可编程缓存引擎的独特之处

研讨会论文Jos´e Peixoto, Alexis Gonzalez*,Janki Bhimani*,Raju Rangaswami*,Cl´audia Brito,Jo˜ao Paulo, Ricardo Macedo*INESC TEC &* 米尼奥大学 *佛罗里达国际大学

arXiv:2603.14357v1 [cs.OS] 15 Mar 2026

摘要——**像 CacheLib 这样的可编程缓存引擎被广泛用于生产系统，以支持多租户环境中的各种工作负载。CacheLib 的设计侧重于性能、可移植性和可配置性，使应用程序只需付出极少的实现工作，就能继承缓存方面的改进。然而，其在动态且不断演变的工作负载下的行为仍未得到充分研究。本文针对动态且易变的环境，对 CacheLib 在多租户设置下进行了实证研究。我们对多种 CacheLib 配置进行评估后发现，在此类环境中，CacheLib 存在若干影响其有效性的局限，包括配置僵化、运行时适应性有限，以及缺乏服务质量支持与协调；这些问题会导致性能欠佳、内存使用效率低下和租户饥饿。基于这些发现，我们概述了未来的研究方向，以提升未来缓存引擎的适应性、公平性和可编程性。**

索引词——**可编程缓存、CacheLib、多租户、适应性。**

I. 引言

数据库、键值存储、内容分发网络 (CDN) 和机器学习引擎等数据密集型系统，已成为现代 I/O 基础设施的重要组成部分 [1]。为了以高吞吐量和低延迟处理大量数据，这些系统依赖内存缓存。通常，每个系统都会采用针对自身特定需求优化的缓存，这些需求包括读写比、访问分布、I/O 粒度（块、文件、对象）以及并发模型（例如，单租户或多租户） [2], [3]。

尽管各不相同，缓存系统通常有着共同的目标，也面临相似的挑战（例如接口、驱逐策略、内存分配策略）。缺乏统一的缓存抽象会导致功能碎片化、重复的工程工作以及大量维护开销。为了解决这些局限，Meta 推出了 CacheLib [2]，这是一款通用可编程缓存引擎，为设计高性能缓存提供了一套通用构建模块。CacheLib 提供灵活且可扩展的 API，涵盖缓存索引、线程安全结构、驱逐策略、内存管理和多租户隔离，使用户能够为各种数据密集型系统设计精细调优的缓存。它的成功源于可配置性和性能可移植性，使应用程序无需大规模重新实现即可受益于缓存改进。

如今，CacheLib 为 Meta 的 70 多项服务（例如，内容分发网络、键值存储、推荐引擎、数据库） [2], Pelikan.io 以及多个研究项目提供支持。为了便于管理并提高成本效益，多个服务和应用程序通常共置于同一个计算节点上，每个应用程序管理自己的缓存实例。即使是单个应用程序，也可能使用多个缓存（例如，采用不同的驱逐策略和大小）来服务不同的工作负载 [2], [5]。

尽管 CacheLib 奠定了坚实的基础，但在共享的多租户部署环境中，仍有一些重要问题有待解答：*i)*CacheLib 如何简化多租户场景下的内存管理？*ii)*它如何应对访问模式不断演变、资源需求动态变化的工作负载？

本文首次对 CacheLib 在动态多租户环境中的性能与适应能力进行了实证研究。我们的研究揭示了若干局限性，也由此开辟了新的研究方向。

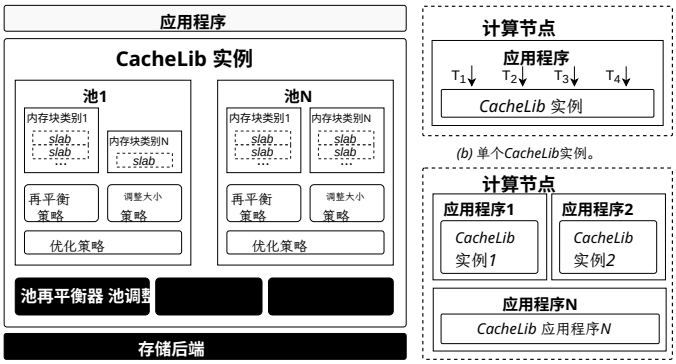
配置模型僵化。 CacheLib 的配置模型过于僵化，难以适应共置工作负载的动态特性。尽管它提供了许多调优选项（即，驱逐策略、内存分配和再平衡方案），但这些参数通常在初始化时设定，无法适应不断变化的工作负载（例如，访问分布变化、租户加入或离开），从而导致性能欠佳。

