arXiv:2611.32653v1 [cs.DC] 24 Nov 2026

Rafał Frączek

Goose、C++ 和 Rust 中的并行内存管理：关于分配、回收与扩展性的实验研究

摘要

编译器管理的内存可以降低分配和回收开销，但其在并行工作负载下的表现仍缺乏充分研究。本研究评估了 *Goose v2026.3*，并将其与 *C++* 和 *Rust* 实现进行比较，其中包括采用单独分配节点和连续区域分配器的树变体。我们在一台配备四个物理核心和八个逻辑 *CPU* 的工作站上，使用多个工作线程对工作线程本地动态数组和二叉树进行了基准测试。重复的树实验包括五轮构建、遍历和回收。每种主要实验配置均进行了 *15* 次独立进程运行测量。我们测量了全进程执行时间、释放操作的时间间隔，以及同步检查点处的内存使用量。我们还考察了强扩展和弱扩展。对于深度为 *20* 的树，在使用八个工作线程时，*Goose* 的执行时间中位数分别比 *C++* 和 *Rust* 区域分配器低 *5.62%* 和 *10.95%*。在 *LIVE* 检查点处，其额外进程常驻集大小与区域分配器相当，且显著低于单独分配节点的树，但其虚拟内存大小要大得多。逻辑回收并不总能减少驻留内存，而且 *Goose* 的弱扩展效率低于 *C++* 和 *Rust* 区域分配器。动态数组的结果则喜忧参半。这些结果表明，对于对象生命周期可预测的并行工作负载，编译器管理的存储可以发挥良好作用。不过，单看执行时间无法体现其在内存和可扩展性方面的权衡。

关键词

内存管理、Goose 编程语言、C++、Rust、并行计算、内存区分配、强扩展与弱扩展

1. 动机

在现代计算机体系结构中，内存分配和回收会影响并行程序的性能与可扩展性，尤其是在创建和释放大量对象时。先前的研究表明，分配策略会影响执行时间和内存消耗 [1] [2]。新编程语言 Goose 使用编译器管理的数据栈和基于作用域的回收 [13]。然而，现有的 Goose 基准测试主要关注单线程工作负载[12]。其在并行分配和回收时的表现仍缺乏充分研究。C++ 使用资源获取即初始化、智能指针和标准容器来管理资源和对象生命周期 [14]。Rust 通过所有权和借用，为安全代码提供静态检查的内存安全保证，且无需垃圾回收器[8]。两种语言都支持单个对象分配和基于内存池的存储。这些方法在分配粒度、数据表示形式和回收行为方面各不相同。

在 Goose 编程语言中，内存分配会推进数据栈指针。恢复已保存的指针后，已分配存储即可重新使用。Goose 还支持紧凑相对引用和工作线程本地数据栈 [13]。基于区域分配器的分配和批量回收并非新技术 [15]。这里的关键在于 Goose 如何实现这些技术，以及它们在并行工作负载下的表现。以往的 Goose 基准测试 [12]并未充分涵盖并行扩展性、重复回收或释放后的内存使用量。此外，逻辑回收并不一定会将物理内存归还给操作系统。

本研究使用三种并行工作负载，将 Goose v2026.3 与 C++ 和 Rust 进行比较：动态数组、二叉树，以及重复的树分配和回收。C++ 和 Rust 的树实现包括单独分配的节点和经过优化的连续内存区。我们测量全进程执行时间、内存使用量和释放操作时长，同时评估强扩展和弱扩展。研究目标是确定托管存储在不同工作负载下的优势与局限。我们并不旨在证明某一种编程语言在所有情况下都优于其他语言。

本文的结构由以下各节组成。第 2 节将回顾相关技术，并概述研究空白。第 3 节将介绍研究问题和研究贡献。第 4 节将介绍实验方法，包括执行环境、基准测试和实验步骤。第 5 节将对实验结果进行比较讨论。第 6 节将突出说明主要发现和局限性。最后是第 7 节结论。

