arXiv:2302.06520v1 [cs.DC] 13 Feb 2023

使用乐观访问释放内存：无锁程序中内存回收与分配的混合方法

Pedro Moreno[1] 和 Ricardo Rocha[2]

[1,2]CRACS & INESC TEC，计算机科学系，理学院，波尔图大学 Rua do Campo Alegre s/n，4169-007 波尔图，葡萄牙 [1]pmoreno@dcc.fc.up.pt[2] ricroc@dcc.fc.up.pt

摘要

无锁数据结构是开发并发程序的重要工具，因为它们能够提供可扩展性和低延迟，并避免死锁、活锁和优先级反转。然而，它们需要某种额外支持来确保内存回收。乐观访问（OA）方法具备内存回收所需的大多数特性，但由于它允许在内存被回收后继续访问该内存，因此与传统内存管理模型不兼容。这使其无法将内存释放给内存分配器或操作系统，因此需要复杂的内存回收机制。在本文中，我们扩展了无锁通用内存分配器 LRMalloc，使其支持 OA 方法。通过这种方式，我们不仅能够简化内存回收方法的实现，还能让同一进程的其他部分重新使用这些内存。我们进一步利用操作系统和硬件提供的虚拟内存系统，使回收后的内存能够释放给操作系统。

1 引言

随着计算机硬件近期的发展将提高并行性作为提升性能的主要途径，配套软件能否充分利用这类硬件至关重要。无锁数据结构是并发/并行软件最基础的构件之一，无锁性有助于提升可扩展性，并确保系统不受活锁、死锁和优先级反转的影响 [10]。然而，与基于锁的数据结构相比，无锁数据结构需要额外的支持来管理内存回收。如果所用的编程运行时提供了垃圾收集器，则可以将这项工作交由垃圾收集器处理；但此类垃圾收集器通常并非无锁的，导致整个系统失去无锁性 [18]。

另一种方法是使用特定的内存回收方法。最常见的方法，例如 *pass the buck* [9]和 *hazard pointers* [15]，基于这样一种理念：线程公布自己的坐标，以防其他线程回收它们正在使用的内存。然而，这种方法要求每个线程持续将自己的坐标写入内存，并执行开销较大的内存屏障，以确保这些内存写入可见。更复杂的方法则试图摊销内存写入及其后续所需的内存屏障开销。例如 *drop theanchor* [2]、*hazard eras* [19]、*interval based reclamation* [23]和 *hazard hash and level* [17]等方法。Dice 等人 [7]还提供了一种降低内存屏障开销的机制，但该机制需要硬件/操作系统的支持。

与其让线程公布自己的坐标，一种称为 *optimistic access* 乐观访问（OA） [5], 的较新策略允许线程乐观地访问其遍历过程中的内存，然后再验证该访问是否有效。为了能够检查内存访问的有效性，OA 方法将公布内存回收已发生这一信息的责任转移给回收线程。这样一来，线程不再需要持续向内存写入以公布自己的位置，只需额外执行读取来检查内存回收是否已发生。这些额外读取开销很小，因为它们大多数时候会访问缓存中的内存位置，而且所需的内存屏障开销也更低。

现有基于乐观访问的方法有一个重要缺点：无法将内存释放给内存分配器或操作系统。这是因为线程可能随时读取已被回收的内存。为了解决这一问题，这些方法实现了一种回收机制来管理正在使用的内存。然而，这会导致以这种方式使用的内存无法在同一进程的其他部分重复使用，也无法释放给操作系统。

在这项工作中，我们提出了一种解决方案，无需让整个应用程序了解内存回收方法。我们的方案是以一种特殊方式扩展 *LRMalloc* [12]，这是一款无锁通用内存分配器，

从而保证内存分配即使在释放后仍可读取。

对于内存的内容，或应用程序其余部分如何重用这些内存，不作任何保证。这与乐观访问（OA）非常契合，因为 OA 已确保忽略对可能已回收内存的读取内容，并通过使用危险指针保护待写入的内存免遭回收。

我们首先从内存分配器层着手解决这个问题，调整 LRMalloc，使其不再将 OA 方法使用的内存释放给操作系统。这样一来，OA 内存回收方法的实现便可简化，因为我们不再需要回收机制来管理线程之间的内存分配。这项任务现在由内存分配器负责，因为从一般意义上说，它本就是为此设计的。我们还可以在整个进程范围内重用 OA 方法回收的内存。正如我们将看到的，这一切只需对 LRMalloc 内存分配器进行少量修改即可实现。

接着，为了完善我们的解决方案，并使 OA 方法使用的内存能够释放给操作系统，我们利用了当前操作系统和硬件使用虚拟内存的方式。我们需要确保这些虚拟地址（页）在 OA 方法使用后仍可访问，但并不关心它们映射到的物理内存（页框）中的内容，因此将所有这些页都映射到同一个页框。这样，我们就可以释放这些页原先映射到的所有页框，同时仍使这些页保持有效且可访问。

现代操作系统采用类似的策略。例如，当向操作系统发出内存请求时，不会立即预留页框，而是将所有页都指向同一个 *copy on write* 零填充帧，使这些页变为有效状态。只有当尝试向这些页写入内存时，操作系统才会将零填充帧复制到一个新的空闲页框中，并将该页映射到这个页框。整个过程对应用程序完全透明，应用程序不会察觉到，一开始分配给它的内存实际上并没有由物理内存提供支持。我们提出的两种页重映射策略中，一种利用了操作系统的这一行为，另一种则利用当前操作系统的共享内存机制，以更加显式的方式进行重映射。