适应能力有限。CacheLib 包含池优化器（*pooloptimizer* ）等机制，可在运行时调整内存分配。然而，我们的研究表明，在工作负载不断变化时，这些机制往往效果不佳，导致性能波动和内存使用效率低下。

缺乏服务质量（QoS）支持。CacheLib 激进的重新分配策略优先考虑全局缓存效率，而非单个租户的需求。缺少服务质量保障、优先级机制或差异化机制，可能导致资源饥饿，尤其是在延迟敏感型工作负载与更激进的租户共享资源时。

缺乏实例间协调。多个CacheLib实例可以在同一计算节点上运行，但彼此隔离。缺乏协调会妨碍资源的高效利用和全局内存再平衡，尤其是在工作负载倾斜的情况下：某些实例资源配置不足，而其他实例资源配置过度。

总而言之，我们的研究结果表明，尽管 CacheLib 为多租户环境提供了关键构建模块，但在处理动态工作负载和共享资源带来的复杂性方面仍显不足。为弥补这一差距，我们指出了若干研究方向，以提升 CacheLib 的适应能力、实例间协同能力和服务质量支持，使其更适用于现代数据密集型应用程序，并更为高效、稳健。



*(a) CacheLib*设计概览。 *(b)* 单个*CacheLib*实例。 *(c)* 多个*CacheLib*实例。

图1： *CacheLib*设计概览（*a*）以及生产环境中的典型适用性模型（*b*-*c*）。

II. 背景

本节介绍 CacheLib 引擎的背景，概述其设计、核心功能和适用性。

A. CacheLib 概述

CacheLib [2]是 Meta 开发的一款可嵌入式缓存库，专为在并发环境中实现高性能而设计。其架构着重于内存效率、工作负载隔离和运行时适应性。它提供了一个简单的线程安全接口，用户可通过键值对抽象存储和检索项目。如图 1 *(a)* 所示，CacheLib 的核心由四个主要组件构成。内存池是一个相互隔离的区域，拥有独立的分配策略、驱逐策略和内存限制，用于存储和提供数据项目。池再平衡器 和 池调整器 通过在池内及池间移动内存，最大限度地减少内存浪费和内存碎片化。池优化器 则通过动态调整各个池的大小，致力于提高整体内存使用率。

内存分配与碎片化管理。CacheLib 采用基于 slab 的内存分配方案，以最大限度地减少外部内存碎片化。它并非分配大小可变的内存块，而是将内存划分为固定大小的 slab，每个 slab 都对应一个用于存储大小相近项目的（slab）类。每个 slab 类管理自己的内存区域，而单个内存池可包含多个 slab 类。当内存分配到各个 slab 类后，系统便进入静态状态，这可能导致*slab* 固化。这种现象发生于 slab 被分配给已不再访问的项目大小时，从而造成内存浪费，并导致其他 slab 类出现分配失败。为缓解这一问题，CacheLib 集成了一个池再平衡器，这是一个后台工作线程，它*1)* 识别内存池中分配不均衡的 slab 类，*2)* 选择拥有多余 slab 的受害类，并*3)*将 slab 转移到需求更高的接收类。此外，每个池都采用可配置的再平衡策略来指导 slab 分配，例如尽量减少分配失败或最大化单项命中率。

多租户支持。为了支持多个并发工作负载，CacheLib 支持创建大量内存池，每个内存池都隔离特定的工作负载或租户，从而实现在同一个 CacheLib 实例中灵活地进行内存分区。默认情况下，每个内存池都会分配一个固定的内存限制。用户可以在运行时调整此限制，但增加内存池的分配量并不保证它能立即获得更多内存，因为其他内存池可能仍占用着内存块，直到显式地重新平衡为止。为了强制执行这些限制，CacheLib 实现了一个池调整器，即一个后台工作线程，它*1)* 识别超出其配置的内存限制的内存池，*2)*从超出其容量的内存池中选取内存块，并*3)* 将其转移到资源配置不足的内存池。此过程有助于系统更好地执行内存限制，但需要手动启动调整大小。此外，每个内存池都会实现一种调整策略 （例如，最大化命中率、最小化分配失败），以指导调整过程。