2. 引言

2.1. 内存管理的最新进展

内存管理方法会影响执行时间、进程内存使用量和并行可扩展性。内存分配方法需要在灵活性、分配成本和内存局部性之间取得平衡。专用策略并不总是优于通用分配器 [2]。C++使用资源获取即初始化和智能指针来管理资源生命周期 [14]。Rust 通过所有权和借用提供内存安全保证。RustBelt 项目为经过建模的 Rust 机制确立了安全性结果 [8]。然而，生命周期管理并不能决定分配策略或数据表示形式。两种语言都支持单独的内存分配，以及具有连续虚拟内存地址的容器。这并不意味着物理内存也是连续的。

基于区域的内存管理算法会将生命周期相容的对象分组，并一并释放。Tofte 和 Talpin [15] 奠定了其形式化基础。Grossman 等人 [5] 在 Cyclone 中引入了基于区域的分配，并提供静态安全保证和生命周期控制。这些方法可以降低单个对象的释放成本。然而，它们可能会延长对象的保留时间，并对对象生命周期施加限制。

内存池和区域分配器提供了相关的分配策略。Lattner 和 Adve[9] 研究了由编译器驱动的分配和数据布局。紧凑的数据布局可以降低分配开销并改善内存局部性。然而，集体回收会限制单个对象的释放能力。

并行内存管理带来了额外挑战，包括同步、分配器竞争和跨线程释放。Hoard 分配器旨在解决多线程程序中的内存利用率和可扩展性问题 [1]。Evans 为可扩展的并发分配引入了 jemalloc [3]。jemalloc 使用多个区域分配器来减少竞争。mimalloc 分配器采用页局部空闲链表分片来改善分配局部性 [10]。危险指针支持对共享对象进行安全回收 [11]。与之不同，我们的基准测试使用工作线程局部数组和二叉树。这些数组的元素和树的节点不会在工作线程之间共享，不过仍然需要同步和通信。

近期研究也凸显了工作负载特征的重要性。Zhou 等人 [16] 研究了对象大小和生命周期对 TCMalloc 性能的影响。Hughes 和 Tratt [7] 研究了 Rust 中的垃圾回收，包括一项基于区域分配器的二叉树基准测试。He 等人 [6]提出了一种支持更灵活区域生命周期的类型系统。这些研究表明，分配性能取决于数据表示、对象生命周期和工作负载特征。

现有研究探讨了内存管理的不同方面，但没有任何单一策略适用于所有工作负载。本文回顾的研究并未直接比较 Goose 的工作线程局部存储与经过优化的 C++ 和 Rust 区域分配器在并行执行下的表现。尤其是，逻辑回收、

进程内存使用量与并行扩展性之间的关系，在这一比较中仍未得到充分表征。

2.2. 研究空白与研究依据

现有的 Goose 基准测试套件比较了使用 Goose、C++ 和 Rust [12] 实现的多种数据结构。其中的二叉树基准测试包括单独分配的节点和优化的内存 arena。该套件还考察了重复的树构建与销毁。然而，这些基准测试主要关注单线程执行时间和峰值内存消耗，并未说明性能和内存使用量如何随工作线程数变化；此外，也没有测量重复分配与回收过程中同步检查点处的内存情况。

仍有三个问题尚待回答。第一，在并行工作负载下，Goose 与经过优化的 C++ 和 Rust 存储方案相比表现如何？第二，逻辑释放如何影响进程的内存使用量？具体而言，驻留内存与虚拟地址空间有何差异？第三，在强扩展和弱扩展条件下，性能如何变化？

我们通过动态数组、二叉树，以及重复的树构建与释放来研究这些问题。我们测量进程执行时间、释放持续时间，以及同步检查点处的内存情况。我们的目标是识别其中的权衡，而不是证明某一种语言始终更快。

3. 研究目标

3.1. 研究问题

本研究旨在评估 Goose v2026.3 的性能和内存管理行为。研究针对包含并行计算的工作负载，对 Goose、C++ 和 Rust 的实现进行比较。研究重点关注内存分配方法、对象生命周期问题以及并发计算的影响。为了明确研究工作的开展方向，本文提出了以下研究问题。