本文其余部分安排如下。首先，我们介绍相关背景知识。接着，详细阐述支撑我们方法的主要思想，并讨论其当前局限性。然后，我们展示一组实验，将我们的模型与原始 OA 方法进行比较。最后，我们总结全文并提出后续研究方向。

2 背景知识

本节简要介绍虚拟内存和内存分配系统的相关背景知识，并进一步详细介绍 LRMalloc 内存分配器和乐观访问（OA）方法。

2.1 虚拟内存

虚拟内存是一种内存管理系统，它作为抽象层，为现代操作系统中的诸多优化提供了支持。其核心思想是在用户进程所见的内存地址与主存中的实际物理地址之间设置一个转换层。转换由硬件中的 *memory management unit* *translation lookaside buffer* (MMU) 完成，并依赖名为 (TLB) 的缓存。这会带来一定开销：使用虚拟内存时，访问某个内存位置前，首先需要查询该虚拟地址位于物理内存中的何处。为获取物理内存位置，需要额外进行内存访问；不过，借助高效的 TLB，这一缺点在很大程度上得到了缓解。现代系统将页/页框的粒度设定为 2 的幂，通常总大小介于 4KiB 和 1GiB 之间。

虚拟内存的主要优点包括：允许进程超额使用内存，使其能够使用超出物理可用容量的内存；允许多个进程拥有相同的地址空间；在内存压力较大时，可将未使用的页从内存移至持久化存储；还可阻止进程访问或修改不属于它的任何内存。虚拟内存还允许进程之间共享内存，最常见的情况是共享库：多个进程可以使用物理内存中的同一份库副本，但各自将其映射到不同的内存地址。另一个重要的应用场景是高效的进程间通信：通过将同一物理内存区域映射到两个或更多进程各自的地址空间中，即可实现这一点。

进一步的优化还包括按需加载页框：也就是说，进程载入内存时无需全部加载，只有在访问相应的页时，才加载必要的页框。例如，从未调用过的错误处理例程，实际上永远不会载入物理内存。当进程向操作系统申请内存时，也可以采用类似的优化：进程申请的每一页最初都可以映射到同一个零填充帧，只有在实际写入时才映射到空闲页框。正如我们稍后将看到的，这正是我们方案将利用的特性之一。

2.2 内存分配

内存分配器充当进程与操作系统之间的接口，以尽可能少的额外内存浪费和时间开销来满足进程的任意大小内存请求。为此，内存分配器首先从操作系统获取页，随后将其划分，以满足更小的分配请求，之后再将其合并，从而把完整的页归还给操作系统。经典内存分配器 [24] 倾向于采用诸如 *best-fit* 之类的策略：寻找能够满足请求的最小连续内存块；如果该内存块仍然过大，则将其分割成合适的大小，以便保留剩余部分供未来分配。另一种策略

是 *first-fit*：它们不寻找能够满足请求的较小连续内存块，而是简单地使用找到的第一个内存块。这种策略具有速度优势，但可能会增加内存浪费。

一种较新颖的策略是使用 *size classes*，将任何请求向上取整到最接近的大小类。大小类的内存块通过将一个较大的内存块拆分成多个相同大小的内存块来创建。因此，必须谨慎选择大小类，避免类别过多，导致分配大型内存块后只能从中进行有限次数的分配；也要避免类别过少，否则由于没有足够大的较小尺寸可供选择，可能不得不提供远大于需求的分配，从而浪费内存。大小类的处理效率很高，而且往往能改善内存局部性，因此除了提升内存分配效率之外，也能改善应用程序的整体性能。

随着多核处理器的出现，为了进一步提升性能和可扩展性，人们采用了不同的方案来尽量减少线程之间的同步。这些方案催生了诸如 *pri-vate heaps* [1]之类的机制，其中每个线程都有一个私有分配器，并采用特定策略处理在非分配该内存的线程中发生的释放。这些策略可以将空闲内存保留在线程中，直到再次分配；也可以立即将空闲内存返还给最初分配该内存的线程；或者等达到某个阈值后再返还。另一种机制是在共享堆之上使用每线程缓存 [11]。

2.3 LRMalloc

LRMalloc [12] 是一种现代无锁内存分配器，采用了上文所述的大小类和线程缓存。它由三个主要部分组成：(i) *thread caches*，每个线程各有一个；(ii) *heap*；以及 (iii) *pagemap*。图 1 展示了这三个组件、用户应用程序和操作系统之间的关系，

线程缓存为每个大小类使用一个栈，因此，内存请求只需从对应大小类的栈中弹出一个元素，而释放内存则只需将元素压入栈中。当发出内存请求且对应的栈为空时，就会从堆中填充该栈；当释放内存且栈已满时，则会将其内容刷新回堆中。栈的大小受到限制，以防止 *blowup* [1]。缓存是线程本地的，因此它们只

当堆发生填充或刷新操作时，与其他线程同步。

堆负责管理 *superblocks*，这些是从操作系统获取的大块内存，随后会被划分为特定大小类别的内存块，分配给线程缓存。超块通过 *descrip-tors* 进行管理；这是一种包含超块元数据且永不回收的对象。当超块被释放给操作系统时，关联的描述符会被添加到回收池中，以便将来用于新的超块。描述符包含诸如超块起始位置、关联的大小类别、其包含的内存块数量、首个可用内存块的索引以及可用内存块数量等信息。