自适应内存管理。 为减少手动配置开销，CacheLib 实现了池优化器，这是一个用于自动调整内存池大小的后台工作线程 [4]。它持续监控工作负载，分析驱逐队列末端的访问模式，以估计工作负载压力，并为每个内存池提出新的内存限制，从而提高整体命中率。为了执行这些限制，池优化器和调整器必须协同工作。

B. CacheLib 请求工作流

CacheLib 中的请求工作流分为两部分：前台 流程负责处理应用程序提交的读取和写入请求，而后台流程负责管理内部系统操作。

前台流程。应用程序通过键值对抽象插入和检索数据对象，从而与缓存交互。在写入时，应用程序首先尝试在目标池中为对象分配内存。如果分配成功，应用程序会将数据复制到已分配的区域中，之后事务才能提交到缓存。确保数据持久化到持久存储中是应用程序的责任。在读取时，给定一个键，CacheLib 会搜索所有内存池；如果找到对象，则返回该对象；如果发生缓存未命中，则由应用程序负责从持久化存储中检索数据。

后台流程。后台流程负责维护缓存的内部结构，并通过后台工作线程（包括池再平衡器、调整器和优化器（§II-A））适应工作负载变化。这些操作会定期异步执行，且独立于前台流程，使 CacheLib 能够在不中断应用程序服务的情况下适应工作负载变化。

C. 在生产环境中的使用

CacheLib 支持多种部署场景，具体取决于所涉及的应用程序和租户数量。本节将介绍两种常见的生产环境设置，它们是后续章节讨论的基础。

单实例、多租户。 如图 1*(b)*所示，单个应用程序或服务（例如内容分发网络、键值存储、

社交图系统和数据库）可以仅使用单个 CacheLib 实例来支持多个租户 [2]。租户通常由具有不同请求模式和内存需求的独立线程、进程或组件表示。每个租户都会分配一个专用内存池，以确保隔离和可预测的性能。

多个实例。 如图 1*(c)*所示，多个 Cache- Lib 实例部署在同一计算节点上，每个实例为不同的应用程序或服务提供服务。这在复杂的数据密集型软件栈中十分常见，例如 Uber、Amazon 等所使用的软件栈 [6]–[8]。各实例彼此隔离，分别配置独立的设置和分配策略，并设定固定的内存上限。

III. 理解 CacheLib 性能

为了了解 CacheLib 在动态且异构的环境中的性能和适应能力，我们试图回答以下问题：

- CacheLib 在请求流行度分布各异的工作负载下表现如何?
- 在不同的内存划分方式下，CacheLib 表现如何?
- CacheLib 能否适应动态变化且不断演进的工作负载?

硬件和操作系统配置。实验在 Deucalion 超级计算机的计算节点上进行，节点配备 2× 64核 AMD EPYC 7742 处理器、256 吉比字节内存和一块 480 吉比字节固态硬盘，运行 RockyLinux 8。软件方面，我们使用了 CacheLib v20231101。

方法。实验测试平台由三个组件构成：一个充当应用程序的基准测试程序，用于生成具有不同访问分布的请求；一个 CacheLib 实例，用于缓存应用程序的读取请求；以及作为持久化存储后端的 RocksDB。写操作始终提交给 RocksDB；为避免缓存不一致，如果某个项之前已被缓存，写操作也会传播到缓存。对于读取，请求首先提交给 CacheLib——若缓存命中，它会向应用程序返回相应项；若缓存未命中，应用程序则从持久化存储中获取该项，并将其插入相应的 CacheLib 内存池。此外，为了单独评估 CacheLib 的性能优势，我们禁用了 RocksDB 的内部块缓存和页缓存。

与以往研究（[9]–[11]）一样，我们通过对 4 个租户进行实验，评估了 CacheLib 在异构多租户环境中的性能。每个租户都配置了不同的工作负载特征和内存限制。

工作负载。实验使用只读工作负载开展，这是评估缓存系统时的常见做法 [2], [9], [12]。每个租户都配置了不同的工作负载分布，以模拟生产环境中观察到的访问模式 [13]。具体而言，租户 T_1,T_2,和 T_3 分别采用偏斜因子为 1.2、0.9 和 0.6 的齐夫分布，而租户 T_4 则配置为均匀分布 。此外，每个租户都在一个独占键空间上运行，该键空间包含 2000 万个唯一条目，每个条目大小为 1 千字节，数据总量约为 20 吉比字节。除非另有说明，所有租户均并发运行其工作负载，持续 2500 秒。此外，在工作负载运行前，会将每个租户的 2000 万个唯一键值对（每个大小为 1 千字节）预加载到存储后端中，数据总量约为 80 吉比字节。因此，CacheLib 实例的配置容量仅能存储数据集的 10%，与先前的研究类似（[12], [14]），总内存容量为 8 吉比字节，并划分为四个内存池，每个内存池专供一个租户使用。