问题 1。就执行时间和内存使用量而言，Goose 的内存存储方式与采用逐对象分配和内存区分配技术的方法之间有何差异？

该研究问题涉及与处理动态数据结构的各种方式相关的计算成本和内存成本。尤其是，它区分了单个对象分配与基于内存区的内存管理。实验包括动态数组和二叉树。动态数组和二叉树均具备并发计算能力。

问题 2。不同实现之间的释放时长有何差异？逻辑回收如何影响进程驻留集大小（RSS）？

本问题考察逻辑内存回收的成本。我们还研究回收是否会降低进程常驻集大小。测量涵盖内存

分配前、对象仍保持分配状态时以及逻辑回收后的使用量。我们比较了不同实现之间的这些结果。

问题 3：在强扩展和弱扩展下，执行时间如何随工作线程数变化?

本问题考察执行时间如何随工作线程数变化。强扩展保持总工作负载不变，而弱扩展则按工作线程数的比例增加总工作负载。两种情形均使用并行二叉树工作负载进行评估。

3.2. 贡献

基准测试比较。我们将先前的 Goose 基准测试 [12] 扩展到并行工作负载和重复内存回收。我们比较 Goose 与两种用 C++和 Rust 实现的树表示形式。第一种使用单独分配的节点，在 C++中由 unique_ptr 管理，在 Rust 中则由 Box 管理。第二种将紧凑节点存储在 vector 或 Vec arena 中。

内存测量。我们使用四个同步检查点：树构建前的 BASE、树分配期间的 LIVE、释放后的 ROUND，以及工作线程终止后的 FINAL。我们测量进程常驻集大小和虚拟内存大小 (VMS)。逻辑释放不一定会减少驻留内存。

释放与扩展性测量。我们将构建和遍历一并测量。我们单独计时释放操作，并将这些时间间隔中的同步等待排除在外。我们还评估强扩展和弱扩展。

可复现性。每种配置的主要实验均执行 15 次独立进程执行。我们验证了所有实现的校验和与节点数。我们保留了源文件、执行脚本和测量记录，以支持复现实验结果。

4. 实验方法

4.1. 实验环境与实现

实验使用的硬件配置为 Intel Core i3-14100 CPU，配备四个物理核心和八个逻辑处理器，以及 32 GiB RAM。需要指出的是，所用操作系统为 Linux，内核版本为 7.2.9。

实验使用了 Goose v2026.3、GCC 16.2.1 和 Rust 1.98.0（edition 2021）。Goose 和 C++ 均使用 -O2 进行编译，而 Rust 使用 -C opt-level=2。重复回收基准测试使用了 C++20。动态数组和独立树基准测试使用了 C++17。C++ 构建还使用了 -DNDEBUG 和 -pthread。两个树基准测试均使用了 Rust 选项 -C codegen-units=1。动态数组基准测试未指定此选项。重复回收的元数据记录了 Goose 版本以及 Goose 编译器可执行文件的 SHA-256 哈希值。

实验使用了 1、2、4 和 8 个工作线程。对于最多使用四个工作线程的配置，进程的亲和性被限制在逻辑 CPU 0、2、4 和 6 上。这些逻辑 CPU 分属于四个不同的物理核心。八工作线程配置可以使用全部八个逻辑 CPU。各个工作线程未绑定到特定核心。线程放置由 Linux 调度器控制。

值得注意的是，这三种实现采用了不同的内存分配策略。具体而言，Goose 语言栈采用了按工作线程管理的数据结构。C++和 Rust 树实现使用了两种可选的存储策略。单独分配的节点分别使用 C++unique_ptr 和 Rust Box 进行管理。连续内存区使用 C++ vector 和 Rust Vec 实现，并采用 32 位子节点索引。这种设置有助于比较传统的对象分配技术与紧凑内存结构分配策略。