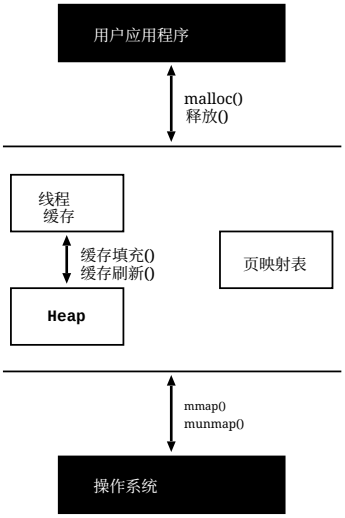


图 1：LRMalloc 概览

包含的内存块数量、首个可用内存块的索引以及可用内存块的数量。

超块有三种状态：（i） *full*，其所有内存块都在使用中；（ii） *empty*，其所有内存块都可供分配；或（iii） *partial*，其中既有可用内存块，也有已分配的内存块。超块的初始状态始终为已满，因为其所有内存块都会立即用于填充缓存。随后，当缓存刷新时，一些内存块被归还给超块，它便会变为部分占用。此时，如果某个线程使用它来填充缓存，它可以再次变为已满；如果所有内存块都归还给它，则会变为空。当超块变为空时，便不能再使用，其内存会被释放并归还给操作系统。线程尝试填充缓存时，会优先使用部分占用的超块；如果没有可用的超块，则会向操作系统申请内存，以创建新的超块。

页映射表是一种简单的无锁数据结构，用于存储每个正在使用的页的元数据。考虑到超块始终与页对齐，且其大小是页大小的倍数，同一页中的内存块始终属于同一个超块。因此，这些元数据包括该页所属的超块及其关联的描述符。页映射表的主要用途是：当某个内存块从缓存中刷出时，查找其对应的超块；或者当通过调用 *free()* 过程从应用程序接收内存时，查找合适的缓存（具有正确大小类别的缓存）。

2.4 乐观访问

无锁数据结构的内存回收方法是一种机制，用于检测从数据结构中移除的节点何时不再被任何正在运行的线程引用，并据此将相应的内存释放给内存分配器／操作系统。通

常，这类方法需要某种验证机制，以避免访问已经回收的内存。

另一种方法是 *opti-mistic access* (OA)方法 [5], ，顾名思义，该方法允许在确认内存尚未被回收之前进行内存访问，随后通过读取特定的 *warning-bit*来检查访问的有效性。如果访问对应的是已回收的内存，则忽略该结果，并从已知有效的内存位置重新开始这一过程。然而，修改操作无法以乐观方式执行，因为乐观的CAS可能会由于ABA问题而错误地成功 [6]。因此，OA采用危险指针策略：在执行任何原子CAS（*Compare-and-Swap*）更新操作之前，先为涉及的所有内存地址分配危险指针以保护它们，然后通过读取 *warning-bit*执行一次额外的有效性检查，从而确保内存在受到危险指针保护时是有效的。随后，这些危险指针会被用于防止回收其所指向的内存。

OA内存回收机制由三个池组成：(i) *ready pool* ，其中包含所有可供分配的节点；(ii) *retire pool* ，从数据结构中退休的节点会被加入其中；(iii) *processing pool* ，用于存放正在回收过程中的节点。回收机制分阶段运行，当就绪池耗尽时，就会触发一个新阶段。新阶段开始时，在该阶段开始前已位于退役池中的节点会被移至处理池。接下来，通过设置所有线程的 *warning-bit* ，通知它们当前正在进行回收。最后，将处理池中受危险指针保护的节点移回退役池，将未受保护的节点移至就绪池。在线程将节点从退役池移至处理池的过程中，若有线程尝试退休一个节点，则必须先协助完成这次移动，然后才能退休该节点。若有线程尝试在一个回收阶段尚未完成时启动新阶段，则必须先协助完成当前阶段，然后才能启动新阶段。

虽然回收机制复杂且耗时，但它很少执行，因此其开销得到了缓解。对于更频繁的操作（例如遍历数据结构），该方法只需针对遍历的每个节点执行一次额外读取，而无需像危险指针内存回收方法那样执行写入；此外，它还只需要成本低得多的内存屏障。在 *total store or-dering* (TSO) 架构（如 x86-64）中，这会转化为一个简单的编译器屏障，不会产生额外的硬件指令。还应注意，写入操作在设置多个危险指针时只需进行一次有效性检查，因此也只需执行一次开销高昂的内存屏障；相比之下，危险指针方法需要为每个节点执行一次内存屏障。这些特性使乐观访问内存回收方法相比当前最先进的方法极其高效且性能出色，同时还具有较低的内存上界，并且不需要操作系统提供任何特定支持。

允许乐观访问可能已回收的节点，意味着节点在被回收后仍需保持可访问，不过，无需保留节点的内容，因为如果访问无效，其结果将被忽略。为确保节点在被回收后仍可访问，系统采用回收机制，使节点能够被重复使用，但不会将其释放给内存分配器或操作系统。

3 我们的方法

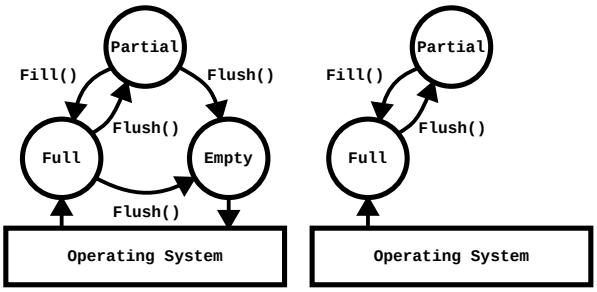
本节首先介绍如何使 LRMalloc 兼容 OA 内存模型，以及如何利用它简化 OA 方法。接下来，我们将介绍如何利用虚拟内存，使内存能够释放回操作系统。

3.1 内存分配器层的内存回收

如前所述，在将无锁数据结构与 OA 内存回收方法结合使用的程序中，回收的内存可以由该数据结构重新使用，但程序的其他部分无法重新使用这些内存，至少在不对内存回收方法和程序其余部分进行大幅修改的情况下是如此。