设置。 为了探究不同内存管理策略的影响，我们使用了三种 CacheLib 配置：基线指默认的 CacheLib，仅启用池再平衡器工作线程；自定义 在基线设置的基础上，集成了自定义内存分配策略，并与池调整器配合，为每个池强制实施用户定义的动态内存限制；优化器 则启用池调整器 和池优化器工作线程。自定义 和优化器 中的工作线程均每秒执行一次；我们为池调整器设置了最大化命中率的调整策略，这最符合我们的需求。

A. 静态工作负载下的内存分区

我们首先在静态环境中评估不同 CacheLib 配置的性能和适应性。在该环境中，所有租户同时开始并完成其工作负载。我们采用以下应用不同内存分配策略的配置：

- 基线： 内存均匀分配给所有租户，每个池获得缓存总容量的 25%。
- 自定义： 优先照顾齐夫偏斜最高的租户，将 91% 的缓存容量分配给 T_1，其余空间则在其他租户之间均匀分配。
- 优化器： 初始采用均匀分配策略，但会在执行期间根据观测到的缓存性能指标动态调整内存分布，并经过调优以最大化总体吞吐量。

结果。 图 2 和图 3 分别展示了三种 CacheLib 配置下各租户的吞吐量与分配内存，以及全局吞吐量与分配内存。正如预期，由于各租户的数据分布不同，其吞吐量存在显著差异。在基线 设置中，经过初始缓存预热后，T_1达到 ≈140kops/s，分别以 14×、25×和 28×的优势超过 T_2, 、T_3，和 T_4 。这些结果表明，当租户呈现异构访问模式时，均匀分配内存可能导致性能失衡。

自定义配置表明，优先处理高度倾斜的工作负载可带来显著的性能提升。如图 3 所示，与基线相比，系统整体吞吐量最高提升至 1.8×，其中 T_1 的吞吐量最高可达 280 kops/s。这是因为发往后端的 I/O请求数量与命中率成反比，也就是说，当命中率已经很高时，命中率的小幅提升也能显著改善吞吐量。对于遵循高度偏斜分布的 T_1, , 若要进一步提高命中率/吞吐量，就需要不成比例地

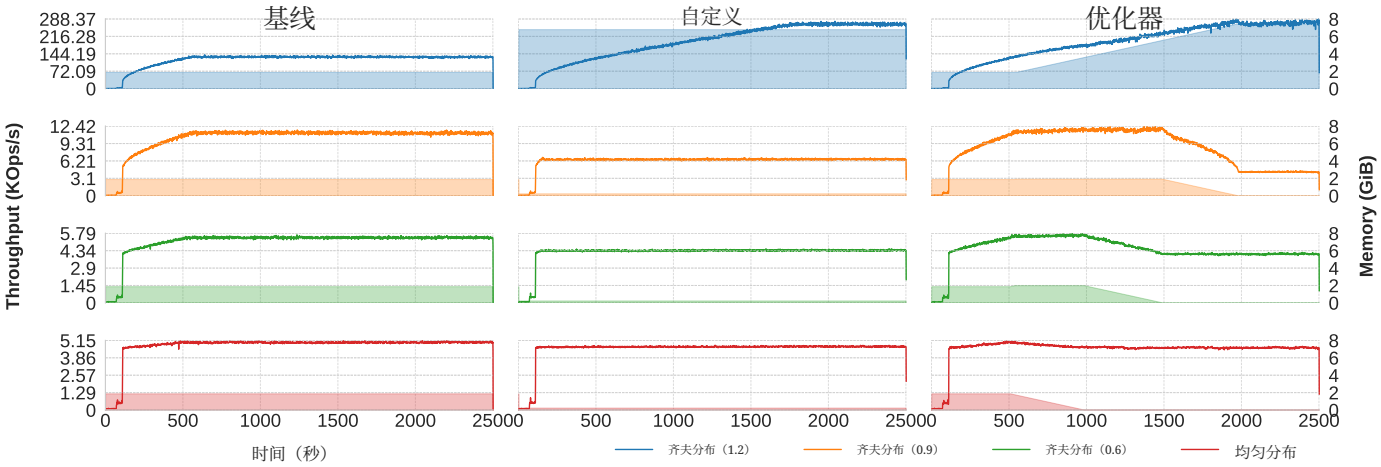


图 2： ***CacheLib*** 基线、自定义和优化器设置（各列）下的每租户性能比较。 每行展示一个租户（T_1 至 T_4）随时间变化的吞吐量（千次操作/秒）和分配内存（吉比字节）。左侧纵轴表示吞吐量（折线），右侧纵轴表示分配的内存（填充区域）。