在紧凑树表示中，每个节点包含一个 32 位值，以及两个 32 位子节点引用或索引。C++ 和 Rust 区域分配器使用子节点索引，而 Goose 使用可空自相对引用。由此得到的节点表示占用 12 字节。

在重复回收基准测试中，每个 Goose 工作线程在每一轮使用一个可增长池。C++ 和 Rust 区域分配器的每个变体，在每个工作线程的每一轮中使用一个预留容量向量。单独分配的变体为每个节点创建一次单独的分配。所有树都会一直保持分配状态，直到 LIVE 检查点。

工作线程不会交换数组元素或树节点，而是将结果和控制信息传递给主线程。在树实验中，Goose 使用类型化消息队列。C++ 使用屏障和共享结果记录。Rust 使用屏障和通道。在动态数组基准测试中，Goose 工作线程通过队列报告部分和。C++ 工作线程将数据写入各自独立的结果槽位，而 Rust 工作线程则通过线程句柄返回部分和。

所有测试的源代码均可在公共代码仓库 [4]中获取。

4.2. 基准测试负载

性能评估涵盖了三种工作负载，以评估分配效率、数据结构遍历效率和内存回收。实验采用不同的并行度，以衡量可扩展性。

动态数组基准测试共处理了 6400 万个 64 位整数。每个工作线程拥有一个独立数组，其中包含数量相等的元素。每个工作线程都会填充并遍历自己的数组。Goose 数组会动态增长。C++和 Rust vector 则使用自动增长或显式容量预留。

二叉树实验使用了深度为 18 和 20 的完全树。深度为 d 的树包含 $2^{d+1}-1$ 个节点，根节点的深度为零。每棵树都遍历了八次，并使用确定性校验和验证计算结果。每个工作线程都有自己的一组数据结构；工作线程之间不会交换树节点。重复回收基准测试使用持久工作线程，连续进行五轮构建、遍历和释放。独立

树基准测试则逐棵构建和释放树。重复回收基准测试在释放前后使用了同步的内存检查点。

我们选择完全二叉树，因为其大小和生命周期是可预测的。它们还使我们能够考察独立节点分配的影响。这是一种受控工作负载，并非所有树应用的模型。B 树和 B+ 树具有不同的节点布局和访问模式。

我们评估了两种扩展场景。对于强扩展，将 8 棵树均匀分配给 1、2、4 和 8 个工作线程。对于弱扩展，每个工作线程处理一棵深度为 18 的树。

4.3. 测量流程与指标

外部启动器使用 CLOCK_MONOTONIC 测量进程总时间。计时从进程创建之前开始，并在 wait4() 函数返回后结束。其中包括初始化、计算、同步、内存采样和终止。工作线程计时器测量树构建和遍历的总时间。释放计时在 LIVE 同步等待之后开始，不包括等待本身。

我们通过 /proc/self/smaps_rollup 测量进程常驻集大小，并通过 /proc/self/statm 测量虚拟内存大小。BASE 在工作线程启动后、树构建之前测量。LIVE 在树保持分配状态期间测量。ROUND 在逻辑释放之后测量。FINAL 在工作线程终止后测量。通过 wait4() 函数获得的峰值常驻集大小单独进行了分析。

我们从释放后驻留集大小比较中排除了最后一轮，因为工作线程终止可能会影响 Goose 快照。附录 A 给出了更多测量细节。

4.4. 统计分析与可复现性

最初，回收实验由 15 次重复组成，每种配置均包含五个内存分配与回收周期。这意味着每次进程执行都被视为一次独立观测，而单次试验中的多次运行不予单独计入。初步冒烟测试阶段获得的结果不纳入后续分析。重复回收基准测试为每种配置执行两次预热执行。独立树基准测试执行三次。这些预热执行均从分析中排除。在测量运行期间，各实现的执行顺序采用随机化处理。