我们的解决方案是在内存分配器层面处理这一限制，确保即使内存已被释放，仍然可以访问它。分配器仍不会对已释放内存的内容提供任何保证，而且我们也不允许向其中写入数据，因为如果这块内存已在另一次分配中被重新使用，就可能导致数据损坏。为此，我们扩展了 LRMalloc，新增了一个名为 *palloc()* (*persistent alloc*) 的函数。

为了实现 *palloc()*，我们遵循与常规过程相同的流程，即较大的分配，但包含正在分配的内存块的 *superblock* 被标记为 *persistent*。随后利用此标记，确保持久超块永远不会进入 *empty* 状态，即使其中所有内存块都可用也是如此。这一更改确保使用 *palloc()* 分配的内存永远不会释放给操作系统，但仍可供同一进程中任意位置的后续分配重复使用。图 2 展示了超块在标记为持久化之前和之后的状态图。



（a）非持久化　　（b）持久化

图 2：超块的状态图

有了满足这些属性的分配器，我们现在可以大幅简化内存回收方法。由于不再需要 OA 方法最初采用的内存回收机制，我们可以使用一种简单得多的机制，类似于危险指针内存回收方法所用的机制，如算法 1 所示。

算法 1 退休（节点 N）

```
待回收列表.add(N)
如果 隔离列表已满则
  对于 T in 线程 执行
    T.警告位.set()
  结束循环
  内存屏障()
  对于T，在线程中执行
    危险指针集合.add(T.危险指针)
  结束循环
  对于M，在待回收列表中执行
    如果不危险指针集合.contains(M) 则
      待回收列表.remove(M)
      释放(M)
    结束 if
  结束 for
  危险指针集合.reset()
结束 if
```

其思路如下。节点退休时，我们将其加入回收线程的 *limbo list*中；当列表大小达到某个阈值时，便执行回收过程。在此过程中，我们只需设置所有其他线程的 *warning-bit*，然后释放所有未受危险指针保护的节点。

然而，对于具有较长链的 data structures（例如链表），这种机制并不理想，因为随着触发的警告越来越多，需要进行更多次重启。对于链较短的数据结构（例如哈希表），重启的代价不高；但对于链表则不然，这不仅是因为重启会损失大量工作，还因为重启时链的起始部分很可能已经不在 L1 缓存中。

为缓解这一问题，我们实现了另一种警告机制，该机制基于 *Version BasedReclamation* (VBR) 方法 [21]所采用的机制。在这种机制中，我们不再为每个线程设置一个警告位，而是使用一个单调全局变量；每当我们想向所有线程发送警告时，就递增该变量。线程通过比较全局变量中上次看到的值与当前值来检查是否有警告。借助这种机制，线程可以搭其他线程警告的便车：如果自我们想要回收的节点退休以来已经发出过警告，我们就可以不再发送警告。请注意，我们不仅会在看到全局变量递增时利用其他线程发出的警告，还会在尝试通过 CAS 递增该变量但操作失败时利用这些警告；这意味着另一个线程已成功发出警告，我们可以从中受益。算法 2 展示了退休过程，以及它如何搭其他线程警告的便车。请注意，这在先前的方法中是不可能的，因为警告不是以每个 *warning-bit* 线程为单位进行原子处理的。在该算法中，*GlobalClock* 变量表示单调全局变量，*LocalClock* 是用于存储上次观察到的全局变量值的局部变量，而 *LastRetireTime* 是用于利用其他线程警告的局部变量。

算法 2 退休(节点 N)

```
如果隔离列表已满则
  如果最后退休时间 =本地时钟 则
    CAS(全局时钟, 本地时钟, 本地时钟 + 1)
    本地时钟 ⇐全局时钟
  结束如果
结束如果
如果 最后退休时间 < 本地时钟 且待回收列表.size() > X
then
  内存屏障()
  对于 T 属于线程 执行
    HPSet.add(T.危险指针)
  结束循环
  对于 M，遍历待回收列表 执行
    如果不 HPSet.contains(M)则
      LimboList.remove(M)
      释放(M)
    结束条件判断
  结束循环
  危险指针集合.reset()
结束条件判断
最后退休时间 ⇐ 本地时钟
待回收列表.add(N)
```

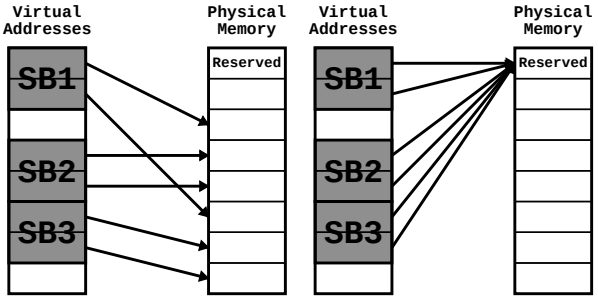
如前所述，采用这种方法，我们会在进程的整个生命周期内保留一部分无法释放给操作系统的内存。如果通过 *palloc()* 分配了大量内存，那么即使进程在剩余生命周期内所需的内存大幅减少，这些内存仍会一直留在进程中。与其他任何无锁内存分配器相比，这种机制的主要优势在于，它不需要操作系统或硬件提供任何额外功能。

3.2 使用虚拟内存

现在，我们已经使内存分配器兼容乐观访问模型，接下来将重点关注它与操作系统的交互。请记住，内存分配器无法释放标记为持久的超块，因为它们必须保持可访问状态。

如果我们结合虚拟内存系统，更仔细地考察这个问题，就会发现，实际需要保持可访问的并非作为后备的物理内存，而是标记为持久化的超块的地址范围，因为对于已回收内存的内容，并没有访问要求。因此，只要能够释放与这些超块相关联的物理内存，同时保留其地址范围，这个问题便可迎刃而解。