增加内存份额，因此分配比例为 91%。然而，如果目标是最大限度地提高所有租户的命中率，或确保公平性，那么这种分区方式可能并非最优。

> **观察 1。** 在异构工作负载下，均匀分配内存可能导致严重的性能失衡。根据工作负载分布微调缓存分区，可以显著提升吞吐量。

如图 2 所示，优化器 设置最初采用与基线相同的均匀内存分布，但随后会根据观察到的工作负载行为逐步重新分配内存。不过，这一过程需要一个预热期来收集足够的统计数据。只有在 ≈500秒之后，优化器 才开始有效调整内存分配，并逐渐收敛至更优状态。尽管存在这一初始延迟，如图 3 所示，优化器 的总体吞吐量仍高于基线配置，达到 ≈290 千次操作/秒；在相同内存容量下，性能提升了 1.85× 。

> **观察 2。** 根据工作负载特征动态重新分配内存，即使在资源约束相同的情况下，也能优于静态配置。

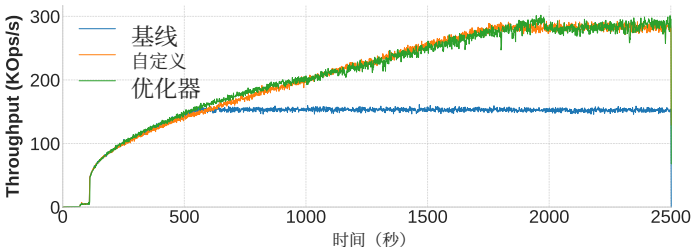


图 3： 不同分配策略下的总体吞吐量。工作负载感知内存分配显著优于统一策略。

有趣的是，进一步检查（图 2）后发现，优化器 将几乎全部缓存容量重新分配给 T_1，，使其内存池从 2 吉比字节增加到 ≈8 吉比字节。这导致其余租户资源饥饿，吞吐量大幅下降，甚至低于基线设置下观察到的性能。虽然这种行为可能适用于以最大化全局吞吐量为首要目标的环境，但并不适用于需要为每个租户提供服务质量（QoS）保证或施行不同优先级的场景。在这些场景中，即使全局吞吐量有所提升，优化器激进的重新分配策略仍会损害租户之间预期的性能隔离。

> **观察 3。** 如果没有服务质量或优先级划分机制，*CacheLib* 的动态重新分配策略可能导致资源饥饿，损害租户间的性能隔离，并造成内存分配不公平。

B. 动态工作负载下的内存分区

基线和基线 以及自定义 配置都依赖静态内存分配，即在设置时固定分配给每个租户的内存。虽然这种方法在工作负载特征和系统设置随时间保持不变的稳定场景中可能有效，但在更易变的环境中未必如此。

现在，我们评估不同的 CacheLib 设置在此类环境中的表现。在这种环境下，租户会在不同时间加入或离开系统。具体来说，每隔 500 秒依次向系统添加一个租户；待所有租户都活跃 500 秒后，再按照加入顺序的逆序让它们离开系统。各租户最初的内存分配遵循 §III-A 中相同的设置。对于自定义 设置，分配策略按照以下规则进行了调整：*1)*启动时，分配给任何租户的内存都不会超过其预期需求；*2)*如果有剩余内存容量，则按比例分配给活跃租户，以避免内存资源利用不足。最后，本实验省略基线 配置，因为它

