LLFree：可扩展且可选持久化的页面帧分配

Lars Wrenger、Florian Rommel 和 Alexander Halbuer，汉诺威莱布尼茨大学；Christian Dietrich，汉堡工业大学；

Daniel Lohmann，汉诺威莱布尼茨大学

https://www.usenix.org/conference/atc23/presentation/wrenger

本文收录于 2023 年 USENIX 年度技术会议论文集。

2023 年 7 月 10 日至 12 日 • 美国马萨诸塞州波士顿

978-1-939133-35-9

2023 年 USENIX 年度技术会议论文集的开放获取由以下机构赞助：

LLFREE：可扩展且可选持久化的页帧分配

拉尔斯·伦格
汉诺威莱布尼茨大学

弗洛里安·隆美尔
汉诺威莱布尼茨大学

亚历山大·哈尔布尔
汉诺威莱布尼茨大学

克里斯蒂安·迪特里希
汉堡工业大学

丹尼尔·洛曼
汉诺威莱布尼茨大学

摘要

在操作系统内存管理子系统中，页帧分配器是最基本的组件。它管理物理内存帧，这些帧是填充页表树所必需的。尽管

异构、非易失性和大容量内存的出现极大地改变了内存层次结构，但我们仍然

使用 20 世纪 60 年代的开创性方法管理物理内存。

本文认为，现在是时候重新审视页帧分配器的设计了。我们证明，Linux 帧分配器不仅在多核系统上扩展性差，而且还具有很高的内存开销，容易出现巨大的帧碎片，并且使用分散的数据结构，这阻碍了它作为持久内存分配器的应用。我们提出了 LLFREE，这是一种新的无锁无日志分配器设计，它具有良好的扩展性、较小的内存占用，并且可以轻松应用于非易失性内存。LLFREE 使用缓存友好的数据结构，并表现出抗碎片行为，而不会引入额外的性能开销。与 Linux 帧分配器相比，LLFREE 将并发 4 KiB 内存分配的分配时间最多缩短 88%，将 2 MiB 内存分配的分配时间最多缩短 98%。对于内存压缩，LLFREE 将所需的页面移动次数减少了 64%。

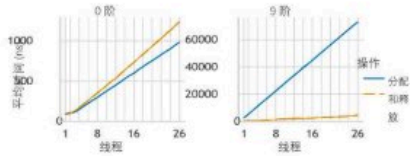

图 1：Linux 帧分配器在并发分配 4 KiB（0 阶）和 2 MiB（9 阶）巨型帧时的性能。

1 引言

在任何虚拟内存子系统中，物理内存的分配都是至关重要的基础原语。传统上，操作系统以内存管理单元 (MMU) 规定大小的页帧形式分配物理内存，并使用简单的空闲列表 [42, 50]（Windows、Darwin）或专门的伙伴分配器 [28]（Linux、FreeBSD）来管理多种页帧大小。然而，近期的硬件发展趋势（例如，高核心数和非易失性内存 (NVRAM)）对这些页帧分配器设计提出了挑战。

一个显著的趋势是快速[49]字节寻址非易失性随机存取存储器（NVRAM）的出现，例如英特尔 Optane DIMM。受其持久性特性的启发，人们提出并评估了新的编程模型 [5, 39, 45, 52]、文件系统 [10, 40, 48] 和容错数据结构 [8, 13, 46]。尽管英特尔宣布 [25] 将逐步停止其 Optane 业务，这将使未来几年 NVRAM 的获取更加困难，但已有研究表明，低成本、高容量的 NVRAM 是可行的，并且具有巨大的潜力 [1, 23, 31]。此外，多位研究人员意识到持久性和可扩展性是深度纠缠的属性[26, 29]，它们既受益于无锁算法[8, 16, 46]，也受益于对不一致中间状态的建设性避免。

随着众多核心竞争资源，大容量 NVRAM 模块引发了一个问题：操作系统如何分配可用于持久化数据的可用内存？分配的成本是多少？又有哪些保证？例如，对于数据库而言，当前的虚拟内存子系统会对其设计 [12, 35]、查询处理速度 [14] 和缓冲区管理 [12, 32] 产生重大影响。因此，我们认为现在是时候重新审视整个虚拟内存堆栈了，首先从底层——帧分配器开始。

首先，我们研究了 Linux 帧分配器 [18] 及其底层伙伴系统 [28] 是否仍然满足 TOS 要求，不仅要考虑从 NVRAM 安全地分配帧，还要考虑 DRAM 的可扩展管理。图 1 显示了批量分配的多核可扩展性。当所有 26 个核心并行分配时，4 KiB 的分配速度降低了十倍（94 ns→984 ns），而 2 MiB 的分配速度则降低了十倍。

甚至慢了 27 倍！这种糟糕的可扩展性影响了许多多核和内存密集型工作负载 [7]。根本原因是分配器状态分散以及全局锁的使用，这两者对于容错 NVRAM 适配来说也都是问题 [16, 17]。

关于本文

我们提出了 LLFREE，一种持久的、无锁无日志的页面帧分配器，它：(1) 专注于内存管理单元 (MMU) 特定的内存大小；(2) 通过减少内存共享，在多核系统上具有良好的扩展性；(3) 由于其元数据量少，因此内存效率高；(4) 具有自动大帧碎片整理功能；(5) 始终处于一致状态，因此非常适合持久内存。本文的主要贡献如下：

- 我们探讨了 Linux 伙伴分配器和更简单的基于列表的帧分配器的弱点。
- 我们推导出以硬件为中心的锁和无日志物理内存管理的设计原则。
- 借助 LLFREE，我们提供了一个适用于易失性和非易失性内存的页帧分配器。
- 我们将 Linux 伙伴分配器替换为 LLFREE，并进行全面评估，从性能、可扩展性、空间开销、碎片行为和崩溃一致性等方面比较这两个分配器。

2 问题分析：Linux 帧分配器

页面帧分配器必须提供物理内存帧，这些帧具有 MMU 特定的粒度，自然对齐，并用于设置虚拟地址空间。本文将使用 AMD64 MMU 及其帧大小（4 KiB、2 MiB、1 GiB），我们称之为自然（分配）大小。但是，该通用设计可以适应其他页面大小。对于 4 KiB 的数据，我们分别称之为基本帧；对于 2 MiB 的数据，我们称之为巨型帧；对于 1 GiB 的数据，我们称之为超级帧。虽然一些内核（例如 Windows 和 Darwin）使用简单的空闲列表 [42, 50]，但 Linux（以及 FreeBSD）使用伙伴分配系统 [18, 28]。

Linux 伙伴分配器 伙伴分配器通过仅允许 $2^o \times P$ 形式的分配大小来避免外部碎片，其中 P 是最小大小，o 是分配顺序。对于每个顺序，伙伴分配系统维护一个自然对齐的空闲块桶。如果分配操作命中空桶 o，则会请求桶 o + 1 中的一个块，并将其分成两个伙伴块，这两个伙伴块的起始地址仅相差一位。一个伙伴块被返回，另一个被放入顺序为 o 的桶中。释放操作会尝试递归地将该块与已释放的伙伴块合并，然后再将该块放入桶中。为了加快合并速度，存储桶通常实现为双向链表，并使用一个单比特标志来跟踪哪些块可用于合并。

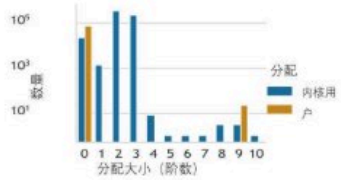

图 2：系统启动期间请求的分配大小以及 120 秒 memcached+memtier 基准测试。

Linux 为每个内存区域（例如，每个 NUMA 节点）使用一个伙伴分配器，支持 $o \in \{0,.., 10\}$ 的顺序，并使用基本帧大小作为 P。因此，在 AMD64 架构上，支持的大小介于 4 KiB 和 4 MiB 之间。与通用伙伴分配器不同，Linux 不会将列表指针存储在空闲内存中，因为这需要映射所有内存，而并非所有架构都支持映射。相反，它使用 struct page 结构体来存储元数据。[1]此外，每个区域分配器都受到自旋锁的保护，该自旋锁会串行化所有拆分和合并操作。为了减少对这些锁的争用，Linux 还为最常用的指令使用了每个 CPU 的缓存。