执行时间汇总包括中位数和中位数绝对偏差（MAD）。为了测量回收间隔，首先确定所有工作线程每轮花费的最大时间。对于每一轮，计算所有工作线程中释放操作的最大执行时间。对于每次进程执行，我们从存活和 ROUND 测量值中减去其 BASE 常驻集大小。我们计算了五个 LIVE 检查点的中位数，以及前四个

ROUND 检查点的中位数。随后，将 15 次独立进程执行的结果计算为中位数。

系统性能评估依据执行时间中位数之比进行。强扩展评估采用固定的总工作负载。加速比计算为 $S(p) = T(1)/T(p)$。其中，$T(p)$ 表示 p 个工作线程的执行时间中位数。弱扩展评估采用每个工作线程固定的工作负载。归一化执行时间计算为 $N(p) = T(p)/T(1)$。总工作负载随工作线程数成比例增加。在这两种情况下，执行时间均指整个进程的执行时间。性能数值之间的差异是在未进行统计显著性检验的情况下估算的。此外，使用 20000 次自助法重采样，对选定的比较进行了分析，重采样基于进程级独立测量值，置信区间采用百分位数法计算。所得的 95% 置信区间用于描述抽样不确定性。

5. 实验结果与比较

5.1. 速度与强扩展比较

我们使用动态数组、独立二叉树以及重复的树构建与回收，对并行执行进行了评估。评估重点比较了 Goose 与单独分配的对象、连续内存区在 C++ 和 Rust 中的表现。

图1展示了树深度为18和20时，二叉树重复回收基准测试的执行时间中位数结果。每次基准测试运行在每次迭代中处理八棵树，其中包括五次构建和回收迭代，以及八次遍历。数据通过对每种进程实现分别进行15次运行收集。Goose 在测试的五种实现中取得了最快的执行时间中位数。

树深度为20、由单个工作线程执行时，Goose 的执行时间为1098毫秒，而 C++ 向量竞技场和 Rust 向量竞技场分别为1186毫秒和1401毫秒。使用八个工作线程时，执行时间分别降至296毫秒、313毫秒和332毫秒。使用八个工作线程时，与优化后的 C++ 竞技场相比，Goose 将执行时间中位数降低了5.62%；与 Rust arena 相比，则降低了10.95%。95%自助法百分位置信区间分别为4.90–6.33%和10.31–11.20%。这些估计基于每种配置下15次独立进程执行。

逐个分配对象时，性能大幅下降。对于深度 20 和八个工作线程，C++ unique_ptr 实现的运行时间为 1106 毫秒，Rust Box 的运行时间为 1144 毫秒，这意味着它们的运行时间分别约为 Goose 的 3.74 倍和 3.87 倍。Goose 与优化的内存 arena 之间差异相对较小，这表明分配粒度和表示形式对性能差异影响很大。

