmurHash3[1] 中的 x64 版本，输出的是 128 位 Hash 值。本实验对其输出的 128 位 Hash 值做`h[0] ^ (h[1] << 1)`处理以适应所使用的 HashMap 对象的要求。单线程版本的 LRU 缓存使用`std::unordered_map`作为 hash 索引，其他两个无锁 LRU 缓存使用`libcuckoo::cuckoohash_map`[5] 作为 hash 索引。无锁队列为`moodycamel::ConcurrentQueue`[2].

1.3.1 单线程/单把锁 LRU 缓存

单线程 LRU 缓存 (后称 N 方案，Naïve-LRU) 实现了严格的 LRU 淘汰算法，直接使用 STL `std::list` 存储缓存数据. 在 hash 表中存储各节点对应迭代器。节点的提升操作使用如下代码实现

```
m_cache_list.splice(m_cache_list.begin(), m_cache_list, it->second);
```

其中 `it->second` 是指向该链表节点的迭代器，该操作将它指向的节点的位置关系移动到`m_cache_list.begin()`之前，所包含的元素并不会发生移动或拷贝。

1.3.2 无锁 FIFO-Hybrid-LRU

该算法使用两个无锁队列和 Hash 表实现（后称 H 方案, FIFO-Hybrid-LRU）。

该方案从设计到实现并不容易，迭代了数个版本。本文最初设想实现一个无锁双链表来克服 N 方案中单把锁造成的竞争，在尝试过程中发现，并无办法实现对两个内存位置原子地进行修改，即 CAS2。历史上确有工作在理论上倚靠 CAS2 操作实现无锁双链表。除了使用 CAS2 的链表理论，还有为单链表增加前驱节点提示的类双链表方案，因依赖这种没有保证的前驱提示来实现 LRU，本文认为其行为不可预测性太大，遂未采用。除了上述两种链表方案，还有一种方案每次新插入链表两个节点的方案: 数据节点的前驱为一 Dummy 节点，Dummy 节点的后继为数据节点。在本文的应用场景中，在 hash 表中存储对应数据节点的 Dummy 节点，当要进行 LRU 提升操作时删除该 Dummy 节点的后继节点，即数据节点。在 LRU 的使用场景中，频繁的提升会造成大量的垃圾 Dummy 节点，且难以利用现有设施，因此本文考虑接下来要介绍的方案。

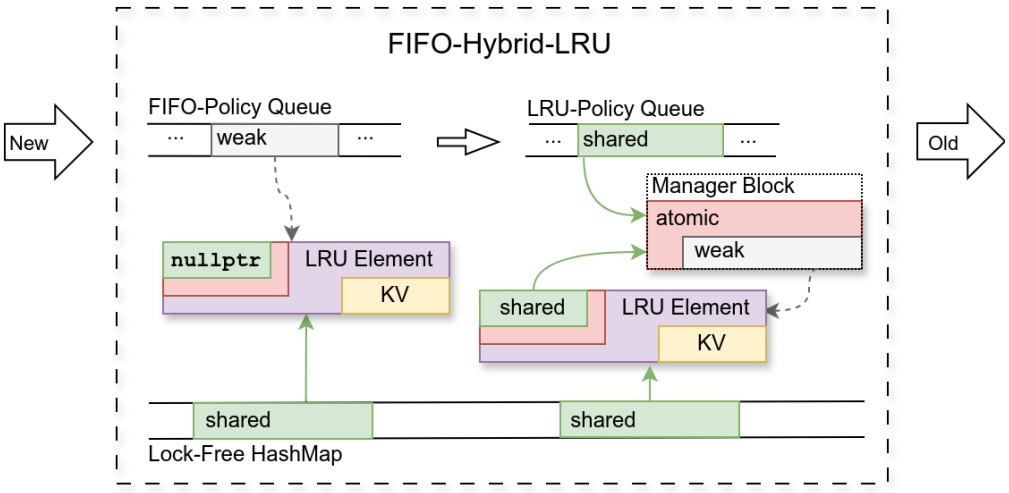


图 1: FIFO 混合 LRU 缓存结构示意图

LRU 中使用链表的根本目的是为了利用链表节点前驱后继关系表达 LRU 的提升和 Aging. 使用双链表是为了方便提升操作将节点从链表中分离，然后再插入链表前部。除此之外，LRU 场景下的双链表就像队列一样工作。因此本文的方案是直接使用无锁队列代替双链表，入队的对象类型一智能指针，并用 hash 表索引与该智能指针相关的数据。