问题 1：职责混合 Linux 的帧分配器不仅用于分配硬件大小的页帧，还用于分配各种阶数的连续物理内存范围。虽然在 I/OMMU 广泛应用之前，这对于分配直接内存访问 (DMA) 缓冲区是必要的，但现在技术上已不再需要这样做。然而，Linux 开发人员仍然使用这些非自然大小来分配更大的内核对象（例如，栈）。为了更好地理解这一点，我们记录了启动期间以及随后的 memcached 基准测试中请求的内存大小（见图 2）：我们发现，用户空间内存仅被请求为自然阶数 0 和 9，而内核则大多请求非自然阶数。为内核对象分配连续的帧块可能仍然有利于节省 TLB 条目（Linux 使用巨型帧将所有物理内存映射到内核空间）。然而，通过混合帧分配和内核对象分配，分配器必须通过其接口提供所有阶数，这会导致一系列次要问题。

问题 2：合并成本 由于基本帧和巨型帧之间存在九个伙伴指令，因此两者之间的转换成本很高：在最坏的情况下，从 512 个 4 KiB 帧开始，需要 511 次伙伴合并操作才能形成一个 2 MiB 帧。对于每次合并和加锁操作，我们必须操作五个缓存行中的列表指针；其中四个位于通常尚未进入缓存的结构页中。

问题 3：可扩展性 此外，正如我们在图 1 中看到的，如果重新……，这些已经成本高昂的操作的可扩展性会很差。

[1]借助结构页（struct page），Linux 拥有一个每帧的信息存储，可供不同的子系统使用和重新利用。在 AMD64 架构上，它的大小为 64 字节。

多个线程同时请求。这是因为上述每个区域锁存在争用，Linux 试图通过维护每个 CPU 的缓存来缓解这个问题。每个 CPU 缓存都维护一个空闲块列表，当内存压力过大或缓存超过水位线时，这些空闲块会被释放。虽然很长一段时间以来，只有 0 阶分配会被缓存，但 Linux 5.13 将缓存扩展到了 1-3 阶和 9 阶（2 MiB）。然而，分配密集型工作负载很容易使这些 CPU 缓存不堪重负。此外，它们还会加剧碎片化和复杂性。

问题 4：大帧碎片 尽管伙伴分配系统优先使用最小匹配桶，但它在处理大页碎片方面存在问题：例如，如果一个原本空闲的 2 MiB 帧中有一个 4 KiB 的片段正在使用，那么其余 511 个基本帧将以 9 个不同大小的块（4 KiB 到 1 MiB）的形式存在。由于桶是无序集合，并且分配器没有"几乎满的大页"的概念，因此这些块与其他任何大帧中的任何其他块一样，都有相同的机会被分配。因此，伙伴分配系统并没有专门针对最小化大帧碎片而设计。我们将在 5.6 节中进一步讨论这个问题。

每个 CPU 的缓存会加剧碎片，因为它们会延迟合并操作。即使一个 2 MiB 帧中最后一个缺失的 4 KiB 帧已被释放，它也可能驻留在每个 CPU 的缓存中，而要完成大帧的合并，必须先刷新该缓存。此外，由于每个 CPU 的缓存会隐藏内存，使其无法被伙伴分配器访问，因此我们无法采用桶内启发式方法来提高大帧合并的可能性（例如，追加到桶列表的末尾）。相反，一个最近释放的、足以完成 2 MiB 帧的内存块更有可能被再次分配。

为了减少大页碎片，Linux 自 2015 年以来支持"高原子页块"，以隔离更大的块并防止它们被小分配碎片化。此外，Linux 还采用了主动碎片整理（内存压缩），其中后台任务遍历内存区域并将页面移动到开头，并在末尾清除较大的块。然而，这两种方法都会增加复杂性。

问题 5：持久分配 鉴于其当前结构，Linux 帧分配器不适合持久分配。对于持久性 NVRAM 区域，分配器必须能够在断电后恢复其状态，以确保所需元数据的持久性。对于复杂算法（例如受锁保护的递归帧合并）、分布式状态（例如双向链表）或冗余（例如每个 CPU 的缓存）而言，这极具挑战性 [17]。虽然理论上，这些问题都可以通过额外的日志协议来解决 [36, 41, 45]，但这样做会对性能、内存和复杂性造成过大的影响。此外，尽管崩溃通常极其罕见，但每次常规操作都需要付出这种日志开销。因此，我们认为这不是一个现实可行的方案。据我们所知，目前还没有任何持久（页帧）分配器能够达到与其易失性对应分配器相近的分配速度[3, 9, 33, 36, 41, 45, 52]。

问题总结：复杂性 最终，Linux 物理内存分配器面临着复杂性的问题。帧大小和其他分配量之间的冲突（问题 1）促使了伙伴结构的出现，然而，这导致了高昂的合并成本（问题 2）以及基于锁的双向链式遍历，从而阻碍了多核可扩展性（问题 3）。虽然通过增加每个 CPU 的缓存数量可以缓解这个问题，但这（与伙伴系统结合）却加剧了大帧碎片化（问题 4），需要额外的机制，例如高原子页块和内存压缩，从而进一步增加了复杂性。所有这些都导致设计不适合持久分配（问题 5），因为数据和状态的存储是冗余的和分布式的。

所有这些设计决策在当时很可能都是合理的。我们认为，现在是时候重新审视我们系统中最基本的内存管理器——页帧分配器的设计和结构了。

3 LLFREE 页面帧分配器

我们最初将 LLFREE 设计为仅支持自然帧大小（4 KiB、2 MiB）的页帧分配器，目标是实现高可扩展性和对持久分配的适用性。为了与 Linux 集成，我们后来不得不扩展它以支持非自然分配顺序，令我们惊讶的是，这在不影响任何设计目标的情况下成功实现了。下面，我们将描述原始设计，而集成细节将在第 4 节中讨论。

3.1 设计原则

对于我们的分配器，我们规定了三个主要设计原则：

尊重硬件 硬件特性决定了软件实现的结构元素和功能。我们利用MMU定义的帧大小、数据结构中的缓存行粒度以及算法中可用的原子操作。

避免共享 在多CPU系统中，真共享和假共享都是可扩展性的主要瓶颈，在NUMA系统中尤为突出。锁是导致共享的常见原因。我们减少对共享数据结构的访问，并且不使用锁。

谨慎的冗余 冗余信息（例如软件缓存和副本）必须保持同步。当目标是持久内存上的崩溃一致性时，这一点尤其难以实现，因为潜在的崩溃可能会破坏这些冗余数据结构的同步。因此，我们严格限制崩溃恢复所需状态的冗余。

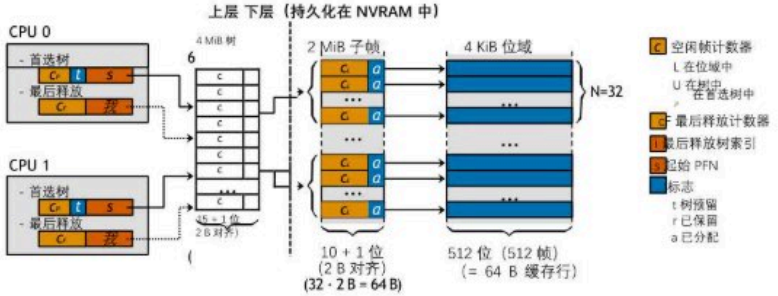

图 3：LLFREE 分配器的架构：树形条目、子条目和位域依次存储在大型数组中。从 PFN 中，我们可以直接提取相应树形条目、子条目、位域以及帧内已分配位的索引。