可访问。[1] 为此，我们可以将变为空的持久超块的地址范围重新映射到一个默认的预留帧。这样，无论有多少个空超块，它们总共只会占用一个物理内存页框。只要能确保该页框在进程的整个生命周期内始终可访问，这个唯一的页框甚至可以是进程已在使用的页框。图 3 展示了这一重新映射过程。图 3a 展示了多个持久超块，每个超块使用 2 页，且每页都映射到不同的页框；图 3b 则展示了如何重新映射这些超块，以释放它们占用的所有页框，同时确保它们仍可有效访问。



(a) 之前　　(b) 之后

图 3：重新映射前后的内存映射

不过，我们需要谨慎处理，因为虚拟地址空间虽然充裕，却仍然有限。因此，仍需采用某种机制来回收重新映射的超块所占用的虚拟地址。但当 LRMalloc 需要回收包含超块元数据的 *descriptors* 时，实际上几乎已经完成了这项工作。请记住，当一个非持久化超块变为空时，该超块会被取消映射，其描述符则会被加入回收池。之后，当需要一个新的超块时，首先从回收池中获取一个描述符，然后从操作系统映射一个超块，最后用新超块的元数据重写描述符中的元数据。因此，如果我们改用从回收池获取的描述符中存储的地址范围来映射新超块，就能借助描述符顺势回收虚拟地址空间。在实际实现中，我们通过这一机制额外添加了一个回收池，并优先从中获取内存块；原有的回收池则保留给源自非持久化超块的描述符。第二个回收池的用途将在第 4 节中变得更加清晰。

对于实际的重新映射过程，我们提出了 2 种方法。第一种方法是通知操作系统不再需要这段内存。在 Linux 中，这是通过使用带有 $MADV\ DONTNEED$ 标志的 $madvise()$ 系统调用来实现的，该调用会将内存映射恢复到类似于超块最初分配时的状态，即所有页都映射到同一个写时复制的零填充帧。这样会释放此前与该映射关联的所有物理内存，直到再次向其写入数据。请注意，读取这些内存地址范围不会引发缺页异常，只会实际读取零填充帧。使用这种方法，从回收池中获取描述符时，无需额外执行重新映射操作，因为原始地址范围已经有效，可以直接使用。

第一种方法的优点是更简单、更高效，但有两个主要缺点。一个缺点是，尽管 POSIX 标准定义了这个系统调用和标志，但该标准本身并未强制要求采用 Linux 中观察到的行为，因此这种方法不具备可移植性。另一个缺点是，一些乐观访问派生方法（例如 VBR） [21], 会对已回收的内存使用双宽比较并交换（DWCAS）。尽管 DWCAS 肯定会失败[2] （否则会导致损坏），但操作系统无法确认这一点，因此会通过写时复制机制调入一个页框。这不会造成正确性问题，但可能导致一些内存泄漏，因为某些页会被保留给尚未分配的超块。

第二种方法是使用共享内存机制。我们首先定义一个共享内存区域，然后在要释放某个超块时，将其地址范围映射到共享内存区域。我们可以将共享内存区域的大小设为从一页到一个超块之间的任意大小，这会带来不同的性能权衡：如果选择一个超块大小，则重新映射需要一次系统调用；如果选择半个超块大小，则需要两次系统调用，依此类推。请注意，与共享内存区域关联的物理内存此时可以用来存储有用的数据。例如，它可以用来存储 *descriptors*。之后，若要重新使用该超块的虚拟地址范围，我们需要再次将其重新映射到新的内存。请注意，无论共享内存区域的大小如何，这次重新映射都只需要一次系统调用。在 Linux 中，此方法通过使用 $mmap()$ 系统调用实现：使用标志 $MAP\ FIXED$和 $MAP\ SHARED$ 释放物理内存，并使用 $MAP\ FIXED$、$MAP\ PRIVATE$和 $MAP\ ANON$ 重新使用该超块。

尽管这种方法看起来可能有些取巧，但它受到 POSIX 标准的支持。不过，这种支持并非明确规定，而且在 Linux 中，内存统计信息会变得混乱：它会将映射到共享映射的所有地址范围都计入进程的驻留集大小（RSS），尽管实际只使用了一个物理内存共享映射。这种方法也可用于 POSIX 以外的其他操作系统；在回收后的内存上使用 CAS 指令时，也不会导致内存泄漏。它需要额外的系统调用，但我们无法测得这些调用造成的任何性能下降。

[1]请注意，现在我们再次考虑所有超块都可能变为空，即可以释放给操作系统。

[2]它使用带标签指针作为 ABA 防止机制。

4 局限性

LRMalloc 内存分配器采用大小类分配策略，这意味着它通过此机制处理大小不超过合理上限（16KiB）的分配。对于所有大小类分配，LRMalloc 都使用大小相同（2MiB）的超块，这简化了我们的重新映射逻辑，因为我们可以将已退休的超块地址重新用于不同的大小类别。在大多数场景中，这种方式很理想，因为大多数分配都落在大小类别范围内。然而，对于大于最大大小类别的分配，则需要采用不同的机制。对于此类分配，LRMalloc 直接依赖操作系统，其他无锁内存分配器也是如此 [16, 8, 20, 13]。依赖操作系统进行大块分配不会对性能造成显著影响，因为这类分配并不常见。大块分配的处理方式与大小类分配类似，但会跳过线程缓存，并映射一个大小恰好满足分配需求的超块。