该方案中，将队中存放的智能指针置空实现类似于分离链表节点的操作，与上文中介绍的 Dummy 节点链表一样，会造成空间浪费。为了缓解这个问题，本实现引入 FIFO 淘汰算法与基于队列的 LRU 混合，如图1所示。

该方案中，FIFO-Policy 队列中直接存放指向 LRU Element 的`weak_ptr`，对应的`shared_ptr`被 hash 表持有。LRU Element 包含用户 KV、指向指向 Manager Block 的指针。LRU-Policy 队列中存放指向 Manager Block 的智能指针，其字段如图所示。

访问缓存时，首先通过 hash 索引到对应 LRU Element 对象，之后立即对其中的指向 Manager Block 的指针做原子 `exchange(nullptr)` 操作 [1]，其返回值由两种结果：

1. `==nullptr` 表示该元素被 FIFO-Policy 队列管理，无需提升;
2. `!=nullptr` 表示该元素被 LRU-Policy 队列管理，需要提升。

上述`exchange`操作只有一个线程会返回非空结果，该线程随即对其进行提升. 同一时刻访问该元素的其他线程只能看到空指针，便认为其无需提升.

成功`exchange`到 Manager Block 指针的线程接下来将其指向 LRU Element 的指针字段原子地置空, 相当于链表 LRU 中将节点从链表中分离的操作。此后，该线程将被访问 LRU Element 对应的`weak_ptr` 重新推入 FIFO-Policy 队列，至此完成提升操作。

[1]实现中，所有原子操作遵循：读使用 acquire 内存序，写使用 release 内存序，exchange 使用 acq_rel 内存序

插入新元素时有必要的话先淘汰一个旧元素，然后将分配好的包含用户数据的 LRU Element 智能指针先插入到 hash 表，入队 FIFO-Policy Queue 即可；删除特定元素操作仅需要从 hash 表中删除对应元素即可，两个队列中引用对应 LRU Element 的`weak_ptr` 不会阻碍用户数据所占内存的回收。

旧元素**淘汰操作**需从 LRU-Policy Queue 中出队一个指向 Manager Block 的智能指针; 原子地 `exchange(nullptr)` 该 Manager Block 中指向 LRU Element 的指针对象，若返回值为 `nullptr` 则说明对应元素要么被其他线程淘汰，要么当前 Manager Block 为提升操作产生的垃圾。继续尝试前面描述的步骤，直到能够锁定一个 LRU Element. 进而获得对应元素的 Key，从 hash 表中删除。

与其他两种方案不同的是：由于结合了两种算法，采用了两个无锁队列。需要平衡两个队列的大小。本文中的方案为用户配置固定比例，默认 FIFO-Policy Queue 占总容量的 80%, LRU-Policy Queue 占总容量的 20%. 每次新增或者提升元素后，需要触发一次平衡操作，若该比例失衡则从 FIFO-Policy Queue 出队元素，入队 LRU-Policy Queue, 直到比例恢复。

该方案的两个队列大小关系可动态地结合多个缓存指标进行学习，这是一个优化的方向。但是从软件结构和算法上来看，该方案逐渐趋同于 FrozenHot。后者从各个方面来看都要比本方案优秀太多。作为实验报告，本文不再进一步探讨 FIFO-Hybrid-LRU 方案的优化空间。

1.3.3 无锁采样近似 LRU

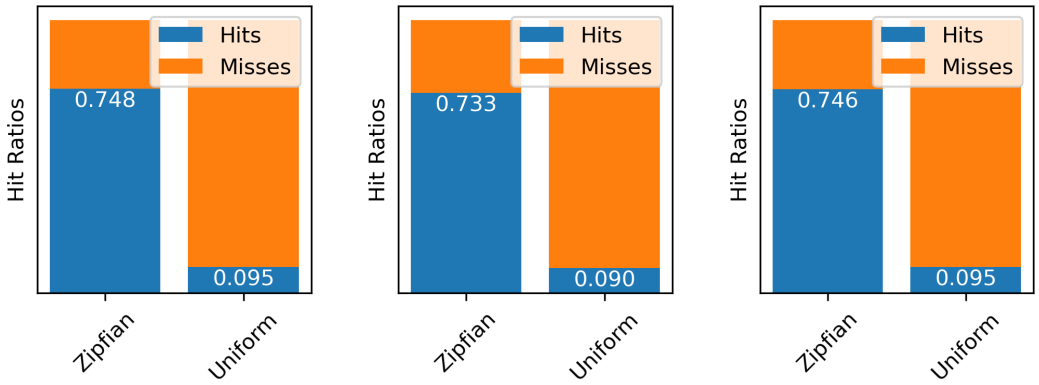
本实验复现了 Redis 中的采样 LRU 算法（后称 S 方案，Sampling-LRU），并为其添加了无锁并发支持。