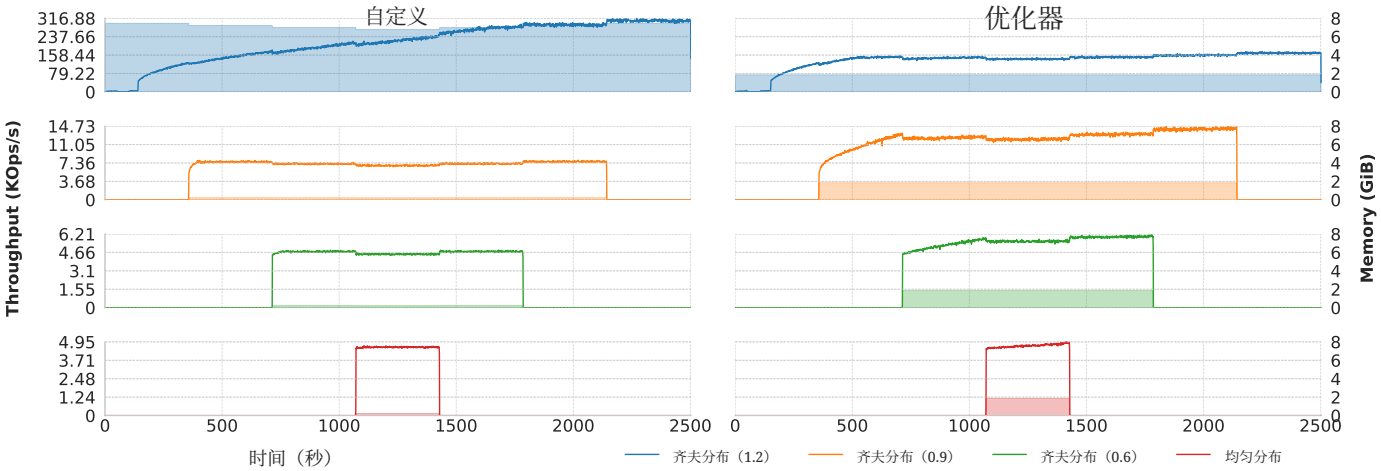


图 4： 动态工作负载下 ***CacheLib*** 自定义设置与优化器设置的每租户性能比较。 每一行展示了各租户（T_1 至 T_4）随时间变化的吞吐量（千次操作/秒）和分配内存（吉比字节）。左侧纵轴表示吞吐量（折线），右侧纵轴表示已分配内存（填充区域）。

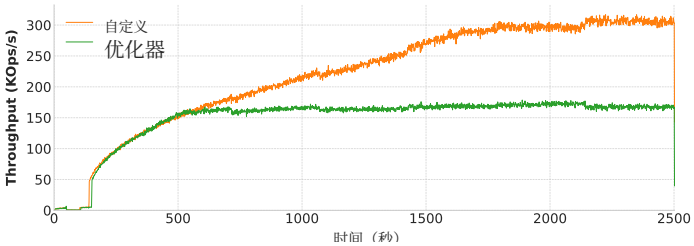


图 5： 动态工作负载下的总体吞吐量。 基于工作负载的重新分配优于静态分配或延迟适应。

无法适应动态工作负载，因此表现会与 §III-A 中相同。

结果。 图 4 展示了在自定义 和优化器 设置下各租户的吞吐量和内存分配情况。与先前的结果（§III-A）相反，由于租户的活跃时间太短，无法提供足够的数据来触发重新分配，因此优化器 未能动态重新分配内存并收敛到更优状态。尽管如此，即使租户交错运行的时间更长，面对工作负载快速变化的特性，优化器 仍可能因内存分配决策有误而无法收敛。无论如何，优化器 未能适应变化，仍将总缓存容量的 25% 静态分配给每个租户，最高吞吐量为 ≈170 kops/s。

另一方面，如图 5 所示，自定义配置的峰值吞吐量达到 ≈315 kops/s，比相同内存约束下的优化器 高出 1.85× 。这一提升源于两个关键因素。首先，自定义 配置的初始内存分配与预期工作负载特征更加吻合，确保每个租户获得的内存份额更接近其实际需求。其次，自定义策略会按比例回收未使用的内存并重新分配给活跃租户，从而提升整体资源使用率和吞吐量。例如，在实验开始后的前 500 秒内，只有 T_1处于活跃状态，此时优化器 将 T_1 限制在缓存的 25%，自定义设置会分配全部缓存容量，从而最大限度地提升其性能。随着更多租户加入，自定义策略会继续按比例动态调整内存分配，确保有效利用缓存。

> **观察4**。要在动态环境中最大限度地提升缓存效果，必须将工作负载感知内存分配原语与未使用内存资源的按比例重新分配相结合。

IV. 讨论

CacheLib 对可编程缓存引擎的设计作出了重大贡献，尤其是在多租户环境中。尽管它提供了池优化器 以改善资源管理，但我们的研究表明，CacheLib 在多租户部署中处理动态工作负载时仍面临挑战，这也为新的研究开辟了道路。