这种处理大型分配的方式并不理想，因为要回收此类分配所占用的虚拟地址范围，需要采用不同的机制。鉴于此，我们选择将持久内存分配限制在与大小类别兼容的尺寸范围内。在大多数情况下，这并不成问题，因为无锁数据结构通常会为其内部结构使用小型分配，或者大型分配会一直存在于数据结构的整个生命周期内，因此无需回收；Michael 的无锁哈希表 [14] 就是一个例子。例外情况是使用大型可调整大小数组的无锁哈希映射，因为在调整大小的过程中，它们需要分配一个新数组并回收旧数组。采用这些机制的数据结构相当少见，因为调整大小的过程往往复杂且需要大量同步，从而导致性能下降。因此，我们将这一局限性的解决留待未来工作。

当超块变为 $empty$ 时，我们还需要另一个用于描述符的回收池，原因也在于这一局限性。如果超块未标记为持久[3]，，则会解除超块的映射，并按照原有行为将描述符添加到该池中。如果超块已标记为持久，我们会按照上一节所述重新映射超块，并将描述符添加到新池中。需要新的描述符时，我们会按以下优先顺序尝试获取：（i）新池，该池已关联超块的虚拟地址范围，因此仅适用于用于大小类分配的超块；（ii）包含通用描述符的原始池；最后，（iii）分配一个新的描述符。只有当池不兼容或已耗尽时，我们才会转向优先级列表中的下一项。

5 实验结果

为了评估我们对 OA 方法所做更改的影响，我们将两种 OA 方法实现的结果——一种使用警告位，另一种使用单调全局变量——与原始 OA 方法以及不回收的方案进行比较；在不回收的方案中，内存永远不会被回收、重用或释放。从此处开始，我们将每个线程使用一个警告位的简化 OA 方法称为 $OA\text{-}BIT$，将使用单调全局变量的替代方案称为 $OA\text{-}VER$，并将不回收的替代方案称为 NR。

[3]请注意，只有用于大小类分配的超块才能变为持久状态。

5.1 方法

所用硬件是一台配备 2 个 AMD Opteron(TM) Processor 6274 的机器，每个处理器有 16 个核心，每个核心配备 16KiB 的 L1 缓存，每对核心共享 2MiB 的 L2 缓存，每个 CPU 配备 12MiB 可用的共享 L3 缓存。该机器共有 32GiB DDR3 内存。

我们使用常见的 Michael 的无锁哈希表 [14] 和 Harris-Michael 的无锁链表 [15]对这四种方法进行了基准测试。对于所有基准测试，我们都使用 LRMalloc 作为内存分配器；尽管在简化版本中，它使用新的 $palloc()$ 过程进行分配，但对于原始 OA 和不回收方法，它使用常规的 $malloc()$ 过程。请注意，OA 方法仅在基准测试开始前使用分配器创建内存池，在基准测试过程中不进行任何分配。

基准测试采用了不同的搜索、插入和删除比例，但为了使数据结构的大小在整个基准测试过程中保持不变，我们将插入与删除的比例固定为 1:1。对于链表，我们在运行基准测试前预先插入了 5K 个节点。对于哈希表，我们分别使用了 10K 和 1M 个节点，负载因子为 0.75。结果为 10 次运行的平均值，每次运行 1 秒；我们以吞吐量（每秒操作数）的形式，展示线程数从 1 到 32 时各种组合的结果。

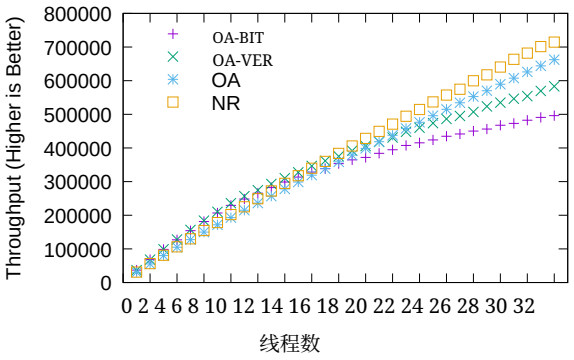
在所有这些实验中，我们没有展示不同内存重映射方法之间的比较，因为我们无法测出它们在性能上的差异（误差范围之外）：将内存保留在分配器中、使用$MADV\ DONTNEED$ 向操作系统提供建议，以及通过共享内存区域进行重映射。

-

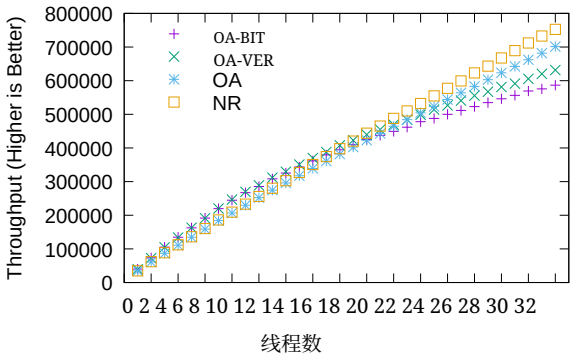
5.2 结果

图 4 展示了基准测试的结果：使用预先插入 5K 个节点的链表。图 4a 展示了仅包含修改操作的情况（ 50% 插入和 50% 删除），图 4b 则展示了操作比例更均衡的情况（50% 搜索、25% 插入和 25 % 删除）。接着，图 5 和图 6 分别展示了使用包含 10K 个节点和 1M 个节点的哈希表进行基准测试的结果。对于这两项基准测试，我们同样测试了仅包含修改操作（50% 插入和 50% 删除）以及操作比例更均衡（50% 搜索、25% 插入和 25% 删除）的情况。

对于仅包含修改操作的链表，OA-VER 方法能够减少警告的触发次数，因此相比 OA-BIT 方法有显著提升。对于包含 50% 搜索操作的链表，由于删除操作较少，这种效果有所减弱；而在使用哈希表的两项基准测试中（图 5 和图 6），由于链明显更短，这种效果几乎可以忽略。