为了支持随机采样，本实现实现了支持随机访问的对象池，该对象池以块 (chunk) 为单位管理内存[2]，不保证内存的连续性。采样算法随机给出的元素下标对应内存位置若未被分配，则返回一个空指针。下游系统 Deallocate 操作将会把当前对象所占内存放入一个无锁队列中回收，下次 Allocate 操作优先从回收队列中获取。

系统维护一个 32 位无符号整形变量作为 LRU Clock,每次对缓存的访问都会使得 Clock 原子地增 1 来表示时间的变化。每个缓存节点包含节点上次访问时刻，节点当前被采样状态等字段。其中节点上次访问时刻为一个 32 位无符号整数，用于表达元素之间的新旧关系。由于大致计算不同节点间的老旧关系并无需准确的数据，因此对该字段的更新和读取以及对 LRU Clock 的访问全部使用 `relaxed` 内存序，以尽可能减少原子操作对性能的负面影响。

节点的**提升操作**将会给其上次访问时刻赋最新 LRU Clock 值，相比前两种通过变化数据结构内部节点位置关系的方式，仅需要 `relaxed`地原子修改一个内存位置。

[2]块大小对齐到 4K



(a) Naïve-LRU Hit Ratio (b) FIFO-Hybrid-LRU Hit Ratio (c) Sampling-LRU Hit Ratio

图 2: 各 LRU 实现单线程下对两种访问类型模拟的命中率测试示意图

插入新节点需要从对象池中申请一块内存，在上面构造后赋予当前访问时间，并插入到 hash 表中。当空间不足时，执行采样算法选取若干待决定元素，从中选取距离本次访问最旧的对象淘汰。采样通过服从均匀分布的随机整数生成器生成若干在区间 $(0, \text{Number of cached})$ 的整数作为采样下标，并通过该下标从对象池中获取该对象存储区域的指针。采样算法对当前时刻分别与两个待决定对象的上次访问时刻做差并对两者进行比较来区分出两者的新旧关系。通过对做差结果取模，可以消除溢出对数据带来的影响。

1.4 实验设计

本小节对上述 LRU 方案从命中率、吞吐量以及内存消耗三个维度进行性能测试。并分析三种方案中并发管理占总时间消耗的比例。其中命中率、吞吐量和内存消耗数据为同一次实验测得。**若未特别注明，则默认缓存大小为总测试大小的 1/10，进行 30 次测试取平均值，每次测试重置缓存**。每次缓存失配罚执行`_mm_pause()`50 次。

本实验使用 Zipfian 分布模拟有局部性的缓存访问，其 skew factor 采用实现默认的 $\theta = 0.99$[3].

1.4.1 命中率

首先对三个缓存实现在**单线程**下，分别用 Zipfian 分布以及 Uniform 分布生成测试数据对它们进行测试。测试结果如图2所示，三者命中率差距较小。两种无锁缓存方案命中率表现略逊于 Naïve 方案。

由于并发方案为了并发性能在 LRU 原始理论的某些方面做了妥协和放松，因此其并发条件下命中率也许随并发规模变化而变化。如图3所示，三种缓存方案多线程下对 Zipfian 类