应对动态工作负载。我们的研究表明，池优化器无法妥善应对动态租户，导致内存分配效率低下，并造成可用资源利用不足。尤其是当租户频繁加入或离开系统时，例如包含后台作业的应用程序（例如，RocksDB 压实），或服务会经历闲置期、随后又出现活动高峰的社交网络（例如，Twitter [13]），池优化器都无法及时、充分地作出响应。为解决这一问题，可以运用未命中率曲线（MRC）等技术，描述命中率、吞吐量或延迟随不同池大小的变化，从而改进预测机制 [15], [16]。

处理多个实例。CacheLib 实例彼此隔离，缺乏全局协调。因此，在多个实例之间处理动态工作负载更具挑战性，因为池优化器只能管理同一实例内的池，无法在实例之间重新分配内存，从而限制了整体资源效率。这种隔离也妨碍了服务质量保障的实施，

使关键租户优先于不太关键的租户。解决这一挑战需要在各实例之间建立协调的资源管理层，并采用服务质量感知策略；还可能需要借助集中式控制器，监控缓存使用情况，并在整个基础设施中执行针对特定租户的性能目标 [17], [18]。

相关工作与研究的普遍性。本文聚焦于 CacheLib，因为它已被广泛采用。尽管如此，已有多种其他缓存解决方案旨在应对多租户工作负载。Dynacache [19], Cliffhanger [20],AdaptSize [21],zExpander [14], FrozenHot [12], LAMA [22],AdaptCache [11], 和 SegCache [10] 着眼于通用缓存优化，例如通过采用工作负载监控、自适应调整大小和概率策略等不同策略，降低延迟、提高吞吐量、提升命中率或平衡内存使用。然而，这些方法并不考虑租户，因为其目标是最大化缓存性能，而非提供租户公平性和隔离；对于 CacheLib 所支持的多租户环境而言，这两者至关重要。这一局限性常导致激进的租户垄断缓存资源，并无意中驱逐其他租户的工作集，进而造成性能下降和服务质量违规 [9], [13]。

至于明确关注服务质量的系统，Moirai [3]、Pisces [9]、Centaur [15], 和 RobinHood [5]采用面向租户的管理技术，以实现特定的性能目标。例如，Centaur 使用模拟退火（SA）与未命中率曲线（MRC）相结合等优化算法，对缓存分区和跨多个租户的资源分配进行优化，以满足规定的性能约束 [15]。然而，这些系统通常针对特定工作负载和存储场景（如虚拟机监控程序和键值存储）进行了精细调优，因此难以推广到更广泛的多租户缓存环境，例如 CacheLib 所针对的环境。举例来说，专为块缓存设计的预测模型假定项大小均匀，但这一假设不适用于处理可变大小项的键值存储，因而会导致性能估算不准确 [15], [16]。

致谢

本研究由欧洲区域发展基金（ERDF）通过 Portugal 2030 旗下 NORTE 2030区域计划共同资助，资助范围为项目 BCD.S+M，项目编号 14436（NORTE2030-FEDER- 00584600）；另获葡萄牙科学与技术基金会（FCT - Fundaç˜ao para a Ciˆencia e a Tecnologia, I.P.）国家资金支持，资助编号 UID/50014/2023（https://doi.org/10.54499/UID/50014/2023）；并获 FCCN 在 Deucalion 项目范围内资助，项目编号为 2024.00014.TEST.DEUCALION。本研究还获得美国国家科学基金会（NSF）的资助（项目编号 CNS-1956229、CSR-2402328、CAREER-2338457 和 CSR-2323100），以及 NetApp 和希捷的慷慨捐赠。