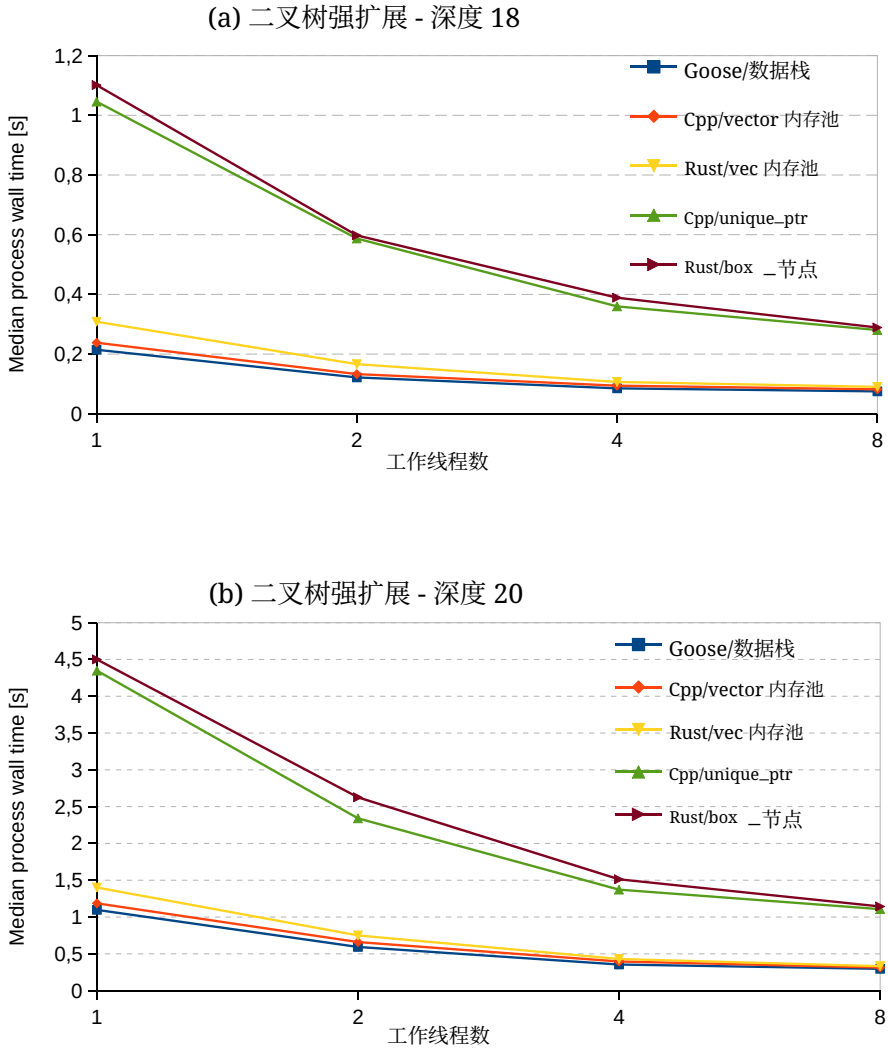

图1. 强扩展下深度为 (a) 18 和 (b) 20 的二叉树工作负载的进程墙钟执行时间中位数。每轮处理 8 棵树，共进行 5 轮，每轮包含 8 次遍历。结果代表每种配置下 15 次独立进程执行。

强扩展加速比根据执行时间计算，执行时间如图1所示。每个加速比均相对于单个工作线程对应的执行时间进行测量。在深度为 20 时，Goose 的加速比达到 3.71

在八个工作线程下，相比之下，C++ 区域分配器的加速比为 3.78，Rust arena 的加速比为 4.22。Goose achieved the shortest execution time。不过，其加速比低于 C++ 和 Rust 区域分配器。因此，Goose 未实现最高的强扩展效率。

独立树基准测试采用了不同的执行策略。每个工作线程依次构造、遍历并释放其树。相比之下，重复回收基准测试会让所有分配的树一直存活到 LIVE 检查点。因此，这两个基准测试的执行时间无法直接比较。对于深度为 20 的二叉树，在八个工作线程下，Goose 耗时 62 毫秒，而 C++区域分配器和 Rust arena 分别耗时 67 毫秒和 69 毫秒。结果遵循了重复回收基准测试所得的排序。动态数组基准测试的执行时间中位数汇总于表 1。

对于动态数组基准测试，得到了一种更加均衡的比较结果。使用四个工作线程处理 6400 万个 64 位整数时，Goose 耗时 40.05 毫秒，C++ vector 耗时 39.53 毫秒，Rust vector 耗时 40.76 毫秒。使用八个工作线程时，Goose 耗时 43 毫秒，而 C++vector 和 Rust vector 分别耗时 42 毫秒和 46 毫秒。因此，Goose 仍具有竞争力，但并不总是优于优化向量，这凸显了容量和工作负载因素的重要性。

重复回收的强扩展分析涵盖了 40 种配置。其相对中位数绝对偏差约为 0.21% 至 3.97%。总体而言，这表明不同运行之间的变异性较低。全进程执行时间结果证实，在所考察的案例中，Goose 具有竞争力；但不同的存储表示、分配策略、编译器优化和同步方式，使得无法将任何优势单独归因于语言特性。