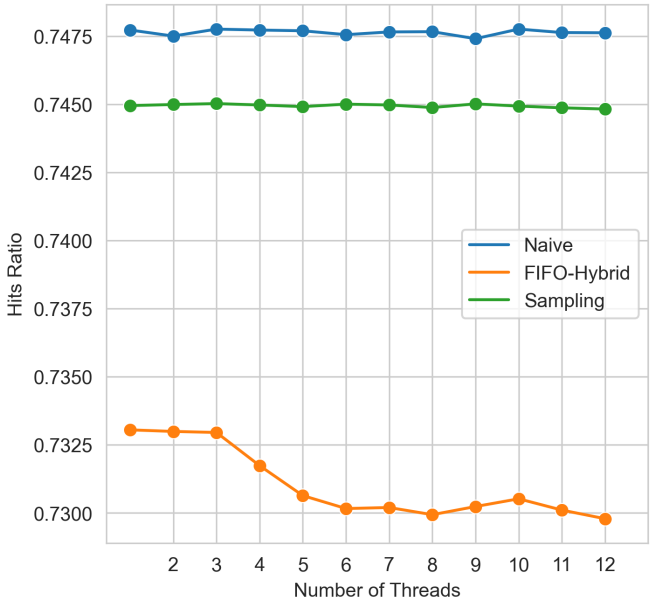


图 3: 各方案在多线程下的 Zipfian 型缓存访问命中率

型访问命中率均维持在 72% ~ 75% 以内，与单线程下测试相近。其中 Naïve 和 Sampling 方案命中率稳定，分别保持在 74.5% 和 74.7% 左右。而 FIFO-Hybrid-LRU 命中率在 72.9% ~ 73.3% 之间浮动。这种现象与两种队列大小比例相关。

1.4.2 吞吐量

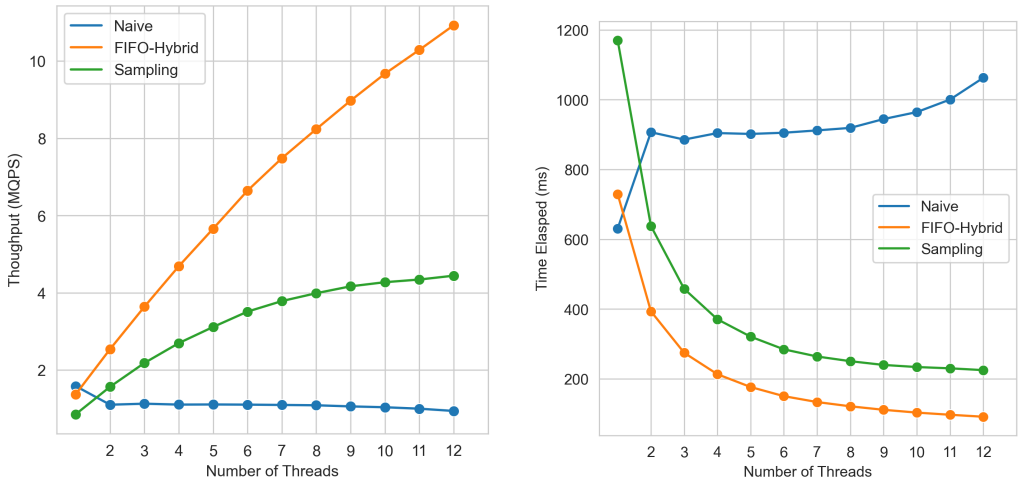
基如图4所示，于上文描述的实验方法，得出三种 LRU 完成试验所耗时间 $l(ms)$ 以及吞吐量 $t(\text{MQPS})$，

$$l = l_{\text{hash query}} + l_{\text{promotion/balance}} + l_{\text{insertion}} + l_{\text{penalty}} \tag{1}$$

$$t = \frac{10^6}{l}\text{Q/ms} = \frac{1000}{l}\text{MQPS} \tag{2}$$

其中4(a)显示专为并发环境设计的两个 LRU 方案其吞吐量随着线程数增加而增加，符合本次实验目标。本实验实现的 FIFO-Hybrid-LRU 更胜一筹，原因有三：

1. 其内部使用的无锁数据结构是经过开源社区检验的优秀实现；
2. 该设计以命中率略差的代价为并发性能赢得了空间；
3. Sampling-LRU 需要为每个元素实现类似锁的机制，以避免并发采样过程中发生元素删除或回收，除此以外相比 Redis 的节点内单线程实现，还有其他并发引入的管理开销。



(a) 吞吐量 (b) 实验耗时

图 4: 各方案在多线程下的 Zipfian 型缓存访问吞吐量和实验耗时

另外，作为基准对比，Naïve 方案吞吐量在线程数 ≥ 2 后相对其他两个实现变化不大，但总体还是随线程数增加而减小。这一点从图4(b)实验耗时上看更为明显。值得注意的是，吞吐量计算过程中，实验所耗事件如等式1所示：缓存失配，以及引发的罚时延迟都算入消耗时间内。