参考文献

[1] Y. Feng、Z. Liu、Y. Zhao、T. Jin、Y. Wu、Y. Zhang、J. Cheng、C. Li 和 T. Guan，《通过分区同步扩展大型生产集群》，载于 2021 年 USENIX 年度技术会议，2021 年。[2] B. Berg、D. S. Berger、S. McAllister 等，《CacheLib 缓存引擎：设计与规模化实践》，载于第 14 届 USENIX 操作系统设计与实现研讨会，2020 年。[3] I. A. Stefanovici 、 E. Thereska、G. O'Shea、B. Schroeder、H. Ballani、T. Karagiannis、A. I. T. Rowstron 和 T. Talpey，《软件定义缓存：多租户数据中心中的缓存管理》，载于第六届 ACM 云计算研讨会论文集，2015 年。[4] （2021） CacheLib：脸书的开源缓存引擎。 [在线]。网址：https://engineering.fb.com/2021/09/02/open-source/cachelib/ [5] D. S. Berger、B. Berg、T. Zhu、S. Sen 和 M. Harchol-Balter，《Robin-Hood：感知尾延迟的缓存——从缓存资源充裕者向资源匮乏者动态重新分配》，载于第 13 届 USENIX 操作系统设计与实现研讨会，2018 年。[6] Y. Fu 和 C. Soman，《Uber 的实时数据基础设施》，载于 2021 年国际数据管理会议，2021 年。[7] N. Armenatzoglou 等，《重塑 Amazon Redshift》，载于 2022 年国际数据管理会议，2022 年。[8] T. B¨ottger 等，《Open Connect Everywhere：透过 Netflix 内容分发网络一窥互联网生态系统》，载于 ACM SIGCOMM《计算机通信评论》，2018 年。[9] D. Shue、M. J. Freedman 和 A. Shaikh，《多租户云存储的性能隔离与公平性》，载于第 10 届 USENIX 操作系统设计与实现研讨会，2012 年。[10] J. Yang、Y. Yue 和 R. Vinayak，《Segcache：面向小对象的高效内存、可扩展内存键值缓存》，载于第 18 届 USENIX 网络系统设计与实现研讨会，2021 年。[11] O. Asad 和 B. Kemme，《AdaptCache：分布式对象缓存的自适应数据分区与迁移》，载于第 17 届国际中间件会议论文集，2016 年。[12] Z. Qiu、J. Yang、J. Zhang、C. Li、X. Ma、Q. Chen、M. Yang 和 Y. Xu，《FrozenHot Cache：重新思考现代硬件的缓存管理》，载于第 18 届欧洲计算机系统会议，2023 年。[13] J. Yang、Y. Yue 和 K. V. Rashmi，《对 Twitter 数百个内存缓存集群的大规模分析》，载于第 14 届 USENIX 操作系统设计与实现研讨会，2020 年。[14] X. Wu、L. Zhang、Y. Wang、Y. Ren、M. Hack 和 S. Jiang，《zExpander：兼具高性能与低未命中率的键值缓存》，载于第 11 届欧洲计算机系统会议，2016 年。[15] R. Koller、A. J. Mashtizadeh 和 R. Rangaswami，《Centaur：主机侧固态硬盘缓存与存储性能控制》，载于 2015 年 IEEE 国际自主计算会议，2015 年。[16] C. A. Waldspurger、N. Park、A. T. Garthwaite 和 I. Ahmad，《使用 SHARDS 高效构建 MRC》，载于第 13 届 USENIX 文件与存储技术会议，2015 年。[17] R. Macedo、M. Miranda、Y. Tanimura、J. Haga、A. Ruhela、S. L. Harrell、R. T. Evans、J. Pereira 和 J. Paulo，《通过动态、应用无关的服务质量控制驯服元数据密集型高性能计算作业》，载于第 23 届 IEEE/ACM 集群、云与互联网计算国际研讨会。IEEE，2023 年。[18] M. Miranda、Y. Tanimura、J. Haga、A. Ruhela、S. L. Harrell、J. Cazes、R. Macedo、J. Pereira 和 J. Paulo，《当前的软件定义存储控制器能否适应现代高性能计算基础设施的规模？》，载于高性能计算、网络、存储与分析国际会议研讨会。IEEE，2024 年。[19] A. Cidon、A. Eisenman、M. Alizadeh 和 S. Katti，《Dynacache：动态云缓存》，载于第 7 届 USENIX 云计算热点问题研讨会，2015 年。[20]——，《Cliffhanger：扩展 Web 内存缓存中的性能悬崖》，载于第 13 届 USENIX 网络系统设计与实现研讨会，2016 年。[21] D. S. Berger、R. K. Sitaraman 和 M. Harchol-Balter，《AdaptSize：为内容分发网络协调热点对象内存缓存》，载于第 14 届 USENIX 网络系统设计与实现研讨会，2017 年。[22] X. Hu、X. Wang、Y. Li、L. Zhou、Y. Luo、C. Ding、S. Jiang 和 Z. Wang，《LAMA：面向键值缓存的优化局部性感知内存分配》，载于 2015 年 USENIX 年度技术会议，2015 年。