表 1动态数组基准测试：包含 6400 万个 64 位整数时的中位全进程执行时间（毫秒）。

实现	1 个工作线程	2 个工作线程	4 个工作线程	8 个工作线程
Goose 栈增长	92.86	53.35	40.05	42.86
C++ 向量增长	143.28	99.45	80.50	81.78
C++ 向量预留	92.69	52.97	39.53	42.09
Rust vector 增长	92.41	57.42	44.31	48.26
Rust vector 预留	88.00	53.18	40.76	46.47

5.2. 内存消耗

程序的内存使用量基于多次内存回收过程中收集的同步进程级指标进行评估。图2所示结果表示存活检查点处常驻集（RSS）的额外大小，其计算方式是减去相应的BASE RSS测量值。

对于包含八个工作线程的深度为 20 的树，Goose 实现相对于 BASE 增加了 197 MiB 的进程常驻集大小。相应地，C++ arena 和 Rust arena 的数值均为 192 MiB，如图2所示。相比之下，单独分配的 C++ unique_ptr 和 Rust Box 节点分别占用约 512 MiB 和 512 MiB，约为相应基于 arena 的实现额外常驻集大小的 2.67 倍。对于包含八个工作线程的深度为 18 的树，Goose 增加了约 53 MiB 的常驻集大小。C++ 和 Rust arena 使用了约 48 MiB。单独分配的实现使用了约 128 MiB。

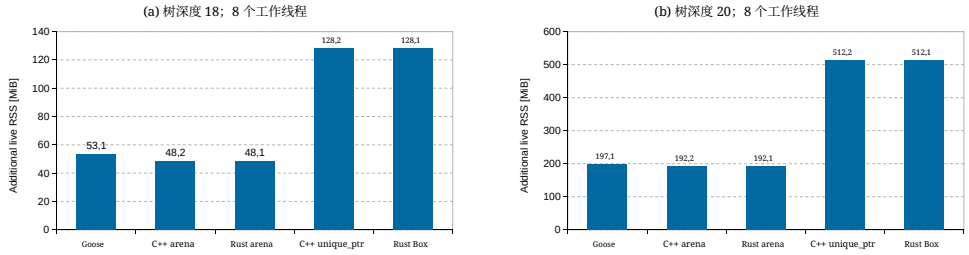

图2. 二叉树工作负载在八个工作线程下，于同步的 LIVE 检查点测得的额外驻留集大小（RSS）：(a) 深度 18 和 (b) 深度 20。

图3a比较了逻辑回收前后的进程常驻集大小。分析涵盖了前四轮。对于八个工作线程的深度为 20 的树，释放后，Goose 相较于基线仍保留了约 197 MiB 的额外 RSS。C++ 和 Rust 区域分配器各自保留了约 192 MiB，而 C++ unique_ptr 保留了约 512 MiB。相比之下，释放后 Rust Box 的额外 RSS 约为 65 MiB。使用一个工作线程时，竞技场实现将 RSS 降至接近基线的数值。

虚拟地址空间消耗的测量结果揭示了所研究编程语言之间另一个有趣的区别。图3b显示了不同工作线程数量下进程的虚拟内存大小（VMS）总量。对于 Goose，VMS 从一个工作线程时的约 10.08 GiB 增至八个工作线程时的 24.52 GiB。这些较大的 VMS 数值在树构建之前的 BASE 检查点处就已出现。其他语言的数值则低得多；在八个工作线程的配置下，约为 0.48 至 1.01 GiB。

Goose 的驻留内存消耗与连续内存区接近。然而，它预留的虚拟地址空间明显更多。这些预留空间并未对应等量的物理常驻内存。根据所呈现的结果，可以对以下三个不同方面得出结论：对象表示、驻留内存消耗以及虚拟地址空间需求。