我们并不声称这些原则具有根本性的创新性：前两个原则在可扩展操作系统内核开发领域已得到广泛认可；它们的优势已被多次报道 [6, 15, 22, 47]。但第三个原则并非如此，它更侧重于实现崩溃一致性而非可扩展性。相反，采用冗余而非限制冗余是内核设计中避免资源共享和提高可扩展性的常用技术（如前一节所述），因此两者之间存在权衡。持久性和可扩展性之间的这种深度纠缠，以及避免中间状态对于在NVRAM上实现崩溃一致性的重要性，已经在数据结构和算法领域有所报道[8, 16, 26, 29, 46]。本文的贡献在于严格地结合并应用这些原则，并在设计一个既能在DRAM上可扩展，又能选择性地在NVRAM上实现崩溃一致性的页帧分配器时，权衡了它们之间的利弊。

3.2 LLFREE：设计概述

图3展示了LLFREE的架构，该架构是针对自然分配顺序0（4 KiB基本帧）和9（2 MiB大帧）设计的。从概念上讲，LLFREE分为两层：底层执行实际的分配操作，上层实现分配策略以避免共享和碎片化。简而言之，底层负责崩溃一致性——它只需要在崩溃一致的NVRAM区域上保持其状态的持久性——而上层则负责可扩展性。

3.2.1 底层 - 分配机制

底层管理基本帧和巨型帧的释放/分配。为此，它为每个巨型帧使用一个表项和一个 512 位的位域（图 3：2 MiB 子节点，4 KiB 位域）来标记空闲 (0) 和已占用 (1) 的基本帧。首先，它原子地递减 c_i（这可以防止争夺最来自这个巨大的帧。空闲基帧的数量也由计数器 c_i 维护。基帧的分配方式如下

后一个空闲帧），然后在位域中搜索 0 位，此操作由特殊的处理器指令支持。巨型帧的分配只需将计数器 c_i从 512 更改为 0，并将已分配的平面标志 a 设置到位域中，所有操作都在一个 16 位比较并交换 (CAS) 操作中完成。位域保持不变，并且在这种情况下全为零，必要的簿记（例如，用于崩溃恢复）在 a 标志中完成。

因此，在大多数情况下，底层分配器只需访问两行缓存即可释放/分配一个基本帧（表项、位域），而释放/分配一个大帧（表项）只需访问一行缓存。[2] 即使当前子树包含的帧不足以进行分配，我们的顺序搜索也与处理器的缓存行预取机制完美契合。

3.2.2 上层 - 分配策略

上层通过使用能够有效减少（伪）共享和大帧碎片的分配策略来提供可扩展性。从技术上讲，它将物理内存管理为一个我们称之为树的块数组（图 3）：每个树根指向一个具有 N 个子节点的底层表，这些子节点又指向 N 个位域，每个位域管理 512 个帧，类似于页表树。我们选择 N = 32，以便子数组（在底层）占用一行缓存；因此，一棵树管理 16384 个基本帧，即 64 MiB。

每个树根包含空闲基帧的数量 c_f（用于分配策略），以及一个保留标志 r（用于每个 CPU 的绑定）。我们早期的基准测试表明，当底层子数组中的条目并发更新时，分配会受到伪共享的影响。因此，为了避免共享，每个 CPU 可以绑定 (r = 1) 一个用于其分配的首选树。

如果首选树中没有剩余的空闲帧，则必须预留一棵新树。预留算法遵循一种搜索启发式方法，通过使用 c_f将树分类为以下三类之一来避免大帧的碎片化：

[2] 上层将发生一次额外的缓存行访问。

已分配 几乎所有帧都被占用 (c_U< 12.5%)。
空闲 几乎所有帧都是空闲的 (c_U> 87.5%)。
部分 介于两者之间的所有情况。

该启发式算法优先选择部分树，而不是空闲树和已分配树。因此，（几乎）空闲树随着时间的推移更有可能完全空闲。但是，对于新的 CPU 优先树，我们仍然优先选择空闲树，而不是已分配树，因为后者分配失败的概率很高，尤其是在处理大型帧时。这是性能和碎片化之间的一种权衡。

确定树是空闲、部分还是已分配的阈值是可配置的。我们的基准测试表明，空闲树的阈值为 12.5%（N = 32 时为 2048 个空闲基帧），已分配树的阈值为 87.5%，足以避免碎片化（第 5.6 节）。实际搜索合适树的顺序如下：

1. 首先，在当前 CPU 树的邻域内搜索部分树或空闲树，邻域定义为位于同一缓存行上的另外 31 个树根。
2. 如果搜索失败，则按顺序（首次适应）搜索整个树数组，以查找部分树。
3. 如果没有找到部分树，则重复搜索空闲树或具有足够空闲页的已分配树。
4. 作为最后的手段，分配器会耗尽（取消预留）其他 CPU 的树并窃取它们。

但请注意，即使是保留新页面帧的慢速路径
树不需要锁，可以完全并行执行，因为预留本身只需要原子地更新条目的预留标志。给定一个 256 GiB 的内存区域，
该算法在最坏情况下会访问 128 个缓存行。CPU 本地树根：尽管使用了每个 CPU 的保留-
，当多个 CPU 并发更新共享同一缓存行的条目的空闲计数器时，树数组的更新仍然可能出现伪共享。这可以通过两种方式解决：要么将 2B 的树条目与缓存行大小（x86 架构上为 64B）对齐，要么将计数器拆分为全局部分和 CPU 局部部分。我们采用了第二种方法以保持较低的内存和缓存开销。在树预留时，CPU 会将预留树的空闲计数器 c_i移动到其 CPU 局部数据 c_P，并将 c_i设置为零。现在，来自该 CPU 的分配和释放操作只会更改局部计数器 c_P。由于树已被保留，其他 CPU 的分配将不再发生，但先前分配的外部释放仍然可能发生。在这种情况下，树数组中的全局计数器 c_U只会递增，从而避免使相应本地条目的缓存行失效。如果本地分配耗尽内存 (c_P= 0)，或者在极少数情况下，通过从另一个 CPU 远程排空已保留的树（这也会清除树的保留），则计数器会进行同步。

除了本地释放计数器 c_P之外，CPU 本地条目还包含树内保留标志 t 和起始 PFN，合并为一个 64 位数据类型。起始 PFN 包含最后一个已分配基帧的编号；它充当最后适应指针，加速子数组的分配（除以 512），并标识已保留的树（除以 512 · N）。在为每个 CPU 保留新的树时，t 标志会被原子地设置，以防止与远程排空或并行保留尝试发生竞争。后者仅在区域太小，无法容纳每个核心一棵树时才会发生。在这种情况下，每个 CPU 的数据会在多个核心之间共享，t 标志会协调新树的分配。[3]

释放时保留：虽然每个 CPU 的树可以防止并发分配的竞争和伪共享，但释放操作必须始终指向相应帧的源树。如果帧的源 CPU 在此期间切换到了新树，或者释放操作是从另一个 CPU 调用的，则此树不是每个 CPU 的树。尤其是在内存密集型工作负载下，释放操作仍然可能受到伪共享的影响。

为了缓解这个问题，LLFREE 提供了一种释放时保留的启发式算法，该算法假设分配和释放工作负载具有局部性：CPU 在连续 F 次释放操作之后将该树保留为其首选树，并预期后续的释放操作也会影响该树（图 3 中的 c_P和 i）。在我们的基准测试中，阈值 F = 4 表现最佳。

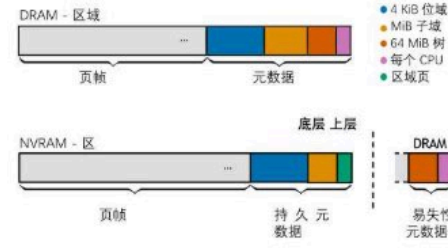

图 4：LLFREE 管理的持久内存和非持久内存中的内存区域布局。

3.2.3 崩溃一致性

LLFREE 可以选择在 NVRAM 区域上提供崩溃一致性。对于崩溃一致性 NVRAM 区域，只需将 allos 底层（参见图 3）存储在持久内存中，外加一个额外的区域页，用于存储魔术标识符、内存区域大小以及启动/关闭标记，以便检测系统在断电后是否需要恢复。顶层始终驻留在 DRAM 中，以减少访问延迟并避免对 NVRAM 进行不必要的写入。它会根据底层信息恢复到 DRAM 中。图 4 描绘了持久内存区域和非持久内存区域的区域布局。