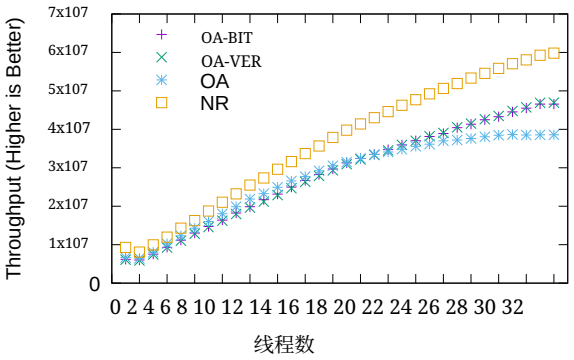


(a) 50% 插入，50% 删除

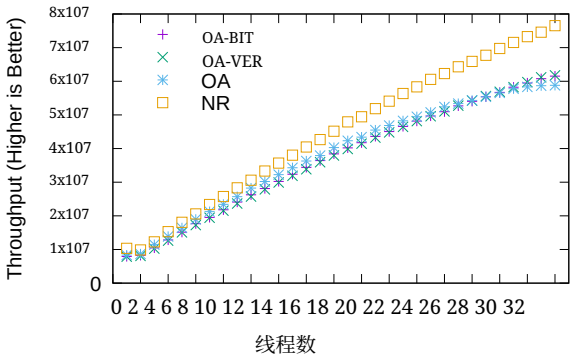


(b) 50% 搜索，25% 插入，25% 删除

图 4：包含 5K 个节点的链表

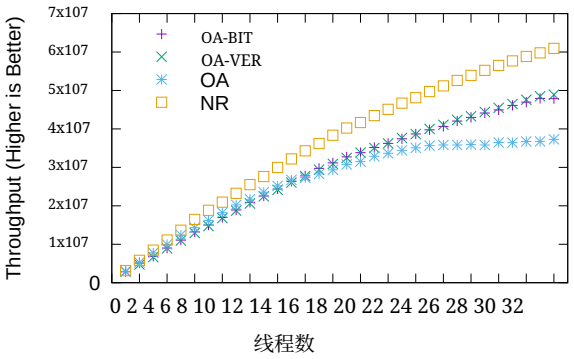


(a) 50% 插入，50% 删除

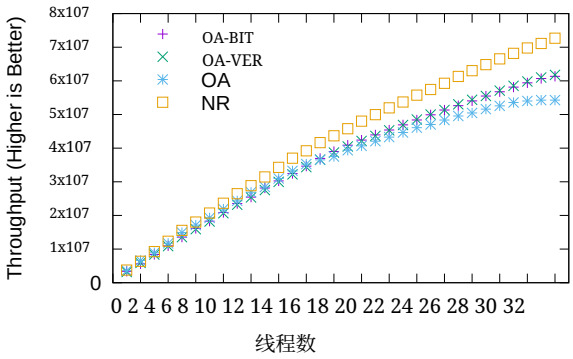


(b) 50% 搜索，25% 插入，25% 删除

图 5：包含 10K 个节点的哈希表



(a) 50% 插入，50% 删除



(b) 50% 搜索，25% 插入，25% 删除

图 6：包含 100 万个节点的哈希表

在线程数较少时，可以看到，针对链表，OA-BIT 和 OA-VER 的表现都优于 OA，甚至优于 NR 方法。这是因为在线程数较少时，f 我们的方法使用的内存较少，大部分内存都保留在较低层级的缓存中。随着线程数增加，f 由于 LRMalloc 的每线程缓存，我们的两种方法开始使用更多内存，因此失去了这一优势。相比之下，OA 方法的内存池大小固定，而 NR 方法则因大量线程之间的同步开销较低而占优。具有不同特性的内存分配器可能会呈现不同的表现。链表是研究系统行为的一个有趣例子，但由于其渐近复杂度特性，在注重性能时并非理想工具。

使用哈希表的基准测试显示，结果在某种程度上出现了反转。总体而言，在线程数较少时，OA 方法的性能略优于我们的方法，但在线程数较高时则明显缺乏可扩展性。在这里，由于吞吐量高得多且内存用量更大，与内存使用量及缓存局部性相比，同步的影响变得更加显著。OA 方法中固定大小的内存池会造成不利影响：随着吞吐量和线程数增加，它需要更多的回收阶段，进而导致同步增加。我们的两种方法都没有这些缺点，因为分配器中的线程缓存和私有暂存列表减少了同步，从而提升了可扩展性。

请注意，本文的主要贡献在于新增了将内存释放回内存分配器/操作系统的能力，以及简化了内存回收方法，而非提升性能和可扩展性，尽管这些提升也值得欢迎。

6 相关工作与未来工作

自 OA 方法提出以来，又有一些方案致力于让 OA 更易于使用，并兼容更多数据结构。其中一个例子是 *Automatic Optimistic Access*（AOA）方法 [4], ，它借助类似垃圾收集器的技术，使数据结构程序员无需调用 Retire。第二个例子是 *Free Access*（FA）方法 [3]，它要求程序员为数据结构函数添加注释；随后通过结合垃圾收集技术与编译器步骤，该方法便能对数据结构应用类似 OA 的内存回收，而无需以规范化形式编写数据结构 [22]。另一个例子是 VBR 方法 [21]，它借助带标签指针和双宽比较并交换（Double-Width Compare-and-Swap, DWCAS），将 OA 扩展至写操作。