1.4.3 两个无锁方案的不严谨热力图分析

本小节对三种缓存实现在 Debug 模式下使用 perf 进行调用热力分析，大致可以反映缓存系统中关键操作耗时占比。其性能表现无法和 Release 模式媲美，因此仅作参考。

如图5所示，FIFO-Hybrid-LRU 的 `get`操作中，耗时主要被 hash 查询占用，有 87.65%；

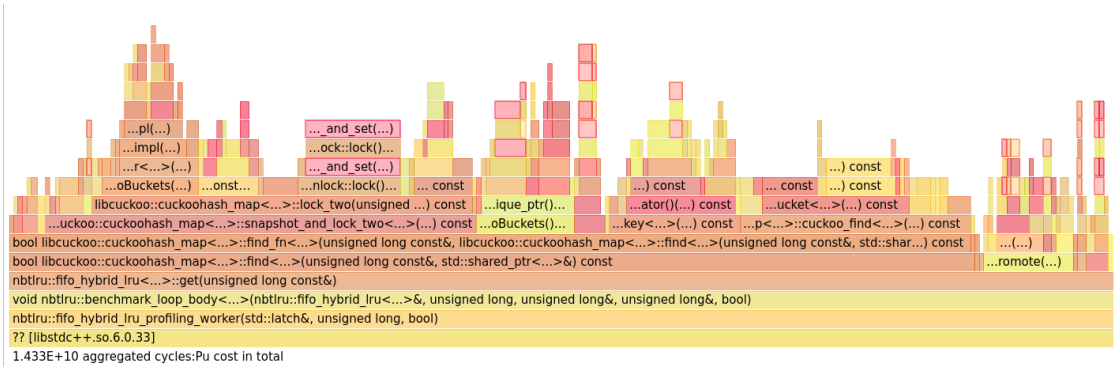
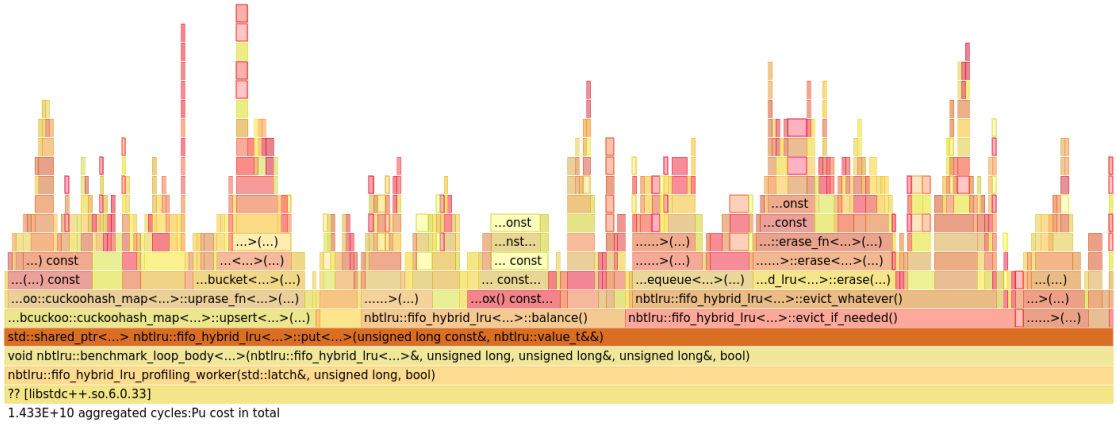
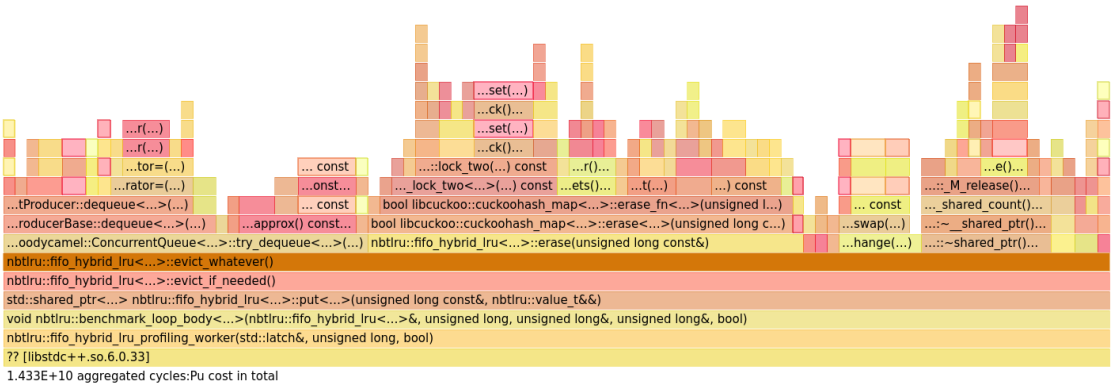