[3]在 Linux-x86 上，这仅适用于 16 MiB DMA 区域，Linux 仍然提供该区域以兼容传统的 16 位 ISA 设备。

LLFREE 设计的关键在于，所有事务都通过对单个缓存行的一次原子更改来权威地完成。对于基本帧，这是对相应位域的原子更改；而对于大帧，则通过对子条目中 a 和 c_i的原子更改来实现。这确保了内存一致性，因为每次权威更改对所有缓存一致的内核都可见，并且所有派生信息（例如计数器值）都通过原子交换操作进行更改，这些操作不受重排序的影响。

单缓存行特性使得在所有提供持久粒度[38]的系统上实现持久一致性变得容易，这些系统在操作结束时至少将一行缓存以原子方式写入NVRAM，这被认为是NVRAM硬件的最低标准[10, 38]。在恢复情况下，会检查子数组中的每个条目是否具有某个标志：如果该标志已设置，则子条目是权威信息——该条目是一个$c_i = 0$的大页（位域全为零）。否则，该条目描述一组基页——位域是权威信息，用于恢复c_i的值。然后可以从子数组恢复上层状态。

虽然在释放/分配期间发生的崩溃永远不会导致分配器状态无法恢复，但它可能会导致帧丢失。如果在分配过程中某个位已被设置，但帧尚未到达调用者，或者释放操作已被调用但该位尚未被清除，则会发生这种情况。理论上，这种影响的最坏情况是并行操作的最大次数，即 CPU 的数量。可以通过两阶段释放/释放协议来缓解这种情况，但这也会改变页面分配器的接口。崩溃是罕见事件——在页面帧分配的关键阶段发生崩溃则更为罕见（受影响的代码序列仅占用几个时钟周期）。因此，在系统生命周期内实际发生帧丢失的概率极低，即使发生这种情况，其成本也是可以接受的。

4 实现

我们将 LLFREE 实现为一个 Rust 模块，并将其集成到 Linux 内核中（并对其进行了扩展），然后用 LLFREE 替换了原有的 Linux buddy 分配器。其分配和释放算法（在前一节中讨论过）可以在附录中找到（图 14）。

4.1 方法

随着 Linux 社区开始采用 Rust 语言，我们借此机会探索了它是否适合用于性能至关重要的底层内核模块。与 C 相比，Rust 的内存管理更加严格（即更安全），并避免了未定义行为。我们相信，这些特性可以防止整类内存错误，从而简化分配器的开发。

模块化的 LLFREE 实现包含一个测试环境，该环境在用户空间的虚拟内存映射上初始化分配器，用于单元测试和基准测试。这使得我们能够及早分析并发现性能瓶颈。除了标准的单元测试之外，我们还针对原子操作的可能顺序开发了特定的竞争条件测试，这些测试在发现一些设计和逻辑错误方面非常有效。

遵循这种方法，我们能够快速实现并比较分配器上层和下层的策略。最终的实现包含 2199 行安全的 Rust 代码（其中 25 行不安全代码用于初始化和地址转换）和 1318 行单元测试。Linux 伙伴分配器是用 C 语言编写的，主要包含在 page_alloc.c 文件中（不包括内存回收和内存压缩），该文件本身就有 6060 行代码——而且没有任何测试。然而，伙伴分配器与其他内存子系统组件紧密耦合，因此很难估计它对该源代码库的实际贡献。

4.2 替换 Linux Buddy 分配器

我们修改了 Linux 系统，使其能够使用我们开发的 LLFREE 内存分配器启动。集成过程需要对 LLFREE 进行一些修改（例如支持非自然顺序），但更重要的是需要对 Linux 本身进行修改，因为伙伴内存分配器的实现与 Linux 紧密耦合。尽管如此，该系统在日常工作负载下运行稳定。

4.2.1 LLFREE 变更

LLFREE 专为硬件定义的自然顺序而设计。然而，Linux 需要支持所有最高到 10 的顺序，我们按如下方式实现：顺序 1 到 6（2 到 64 帧）的分配方式与顺序 0 类似。分配过程会在位域中搜索足够大的、对齐的零集合，并通过一次 64 位 CAS 操作翻转这些位来分配内存。

顺序 7 和 8（128 和 256 帧）使用乐观的无锁算法进行分配，该算法使用 2 到 4 个原子操作。如果其中一个操作由于竞争条件而失败，则其他操作会被安全地回滚，搜索继续进行。然而，这些顺序很少被分配（参见图 2），只有在分配过程中树被窃取或我们在多个 CPU 之间显式共享树时才会发生争用。因此，实际冲突很少发生。[4]

10 阶分配的实现方式是同时分配两个对齐的子条目，类似于 9 阶分配。由于这些子条目只有 16 位（对齐后），因此只需一次 32 位 CAS 操作即可完成。如果由于树碎片化而导致更高阶分配失败，则会保留另一棵树。在这种情况下，分配器会先搜索未碎片化的空闲树，然后再回退到部分树。

[4]这也打破了我们在 3.2.3 节中对 7 阶和 8 阶持久性一致性的假设。但是，我们现在可以安全地忽略这一点，因为非自然阶仅由内核使用，而内核目前不使用持久性。

4.2.2 Linux 变更

在启动过程中，LLFREE 分配器的数据结构由早期启动内存块分配器分配并初始化。与伙伴分配器类似，我们为每个内存区域创建一个 LLFREE 实例，并将指向其数据的指针直接存储在
结构区域。代码库中的大部分后续更改是为了在我们的实现激活时（通过 Kconfig 选项）有条件地禁用并替换伙伴分配器、其每个 CPU 的缓存、区域锁和高原子页面块。LLFREE 模块（LLFREE 分配器的轻量级封装）总共增加了 942 行代码。
在该模块之外，我们更改了 415 行代码，其中 page_alloc.c 文件中就有 296 行。

大多数功能（包括页面回收）都很容易适配到新的分配器。然而，一些直接使用伙伴分配器内部数据结构的高级服务（例如其空闲列表）在不完全重写的情况下无法适配。因此，我们在基准测试中禁用了两个分配器中的这些服务。首先，这包括决定是否执行主动内存压缩的内存碎片启发式算法。该启发式算法利用了伙伴分配器空闲列表的内部计数器。然而，主动内存压缩的需求是一种特殊情况，仅当分配器高度碎片化且几乎没有剩余大页时才会触发。这种情况在我们的基准测试中并未发生。相反，我们在第 5.6 节中比较了碎片化和压缩的成本。其次，我们禁用了内存溢出 (OOM) 处理程序，因为它的检查直接依赖于伙伴分配器的内部空闲列表。考虑到 OOM 处理程序的复杂性，我们认为对其进行重新设计超出了本文的范围。我们的基准测试不会触发两个分配器的 OOM 事件。

为了使 Linux 中更高阶的分配速度具有竞争力，我们不得不实现两种变通方法：第一种方法减少了对 struct page 的写入访问次数。
条目。为了辅助内核内部调试（例如，检测重复释放），Linux 会重新初始化所有支持高阶分配的结构体页面的标志，这是一种代价高昂且大多不必要的开销。Linux 还区分标准分配（可以分部分释放）和复合分配（只能一次性释放）。对于复合分配，除第一个结构体页面外，其余页面都被标记为尾页。这种分散且冗余的复合帧编码方式代价高昂，尤其对于大型帧而言。每次释放/释放操作都需要修改所有 512 个条目（即 512 个缓存行），这会严重影响整体性能，使得难以区分和比较两种分配器的分配速度。因此，我们对标准分配进行了基准测试，并禁用了标志重新初始化。后者没有产生任何可观察到的后果。