我们已经讨论过，对 LRMalloc 的修改如何与 VBR 的乐观双宽度比较并交换兼容，因此我们将 VBR 的简化与适配留待未来工作，以便它也能将内存释放回内存分配器／操作系统。我们还可以使用扩展后的 LRMalloc，以便在 AOA 和 FA 方法中都能动态调整内存池大小（类似垃圾收集器的方式），并通过将内存池释放给内存分配器／操作系统来缩小内存池。链表基准测试的结果表明，这也可能提升性能。

后续工作还包括解除第 4 节讨论的限制，这需要一种能够以无锁方式拆分和合并虚拟地址范围的机制。

7 结论

我们从一个名为 LRMalloc 的无锁通用内存分配器出发，展示了如何扩展它，使其支持 OA 内存回收方法所需的内存模型，从而确保即使释放了内存分配，这些分配的内容仍可读取。我们消除了 OA 方法的主要缺点，同时确保它仍是最高效的内存回收方法之一。在此过程中，我们还简化了 OA 方法的实现，并获得了显示性能提升的结果。

参考资料

[1] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson P. R. , Hoard: 面向多线程应用程序的可扩展内存分配器。 *SIGPLAN Not. 35*, 11（2000 年 11 月），117–128。[2] Braginsky, A., Kogan, A., and Petrank, E. 锚点法：面向非阻塞数据结构的轻量级内存管理。载于 *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures , Systems, Languages, and Applications*（2013），SPAA '13，计算机协会，第 33–42 页。[3] Cohen, N. 每种数据结构都值得采用无锁内存回收。 *Proceedings of the ACM on Programming Languages 2* 2018 143 ,

OOPSLA（），。[4] Cohen, N., and Petrank, E. 面向无锁数据结构的自动内存回收。载于 *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*（2015），OOPSLA 2015，计算机协会，第 260–279 页。[5] Cohen, N., and Petrank, E. 采用乐观访问的无锁数据结构高效内存管理。载于 *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*（2015），SPAA '15，计算机协会，第 254–263 页。

[6] Dechev, D., Pirkelbauer, P., and Stroustrup, B. 理解并有效预防 ABA

问题：基于描述符的无锁设计 .
ternational Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (2010), IEEE, 第 185–192 页。

[7] Dice, D., Herlihy, M., and Kogan, A. 面向高并发数据结构的快速非侵入式内存回收。载于 *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (2016), ISMM 2016, 计算机协会, 第 36–45 页。

[8] Gidenstam, A., Papatriantafilou, M., and Tsigas, P. Nbmalloc：以无锁方式分配内存。 *Algorithmica 58* 2 2010 304 338 ,( ), – .

[9] Herlihy, M., Luchangco, V., Martin, P., and Moir, M. 面向动态大小数据结构的非阻塞内存管理支持。 *ACM Trans. Comput.*, （可能）, –。 *Syst. 23* 2 2005 146 196

[10] Herlihy, M., and Shavit, N. 进步的本质 . 载于 *Principles of Distributed Systems* （2011）, 施普林格, 第 313–328 页。

[11] Lee, S., Johnson, T., and Raman, E. tcsmalloc 的反馈导向优化。载于 *Proceedings of the Workshop on Memory Systems Performance and Correctness* （2014年）, MSPC '14, 计算机协会。

[12] Leite, R., and Rocha, R. LRMalloc：一种现代且具有竞争力的无锁动态内存分配器。载于 *High Performance Computing for Computational Science – VECPAR 2018* （2019年）, 施普林格国际出版公司, 第 230–243 页。

[13] Li, T., Yao, Y., Tang, W., Zhu, F., and Lin, Z. 一种适用于 PDE 应用的高效多线程内存分配器。 *Simulation Modelling Practice and Theory 100* (2020), 102067。

[14] Michael, M. M. 高性能动态无锁哈希表和基于链表的集合。载于 *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* （2002）, SPAA '02, 计算机协会, 第 73–82 页。

[15] Michael, M. M. 危险指针：无锁对象的安全内存回收。 *Transactions on Parallel and Distributed Systems 15* 6 2004 491 504 , （）, –。

[16] Michael, M. M. 可扩展的无锁动态内存分配。 *SIGPLAN Not. 39*, 6（2004 年 6 月）, 35–46。

[17] Moreno, A., and Rocha, R. 关于无锁哈希树设计中内存回收的研究。 *Journal of Parallel and Distributed Computing 155* 2021 1 13 ( ), – 。

[18] Peter, S. 并行数据结构能否依赖自动内存管理器？载于 *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (2012), MSPC '12, 计算机协会, 第 1 页。

[19] Ramalhete, P., and Correia, A. 简短公告：危险时代——非阻塞内存回收。载于 *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (2017), SPAA '17, 计算机协会, 第 367–369 页。

[20] Seo, S., Kim, J., and Maeng, S. Smalloc：一种适用于众核处理器的无锁且基本无需同步的动态内存分配器。发表于 *2011 International Conference on Parallel Architectures and Compilation Techniques* (2011), IEEE, 第 253–263 页。

[21] Sheffi, G., Herlihy, M., and Petrank, E. VMC：基于版本的回收。载于 *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (2021年), SPAA '21, 计算机协会, 第 443–445 页。

[22] Timnat, S., and Petrank, E. 无锁数据结构的实用无等待模拟。载于 *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014年), PPoPP '14, 计算机协会, 第 357–368 页。

[23] Wen, H., Izraelevitz, J., Cai, W., Beadle, H. A., and Scott , M. L. 基于区间的内存回收。载于 *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*(2018年), PPoPP '18, 计算机协会, 第 1–13 页。

[24] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles D. , 动态存储分配：综述与批判性评述。载于 *Memory Management*（1995年）, H. G. Baler 编, 施普林格, 第 1–116 页。