图 5: FIFO-Hybrid-LRU 的 get 操作调用热力图



(a) put 操作总览



(b) evict_whatever

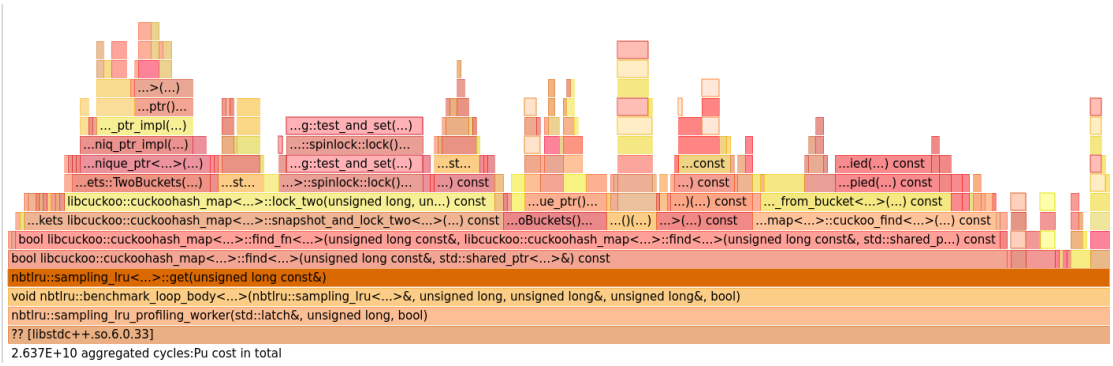
图 6: FIFO-Hybrid-LRU 的 put 操作调用热力图

`promote`操作只占用`get`的 8.02%。者表明本实验实现的 FIFO-Hybrid-LRU 虽然同为利用元素位置关系表达 Aging 的实现，但其`promote`操作并未成为其瓶颈。

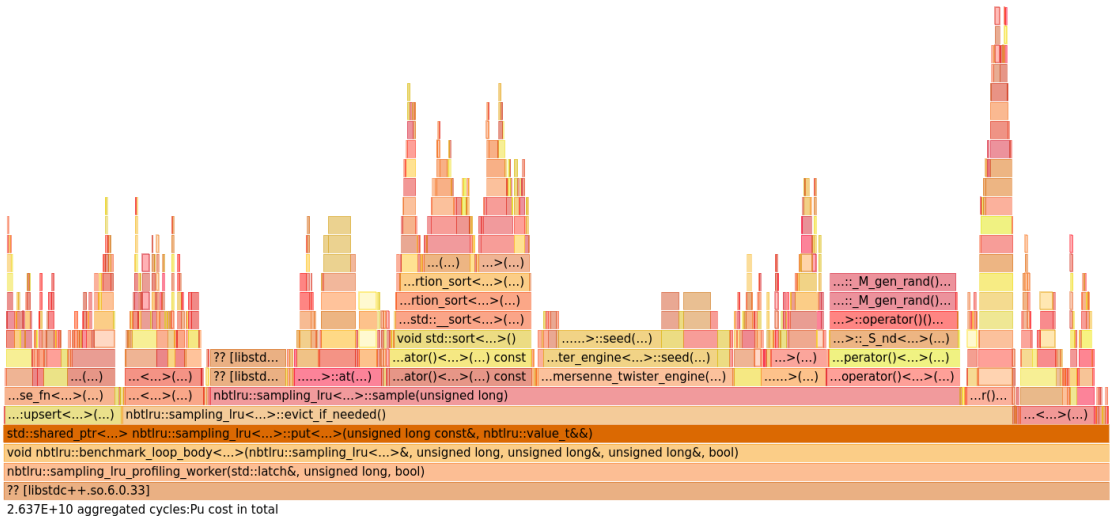
再看图6(a)，FIFO-Hybrid-LRU 的`put`操作中耗时主要集中在更新 hash 表、平衡两队列大小以及淘汰旧元素上。三者耗时分别占`put`总耗时的 28%, 23.81%, 35.2%. 淘汰旧元素操作如图6(b)所示，其中主要耗时有元素出队及从 hash 表删除数据，分别占`evict_whatever`的 33% 和 39.4%.