第二个瓶颈，尤其影响了 LLFREE，是 vmstat 空闲帧计数器的更新，该计数器提供了可用空闲内存的估计值。对于逃逸出 CPU 缓存的释放/分配操作，此计数器会进行更新。为了减少拥塞，每个 CPU 都有一个本地计数器，用于吸收达到一定阈值的更新。然而，其当前值 125 对于大型帧来说太小了。我们将阈值提高了 1024（相当于 9 个 CPU 缓存的大小）。由于 LLFREE 分配器不使用这些缓存，因此全局计数器的精度与使用原始值的伙伴分配器一样高，因为后者可能在其 CPU 缓存中隐藏了同样多的帧。

5 评估

在我们的评估中，我们展示了 LLFREE 在不同的分配模式和大小下都具有良好的可扩展性。我们还研究了碎片化行为，量化了内存开销，并研究了 NVRAM 情况下的崩溃恢复。

5.1 评估设置和基准测试

我们的测试系统是一台戴尔 PowerEdge R750 服务器，配备两颗第三代英特尔® 至强® 金牌 5320 CPU（2×26 个物理核心，主频 2.20 GHz）。每个 NUMA 节点都配备四个 32 GiB DRAM DIMM（总计：256 GiB DRAM）和四个 128 GiB Optane Gen 2 DIMM（总计：1 TiB NVRAM）。我们假设第三代至强黄金架构[24]提供的 eADR持久性保证（内存一致性1→持久性一致性），并在实现中省略显式持久缓存刷新（clwb），因为这样可以更直接地比较DRAM和NVRAM的分配速度。[5]为了获得稳定的结果，我们禁用了主动内存清零（最近引入的可选强化功能）和超线程，这会产生类似的总体性能特征，唯一的区别是物理核心之间的内存共享成本高于逻辑核心之间的内存共享成本。我们在修改后的Linux 6.0内核上执行基准测试，分别在启用和禁用LLFREE的情况下进行。由于Linux会为每个内存区域（例如NUMA-1-DRAM）实例化互不影响的分配器，因此我们使用单个NUMA节点分配器进行隔离测试。

由于分配器的性能与工作负载密切相关，我们创建了三个涵盖各种分配模式的综合基准测试：（1）对于批量基准测试，所有核心同时分配可用内存的一半，然后立即将其释放；此过程重复进行。（2）随机基准测试会分配所有内存（类似于批量基准测试），并以随机顺序释放帧；我们只测量释放操作。(3) 对于重复基准测试，每个核心尽可能快地分配和释放单个帧。重复测试特意设计为 Linux 伙伴分配器的最佳情况，因为它最大化了每个 CPU 本地缓存的预期收益。然而，这并非最现实的情况：常见的情况是延迟分配（由于按需分页）以及批量/随机释放（在程序终止时）。

[5]请注意，LLFREE 也可以在持久性较弱的情况下工作（第 3.2.3 节）。

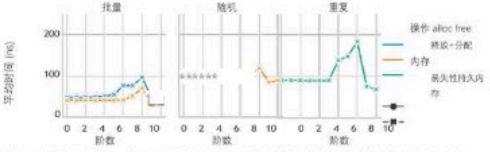

图 5：独立 LLFREE 在 DRAM 和 NVRAM 上的单阶分配时间（8 核，128 GiB）

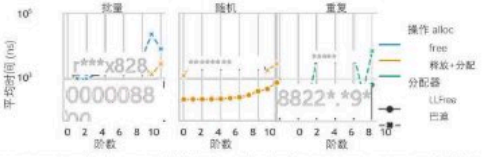

图 6：Linux 集成 LLFREE 的单阶分配时间（8 核，128 GiB DRAM，对数刻度）

由于单个操作执行速度很快，因此逐个操作的时间测量会扭曲结果。因此，我们测量所有操作的时间并将其除以操作数。由于 LLFREE 受益于自由局部性，我们预计随机基准测试将特别具有挑战性。

5.2 分配大小

首先，我们考察不同请求大小（4 KiB – 4 MiB）的分配速度。为此，我们使用 8 个 CPU 绑定线程来管理 128 GiB 的 DRAM 或 DAX 映射的 Optane 内存。基准测试既在我们的用户空间基准测试环境中执行，以测量 LLFREE 的独立性能，也在修改后的 Linux 内核模块中执行。

图 5 显示了用户空间基准测试中每个操作的平均时间。对于批量和重复操作，单次操作耗时不到 100 纳秒，而阶数为 8（1 MiB）的操作耗时最长，因为它距离下一个较低的自然阶数最远。由于随机数的缓存未命中和失效行为，一次释放操作可能需要长达 120 纳秒。尽管已知 Optane 的随机访问延迟约为 DRAM 的两倍 [49]，但由于大多数更新操作都保留在 L3 缓存中，因此 DRAM 和 NVRAM 的分配器性能非常相似。

在这些用户空间结果之后，我们替换了 Linux buddy 使用 LLFREE 进行定量的现场比较。这种集成引入了更高的管理开销（例如，更新结构页），与之前的用户空间基准测试相比，这会导致更多的缓存未命中。由于 Linux 的分配器不具备崩溃一致性，我们现在只关注 DRAM 的性能。同样，第一个 NUMA 节点上的八个核心并行执行相应的基准测试。

图 6 显示 LLFREE 大约高一个数量级在批量和随机基准测试中，其速度比原始 Linux 分配器更快。对于重复基准测试（即重复重新分配单个帧），每个 CPU 的缓存（0-3 阶、9 阶）使得伙伴分配器比 LLFREE 更快（例如，0 阶缓存：112 纳秒 vs. 183 纳秒）。这不仅源于缓存本身，还源于某些统计信息（例如，vmstat 计数器、NUMA 命中/未命中率）未更新。

如果缓存服务于帧请求。然而，对于未被每个 CPU 缓存覆盖的大小，LLFREE 在重复基准测试中的速度大约快 100 倍。

然而，每个 CPU 缓存并非对所有工作负载都有益：在批量基准测试中，我们发现对于 9 阶分配，伙伴分配器比 8 阶和 10 阶分配慢大约十倍。深入分析揭示了问题所在：由于 9 阶缓存的容量只有 2，因此每隔一帧就会调用一次缓存填充操作。这种填充操作会将多个帧（在本例中为两个）的分配批量处理。

将其合并为一个临界区，减少了获取和释放伙伴锁的操作次数。然而，该临界区还包含对已分配帧的所有结构页的检查，这对于高阶分配来说尤其耗时。因此，锁的持有时间更长，与其他在释放锁后执行这些检查的无缓存指令相比，锁争用加剧。

对比图 5 和图 6，可以看出 Linux 的分配路径仍有改进空间。由于其他开销与覆盖的 4 KiB 帧（用于更新结构页）的数量呈线性关系，我们目前无法充分发挥 LLFREE 在 2 MiB 帧上的性能。例如，虽然内核中 4 KiB 的随机释放速度相同，但释放一个大帧所需的时间却要长 4.48 倍。

5.3 多核可扩展性

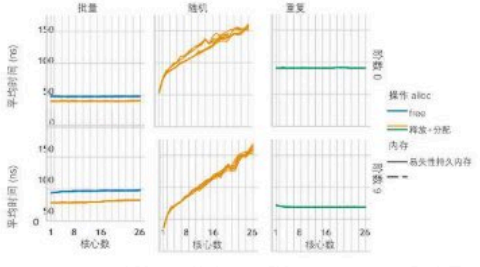

图 7：LLFREE 在易失性和持久性内存中，指令 0 和 9 以及 128 GiB 内存的平均核心数耗时

为了评估多核可扩展性，我们重点关注两种自然帧大小，并将请求核心数从 1 扩展到 26。图 7 再次展示了 LLFREE 的原始性能（以 表示）。

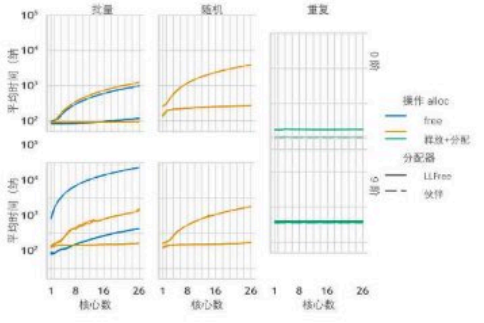

图 8：Linux 内核中 0 阶和 9 阶以及 128 GiB 内存的平均时间（对数刻度）

用户空间）在 DRAM 和 NVRAM 上，而图 8（对数刻度！）显示了内核在 DRAM 上的性能。

对于批量和重复用户空间基准测试（图 7），我们看到 LLFREE 的运行时间几乎保持不变，与核心数无关，内存类型对性能的影响微乎其微。这里，LLFREE 的分配和空闲保留系统避免了大部分共享。只有在随机模式下，缓存失效和子计数器更新冲突更为频繁，我们才能观察到工作线程数量的显著影响。然而，即使有 26 个工作线程并行请求帧，DRAM 的分配/释放操作（无论顺序如何）耗时也低于 170 纳秒。