有关 Sampling-LRU，如图7(a)所示其`get`操作主要耗时在 hash 表查询上，占 90%; 通过更新访问时刻达到提升的操作占比极小，仅有 1.7%，这与其仅写入两个内存位置，并且使用最宽松的 relaxed 内存序有关。可见 Sampling-LRU 非常适合读密集场合使用。

而其写性能就差强人意了，如图7(b)所示: 当缓存容量耗尽时开始，每次新增一个元素就要淘汰一个旧元素。本实现中需要使用随机引擎生成随机下标、对选中的元素进行类似上



(a) get 操作



(b) put 操作

图 7: Sampling-LRU 的 get 和 put 调用热力图

锁操作，再从中挑选一个最久未使用的淘汰。算法最后还要为每个被选中元素解锁。这一过程中相比 Redis 的实现，为了保证并发访问的正确性付出很大的管理代价。这也解释了为什么在本次实验中其性能表现不如 FIFO-Hybrid-LRU. 当然本人的 Sampling-LRU 实现很粗糙，对其进行优化会有更好的性能表现。

1.5 实验总结

本次实验清楚地展示了两种 LRU 缓存实现方案 (基于数据结构内元素位序、基于时间戳) 的基本特点，深化了本人在当前领域的认识，和并发编程能力。更具体地认识到了不当使用锁会造成的严重性能后果。此外, 本人还学习了 Redis 简洁的设计理念。其各节点内单线程运行的方式杜绝了线程间并发管理开销，从软件结构层面避免了很多并发场景性能问题。

2 对系统优化工作的感想

计算机系统是一个有限的系统，对其进行优化就是找到各种权衡 (trade off) 点，放弃一些我们不在乎的特性，来换取一些我们关注的性能。

2.1 软件结构和新算法

本人目前接触到的所有优化相关的知识和技巧，都是在讨论如何提升软件在并发条件下的性能，主要是从软件结构入手进行优化。比如 FrozenHot 的无锁不可变前端缓存 [8]，线程池用的 WorkStealing 队列，本人认为都是一种软件结构上的创新。它们都是为了减少线程间不必要的互操作，但由于其应用场景不同需求不同，其实现形式差异很大。这说明计算机系统方向的优化，虽有成千上万的应用场景，但存在若干基本的共有的权衡点，优化工作可以从这些共同的点入手。

要了解定位到这些点并不容易，需要从业人员对计算机通用硬件、底层软件的原理有深刻的理解和把握。看似简单的计数器，开发者可以在多个线程间简单地原子访问，也可以根据需求不同，设计使用引用计数、设定上限的近似计数 [3] 等计数器 [6]。前者可对其原子访问内存序进行优化：增加引用量不会导致生命期管理事件，任何线程对其大小均不感兴趣，因此可采用 relaxed 内存序；引用计数减小可能会导致内存回收，所有线程需要了解精确的计数大小, 同时也希望其他线程即使看见自己对计数的修改，因此采用 acq_rel 内存序。后者各线程局部维护一个计数器，各线程在计数器未接近指定数值之前线程间减少共享数据的频率，来减少竞争，提升 CPU 缓存一致性。上述优化基本是深度结合应用需求和计算机软硬件底层原理, 在软件结构上进行优化得来的。

在缓存优化工作领域，上述方法为优化缓存并发读写性能。除此之外，还有很多工作结合机器学习等新算法对缓存淘汰算法进行优化, 考核文章中的 GL-Cache 就提出通过分组学习的方式，减少各元素平均学习成本 [9]。不仅从对学习算法进行了创新，还从软件结构层面减小了学习引入的读写开销, 是一种综合两种方向的优化。