如图 8 所示，对于自然大小的缓存，Linux 的性能很大程度上受到其每个 CPU 缓存的影响。对于重复基准测试（每个 CPU 缓存的最佳情况），我们发现 LLFREE 比 Linux 慢了高达 65.18%（26 个核心，4 KiB）。然而，对于超出每个 CPU 缓存容量的批量和随机释放，Linux 分配器在核心数增加时性能急剧下降，因为此时内存直接从伙伴系统请求。虽然 4 KiB 的单核性能仍然几乎相同，但在 26 个核心的情况下，Linux 的批量分配/随机释放时间分别是 LLFREE 的 7.3/13.5 倍。对于 26 个核心请求 2 MiB 帧的情况，这一倍数变为 52.5/9.6 倍。

5.4 基于列表的分配器

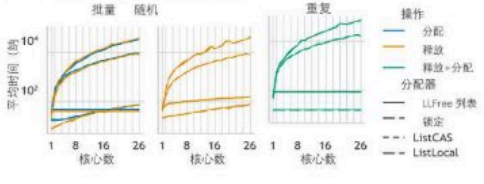

图 9：在 128 GiB DRAM 上，链表和 LLFREE 分配器在每个核心上的平均速度（0 阶）。

除了伙伴系统之外，简单的空闲列表是通用操作系统（Windows [50]、Darwin [42]）和研究操作系统（Twizzler [5]）中页面帧分配器的常见设计。与 Linux 类似，它们通过额外的核心局部列表来缓解全局结构上的锁争用。为了将这些分配器概念与 LLFREE 进行比较，我们构建了三个基于列表的分配器原型实现，并评估了它们的可扩展性。（1）ListLocked 分配器，由基于锁的共享单链表组成（Windows、Darwin）；（2）ListCAS 分配器，它使用 LIFO 无锁链表 [44]（据称优于锁）；

(3) ListLocal 分配器是一种理论上的分配器，它只维护每个核心的列表，不需要任何保护。

永远不会耗尽（理想情况）。所有列表分配器都将其下一个指针存储在一个 64 字节对齐的数组中，类似于 Linux 的 struct page array。

图 9 以对数坐标显示了此次比较的结果。正如预期的那样，由于锁竞争程度高，加锁变体的性能最差。然而，虽然用原子操作替换锁可以将性能提升 81%（26 核，随机模式），但我们发现这仍然没有解决根本的扩展性问题；竞争基本上只是从锁转移到了包含列表头指针的缓存行。

出乎意料的是，LLFREE 甚至在 16 核及以上的情况下优于理想的 ListLocal 变体，后者甚至不适合作为全局帧分配器，但在 26 核上进行批量分配时耗时却高出 36%。这是由链表的状态分散造成的：几乎每次分配和释放操作都会触及至少一条新的缓存行——相比之下，LLFREE 的缓存友好型结构在最佳情况下，512 次分配操作都只使用相同的三条缓存行。在随机基准测试中，LLFREE 还会因为非本地释放操作而出现缓存未命中（这是 LLFREE 的最坏情况）。

5.5 分配器状态分散

为了比较分配器的时空成本，我们提出了状态分散度指标——一个量化指标，表示用于元数据存储的字节数。直观地说，状态分散度是指完整枚举内部状态所需的字节数。然而，必须理解的是，状态分散度并不能直接转化为内存开销，因为分配器通常会重新利用空闲内存或重载其他共享数据结构（例如，结构体）。

page）用于存储其元数据，因此，纯粹的内存开销就变得不再那么重要了。然而，状态分散性高的分配器在其运行期间很可能会导致更多的缓存未命中，从而影响运行时效率。当然，这不仅仅取决于状态的大小。

以及其空间分布。

在表 1 中，我们分析了 LLFREE 的状态分布以及 Linux 分配器的不同组件。为了更好地理解这些数字，我们还展示了分散性如何体现在

分配器	每个区域	每个 CPU	每 1 GiB	1 个区域 128 GiB 52 个 CPU	完全扫描：访问的缓存行
LLFree				**4.1 MiB**	**67 754**
4 KiB 位域			32.0 KiB	4.0 MiB	65 536
2 MiB 计数器			1.0 KiB	128.0 KiB	2 048
64 MiB 树		128 B	32 B	10.5 KiB	168
全局	128 B			128 B	2
Linux Buddy 分配器				**516.0 MiB**	**33 555 169**
空闲列表 已分配标志	988 B		4.0 MiB	512.0 MiB	33 554 448
页块位			32.0 KiB 256 B	4.0 MiB 32.0 KiB	（在上述范围内）
每个 CPU 的缓存	8 B	256 B		13.0 KiB	512 209

表 1：分配器状态分散和缓存开销

系统扩展到我们的基准测试机器（52 个核心，128 GiB，1 个内存区域），以及在此设置下进行完整枚举需要访问多少个不同的 64B 缓存行。

由于 LLFREE 使用了位域和计数器数组，其状态在基准测试机器上仅分散到 4.14 MiB（DRAM 的 0.0032%）。这主要是由 4 KiB 位域（4 MiB）造成的。因此，即使是完整的状态扫描也能轻松容纳在机器的 35 MiB L3 缓存中，而这只需要加载 67754 个缓存行。此外，由于 LLFREE 不重新分配内存，因此内核无需映射物理内存，其状态分散程度与其内存开销相等。

相比之下，Linux 分配器将其大部分状态存储在 struct page。它需要一个标志，并将 LRU 列表指针（双向链表）的 16 字节重新分配给其按顺序排列的空闲列表和每个 CPU 的页面缓存。由于链表的分散特性，Linux 分配器（可能）将其状态分散在 516 MiB（DRAM 的 0.39%）上。更糟糕的是，由于每个 struct page 都位于其自身的缓存行上，一次完整的状态扫描将加载 33,555,169 个缓存行。因此，与 LLFREE 相比，Linux 分配器的状态不仅分散在 125 倍的内存上，而且完整扫描所需的缓存行数也多出 495 倍。

此外，由于 LLFREE 不依赖于 struct page，这也引出了一个问题：这些记录是否可以缩小或消除？这些记录目前占用 1.56% 的 DRAM。遗憾的是，移除分配器对 struct page 的依赖并不会直接导致每个帧的记录数量减少，因为其他内核子系统会在帧分配时出于各种目的重用 LRU 列表指针（在 Linux 源代码中，struct page 本质上是一堆联合体）。因此，缩小甚至消除 struct page 是一项涉及面广且极具挑战性的任务。

然而，像 LLFREE 这样不需要帧级记录的分配器可以显著缓解这一挑战，因为我们只需要为已分配的帧创建帧级记录。例如，随着 Linux 当前向 struct folio [11]，它描述了一组物理上连续的帧，因此有可能分配此动态记录。从这个意义上讲，我们认为 LLFREE 是解开结构页面依赖关系的重要第一步。然而，彻底消除这种依赖关系仍是需要进一步研究的课题。

5.6 碎片和压缩成本

接下来，我们考察两种分配器的大帧碎片化行为。为此，我们首先定义一个衡量碎片化的指标，以及消除碎片化所需的内存压缩成本。为了衡量碎片化程度，我们统计在清空所有缓存但不进行内存压缩的情况下可以分配的大帧数量。然后，我们将此数量与进行内存压缩后可以分配的大帧数量进行比较，从而了解系统中的碎片化程度。