无论是优化算法，还是优化读写，本质上都是为提高缓存吞吐量而服务的。因此优化工不能拘泥于一个方向，应像 GL-Cache 一样，结合发挥多种想法的优势为优化的根本目标服务。

2.2 新硬件

根据本人的了解，新硬件主要为了解决通用硬件难以满足某些特定需求而设计的。计算方面如专为 AI 计算优化的各种 APU，NPU，TPU etc. 以及相对更灵活的 FPGA. 存储方面有持久内存，更廉价的固态硬盘等。新硬件的加入不会改变计算机系统是有限系统的事实，新硬件只是拓展了其解决特定问题的能力，因此上文观点仍然适用。

[3] 软件对计数器当前具体数据不感兴趣，只对计数总量接近设定值时感兴趣

不过现有软件是为通用硬件设计的，如果不对软件加以修改，很难发挥新硬件的优势。比如 NBJL 的 FlatLSM，对 RocksDB 做了适应 PMEM 的改造，PMEM-Table 实现了通用硬件下 WAL 和 MEM-Table 两个组件的功能。通过对 LSM 结构的改造，免除 WAL, 减少 L_0, L_1 的写放大问题，写性能有显著提升 [4]。

新硬件不但拓展了计算机解决特定问题的能力，也为优化工作带来新的机会和突破口。需要开发人员继续结合应用需要以及新硬件特性进行探索, 寻找新的性能权衡点。

2.3 新应用场景

随着 5G 和云计算的发展，目前越来越多的应用上云，这不仅是出于经济的层面考虑，结合低延迟通信和密集的计算存储资源可以创造出新的应用场景, 比如云游戏, 中心化的区域智能交通规划等。

上述场景，针对系统优化工作来说，个人认为主要是以应用集群化，分布式化作为大的优化方向。除了能够提供弹性横向并发规模拓展，计算和存储集中化也是集群带来的优势。集群内较低的通信成本可以允许应用在集群中做一些计算或存储资源共享。比如 CaaS-LSM 微服务化 Compaction，使得 LSM 更适合集群场景 [7]。

2.4 实验后感受

优化工作需要大量的尝试和思考。本次实验中实现的 FIFO-Hybrid-LRU 方案经过几次修改和迭代，甚至推倒重做。试错的过程也是人学习的过程，改的越多，人学到的就越多。LRU 的思想也许一页纸就能概括，但完成整个实验所习得的经验不是一页纸就能概括的了的, 这种感觉是只看书得不到的。

除了计算机基础知识，数学和问题建模能力也是非常重要的。一些参数化问题往往能够利用数学建模的手段直接解决, 机器学习理论也是基于数学发展而来的。在看了 Kangaroo 文后的数学建模后，深感本人在数学方面还有巨大的进步空间。

参考文献

[1] Austin Appleby. Smhasher - murmurhash test suite. https://github.com/aappleby/smhasher/, 2008. Accessed: 2024-08-30.

[2] Granberg Cameron. concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for c++11. https://github.com/cameron314/concurrentqueue, 2024. Accessed: 2024-08-30.

[3] Erik Garrison. Dirtyzipf: A zipf distribution generator. https://github.com/ekg/dirtyzipf/tree/master, 2024. GitHub repository.

[4] Kewen He, Yujie An, Yijing Luo, Xiaoguang Liu, and Gang Wang. FlatLSM: Write-optimized LSM-tree for PM-based KV stores. ACM Trans. Storage, 19(2):1–26, 2023.

[5] Bin Fan Li, Zhang Xiaozhou, Richard P. Dick, and David G. Andersen. libcuckoo: A high-performance, concurrent hash table. https://github.com/efficient/libcuckoo, 2024. Accessed: 2024-08-30.

[6] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? https://github.com/paulmckrcu/perfbook, 2024. GitHub repository.

[7] Qiaolin Yu. CaaS-LSM: Compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. ssociation for Computing Machinery, 2(3):28, 2024.

[8] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. FrozenHot cache: Rethinking cache management for modern hardware. In Proceedings of the Eighteenth European Conference on Computer Systems, pages 557–573. ACM, 2024.

[9] Juncheng Yang, Ziming Mao, Yao Yue, and K V Rashmi. GL-cache: Group-level learning for efficient and high-performance caching. In Proceedings of the 21th USENIX Conference on File and Storage Technologies, 2023.