为了评估压缩成本，我们考虑释放可能的最大巨型帧所需的最少 4 KiB 复制操作次数。为了计算此指标，我们 (1) 计算每个可能巨型帧中空闲的 4 KiB 帧的数量，(2) 对结果数组进行排序，以及 (3) 将“最满”的巨型帧与“最空”的巨型帧进行匹配，同时计算所需的复制操作次数。请注意，Linux 会跳过步骤 1 和 2，并将内存移动到区域的开头，这会导致内存压缩效果欠佳。然而，使用 LLFREE 时，对子数组进行排序（参见图 3）可以使我们接近最优方案。

为了比较这两种分配器，我们进行了以下综合基准测试：首先，我们生成一个初始内存配置，该配置代表物理内存碎片化程度最高时的最坏情况。为此，我们先分配 125 GiB 区域中 90% 的内存，然后再随机释放一半的 4 KiB 帧。从这种碎片化状态开始，我们执行 100 次迭代，每次迭代释放 10% 的已分配内存（以随机选择的 4 KiB 帧的形式），然后再重新分配相同数量的内存（以单独的 4 KiB 帧的形式）。每次迭代后，我们分别清空 CPU 本地缓存（buddy）和树预留（LLFREE），并测量大帧碎片化程度和压缩成本。请注意，我们仍然关闭了 Linux 内存压缩器（如第 4.2.2 节所述）。与我们的其他基准测试一样，我们不会触发无法满足的分配请求，因为这需要同步压缩。我们使用所述的离线计算方法，测量了两种分配器每次迭代的假设压缩成本。

图 10 显示了这两个指标随时间的变化。在此基准测试中，尽管整个内存循环了十次（100 次迭代），Linux 分配器只能恢复一个大帧。此外，压缩成本仅下降了 3.3%，表明大帧的碎片整理速度随时间推移缓慢。另外，我们的方案有利于 Linux，因为我们清空了每个 CPU 的缓存并使用了最优的压缩成本指标。相比之下，LLFREE 可以在一段时间内恢复 46.6% 的初始污染的大帧。

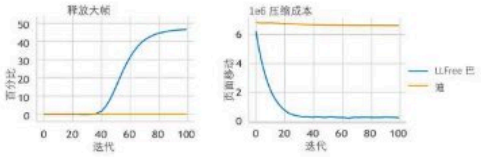

图 10：随机重新分配 10% 已分配内存的迭代过程中，释放的巨型帧（左）和压缩成本（右）。

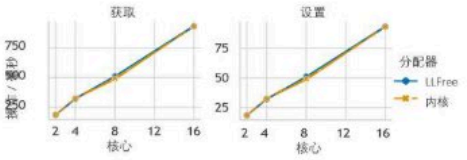

图 11：内存层每毫秒的 Get/Set 次数基准测试。

基准测试。虽然看起来碎片整理大约在第 50 次迭代后才开始生效，但观察压缩成本可知熵从一开始就在下降；只是还没有完全空闲的大帧。在执行总计达到总内存量的重新分配（10 次迭代）后，压缩率达到了 39.1% 成本，经过 50 次迭代后，我们只需要最初所需复制操作的 4.9%。

总体而言，我们看到 LLFREE 表现出被动碎片整理行为，由我们的子树分配策略引导。由于伙伴系统不跟踪拆分桶的"满度"，因此无法以低成本模拟此行为。

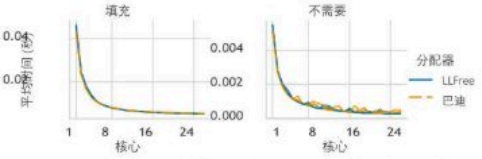

图 12：写入基准测试中填充/释放 128 GiB 内存映射的平均时间。

5.7 崩溃恢复

由于通过真实系统崩溃验证崩溃一致性过于耗时，难以获得可靠的结果，我们使用 DAX 映射的 NVRAM 区域上的用户空间基准测试来模拟 LLFREE 在常规关机和崩溃情况下的恢复（第 3.2.3 节）：为此，（1）我们初始化一个 128 GiB 区域的分配器，（2）随机分配其中一半，以及（3）像第 5.6 节中那样重复分配和释放内存。对于常规关机，进程在（2）之后终止；而对于崩溃，我们通过在（3）期间使用 SIGKILL 随机终止基准测试来模拟。之后，另一个进程从持久内存中恢复分配器的状态。

我们总共注入了 1000 次崩溃，LLFREE 在所有情况下都能恢复其状态；在大约一半的实验中，我们实际上丢失了帧（每个核心最多丢失一帧），这是预期的，因为核心的所有时间都花在了分配/释放操作上。在恢复过程中，LLFREE 会遍历位域以纠正子计数器，这平均需要 2460 微秒。相比之下，常规的 NVRAM 重新初始化，其中 LLFREE
只需遍历子表，仅需 477 微秒。恢复操作以单线程方式完成，但如果恢复时间成为问题，可以并行化并在后台部分执行。

5.8 应用级基准测试

作为实际应用基准测试，我们使用了 memtier[6]，它评估 memcached 键值缓存的性能存储。它测量 Get 和 Set 请求的吞吐量。遗憾的是，如图 11 所示，我们没有看到显著差异。由于页面分配器主要由缺页处理程序使用（该程序采用延迟分配内存），因此其他相关内存管理组件的开销可能会掩盖任何性能提升。

为了进一步验证这一假设，我们创建了写入基准测试，该测试映射一个大的内存区域并并行填充它。填充操作是通过将一个非零值写入页面的第一个字节来完成的，这将触发缺页和随后的分配请求。对于取消映射，使用 madvice/DONTNEED 系统调用。基准测试针对 1-26 个核心执行，内存区域在核心之间平均分配。同样，图 12 中的结果显示 buddy 和 LLFREE 分配器之间没有显著差异，就像 memtier 基准测试一样。

利用 perf 分析器，我们测量了大部分运行时消耗在哪里。图 13 中的火焰图显示，页面分配器（黄色）仅占运行时的 5.3%。主要的瓶颈在于结构体 mm 读写锁（橙色，17.3%）以及 LRU、cgroups、反向映射和结构体页面标志的更新（绿色，28.7%）。由于大多数结构体页面更新宏都是内联的，因此实际耗时可能更高。这些内存管理瓶颈与其他研究结果一致 [7, 12, 14, 32, 35]。

6 讨论

我们的结果表明（除了崩溃一致性之外），LLFREE 在用户级和内核级基准测试中表现出卓越的可扩展性，这得益于其无锁且对缓存友好的设计。然而，尽管存在一些应用程序，但这些优势在端到端基准测试中尚未显现。

[6] https://github.com/RedisLabs/memtier_benchmark

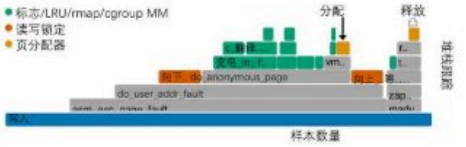

图 13：使用 Buddy 分配器在 8 个核心和 128 GiB DRAM 上进行写入基准测试的火焰图。

操作系统级内存分配性能已是一个大问题 [12, 14, 32, 35]。

我们认为，鉴于系统间深度纠缠和日益增长的复杂性，可扩展性问题只能通过自下而上的方式解决，而 LLFREE 及其设计理念正是朝着这个方向迈出的第一步。我们相信，如果非易失性存储器将来要发挥作用，那么它们的内核级管理必须与易失性存储器协同设计——而 LLFREE 正是朝着这个方向迈出的重要一步。最后，我们的结果还表明，如果从一开始就考虑可扩展性和持久性 [8, 16, 46] 的结合，那么在内核设计中将取得特别好的效果。

7 相关工作

许多用于非易失性内存的通用分配器 [3, 9, 33, 36, 41, 45, 52] 已被提出。与 LLFREE 不同，所有这些分配器都使用日志记录来确保崩溃一致性，这会增加非易失性存储器 (NVM) 的损耗 [49]，并且使用锁来实现多线程操作；一些分配器通过使用多个分配器实例 [36, 41]、每个 CPU/线程的空闲列表 [3, 9] 或范围锁定 [52] 来减少锁争用。其中，PAllocator [36] 与 LLFREE 有相似之处，因为它具有反碎片措施，并且与我们底层的分配器类似，仅将其部分状态存储在 NVRAM 中，并在启动时恢复其易失性状态。然而，所有这些持久分配器都是通用用户空间分配器，因此与 LLFREE 等内核页面帧分配器相比，它们的设计目标有所不同。

在操作系统方面，Twizzler [5] 明确地围绕非易失性内存构建，但它并不包含持久化的页帧分配器。相反，系统会在每次重启时从 DRAM 中的持久对象重建分配器状态。据我们所知，LLFREE 是第一个在操作系统中使用的持久化页帧分配器。

虽然防止外部碎片是分页机制最初的动机之一[2]，但将其扩展到不同帧大小后，这个问题又重新出现。为了简化大帧的回收，放置策略[19-21]对分配的内存进行分类（例如，可移动、可回收），并将它们空间聚类到不同的大帧上。为此，Linux 为每个伙伴顺序设置了多个空闲列表，每个列表对应一个不同的类别。LLFREE 目前不支持这种分类。但是，可以通过专门的树（参见第 3.2.2 节）轻松扩展其功能。对于 Linux，已经提出了具有更好聚类特性的策略[37]。

其他减少大帧碎片化的措施包括主动压缩 [30] 和预期连续内存预留 [27, 34]。甚至已经提出了硬件解决方案，例如构建非连续内存的大帧 [43]，或者在地址转换中增加一个类似于嵌套分页 [4] 的层级 [51]。相比之下，LLFREE
是一种纯软件解决方案，它能够被动地对大页面帧进行碎片整理，同时保持速度快且崩溃一致性。

8 结论

管理物理内存的页帧分配器是现代操作系统所有内存管理的核心。然而，正如我们在 Linux 的示例中所展示的，其传统的基于锁的设计，以及大量的列表和分布式元数据，已经跟不上硬件向大规模并行系统发展的步伐，而这些系统需要大量的异构易失性和非易失性内存。这导致了内部复杂性、可扩展性差、内存碎片化严重，以及普遍不适合在非易失性内存上实现崩溃一致性分配。

我们提出了 LLFREE，这是一种新型的无日志、无锁页面帧分配器。凭借其无锁和以缓存为中心的设计，LLFREE 在并行分配方面实现了出色的可扩展性（在 26 个核心上并行分配 2 MiB DRAM 时，速度比 Linux buddy 分配器快 53 倍），同时有效地降低了大页面碎片。所有内存释放/分配操作都通过单缓存行事务在内存中体现，LLFREE
可以在 eADR 系统上以接近 DRAM 的速度，无需日志记录即可为持久化 NVRAM 提供崩溃一致性。我们将 LLFREE 集成到 Linux 中取得了成功，但也暴露出其内存管理子系统的诸多瓶颈，以及伙伴分配器与其之间的深度纠缠。这些问题需要进一步研究和重新设计。我们认为 LLFREE 是朝着彻底重构操作系统内存管理结构迈出的关键第一步。

致谢

我们感谢匿名审稿人提出的宝贵意见和建设性建议。特别感谢 Godmar Back，他严谨而又充满鼓励的指导极大地帮助我们改进了本文的内容和质量。

这项工作由 Deutsche Forschungsgemeinschaft（DFG，德国研究基金会）资助 - 468988364、501887536。

附录

(1) 如果保留计数器 cP ≥ $2^{o'}$，则 cP ← cP − $2^{o'}$并继续执行 (2)。
  (a) 否则，与全局 cU 同步。如果计数器足够大，则重复步骤 (1)。
  (b) 否则，保留一棵新树并重复 (1)。
(2) 依次在相应的子数组中搜索 cL ≥ $2^{o'}$的条目。如果搜索失败，则保留一个新的子树并重复步骤 (1)。
  (a) 对于基本帧，递减 cL，在相应的位域中查找零位并将其设置。
  (b) 对于巨型帧，设置 cL = 0 和 a = 1。
(3) 返回已分配的页帧号 (PFN) 或 NULL。

(a) 分配一个 o 阶帧。

(1) 检查相应的子条目。
  (a) 对于基本帧，检查 cL 是否 ≤ 512 − $2^{o'}$且 a = 0，如果是，则继续执行步骤 (2)。
  (b) 对于巨型帧，检查 a = 1，将其设置为零，并继续执行 (4)。
(2) 切换第一层位域中的相应位。
(3) 增加子计数器 (cL ← cL + $2^{o'}$)。
(4) 如果此释放操作在另一个树中，则递增保留的 cP 或全局 cU。
  (a) 当更新全局部分树条目时，如果过去的 F 分配也影响了它，则保留它。

(b) 释放一个 o 阶帧。

图 14：分配和释放算法。此描述不包括所有边界情况和错误路径。

A 工件附录

摘要

该工件包含评估 LLFree 所需的必要工具和资源。LLFree 是一种新型的无锁无日志分配器设计，具有良好的可扩展性、内存占用小，并且易于应用于非易失性内存。为了简化评估，该工件被打包成一个 Docker 镜像，其中包含论文评估中使用的各种基准测试及其所有依赖项。这些基准测试旨在对分配器在各种场景下的性能进行测试。它们允许其他研究人员将 LLFree 的性能与传统的 Buddy 分配器进行比较，并复现我们的实验结果。此外，该镜像还包含论文图表的原始数据和脚本，使我们的评估过程尽可能透明。

范围

这些基准测试表明，在具有高并行性的系统和工作负载上，LLFree 分配器的可扩展性优于 Buddy 分配器。然而，在虚拟机（即使使用 KVM）中执行这些测试会导致结果不够准确。因此，在论文中，我们在原始硬件上构建并测试了修改后的 Linux 系统。尽管如此，结果应该与论文中的评估结果呈现相似的趋势。

内容

Docker 镜像包含构建和运行基准测试所需的所有依赖项和脚本，以及生成相关图表和数据所需的脚本。它还包含一个 Python 脚本 (run.py)，作为管理构建、基准测试和绘图过程的中央命令中心。分配器可以在用户空间和自定义构建的 Linux 内核中进行测试，该内核集成了在 QEMU+KVM 虚拟机中运行的 LLFree。对于后者，镜像包含一个 QEMU+KVM 虚拟机以及用于启动它并运行内核基准测试的脚本。它还包含论文中所示的基准测试的原始数据和图表。

托管

Docker 镜像以及分配器、修改后的 Linux 内核和基准测试的仓库托管在 GitHub 上：

- **Docker 镜像**：此镜像包含一个执行环境，可以轻松运行和评估基准测试。
- **自由长椅**：基准测试脚本和结果。（标签 = atc23-artifact-eval）
  - 工件说明可在 artifact-eval/README.md 中找到。

- **llfree-rs**: LLFree 分配器的 Rust 实现。（标签 = atc23-artifact-eval）
- **llfree-linux**：修改后的 Linux 内核，可配置为使用 LLFree 而不是 Buddy 分配器。
  （标签 = atc23-artifact-eval）
- **linux-alloc-bench**：用于对页面分配器进行基准测试的内核模块。（标签 = atc23-artifact-eval）

要求

由于我们的基准测试打包在 Docker 镜像中，并且不依赖于特定的硬件，因此唯一的前提条件是：

- 基于 Linux 的 KVM 系统。我们已在 Linux 6.0、6.1 和 6.2 上进行了测试。
- 至少需要 8 个物理核心和 32GB 内存（越多越好）。配置较低也能运行，但结果可能意义不大。
- 为获得更稳定的结果，应禁用超线程和 TurboBoost。由于虚拟机未进行相应配置，内核基准测试可能会受到特别大的影响。
- 已正确安装并运行的 Docker 守护进程。

后续步骤

工件说明可在 llfree-bench 仓库中找到artifact-eval/README.md. artifact-eval 目录还包含文档中提到的所有脚本，以及如果需要自行构建镜像的 Dockerfile。