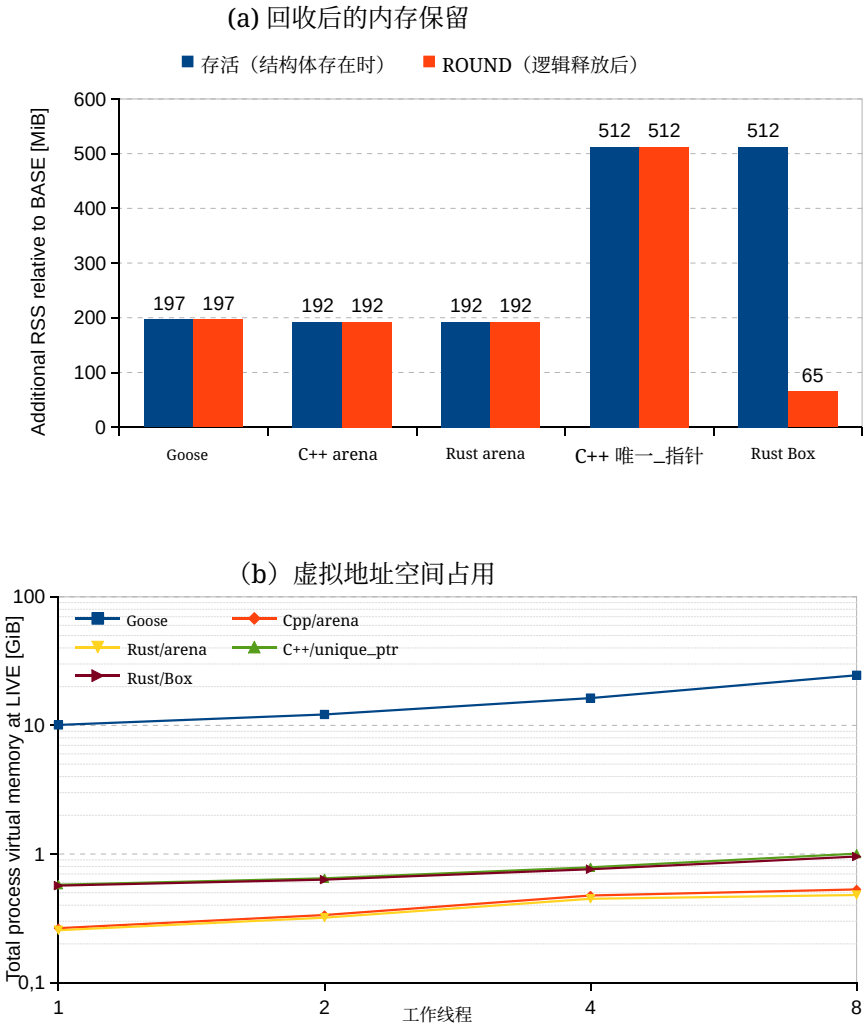

图 3. 经评估的实现的内存管理行为：(a) 八个工作线程处理深度为 20 的二叉树时，逻辑回收前后的基线调整后驻留集大小；(b) LIVE 检查点处的虚拟内存总大小随工作线程数的变化。结果为 15 次独立执行的中位数。释放后的 RSS 值不包括最后一轮。

5.3. 回收性能

对回收算法性能的评估包括测量执行存储释放操作所需的时间。我们注意到，经过修正的

检测机制未将 LIVE 同步屏障阶段产生的等待时间纳入考量。在每一轮中，记录工作线程完成释放操作所需的最长时间。每次独立进程运行均重复 15 次，并分别收集五轮的中位数统计数据。

分别计算树在每个深度下的释放间隔。图 4 展示了深度为 20 的树的释放间隔。参与计算的工作线程数为 8 时，C++ unique_ptr 和 Rust Box 的工作线程最大释放间隔中位数分别为 44 毫秒和 51 毫秒。与此同时，连续 C++ arena 和 Rust arena 的释放间隔分别约为 0.9 秒和 0.8 μ 秒。测得 Goose 库的作用域退出延迟约为 0.2 μ 秒。深度为 18 的树的定性比较结果与此类似。

释放间隔还取决于参与工作的工作线程数。当深度为 20 且只有一个工作线程处于活动状态时，C++和 Rust arena 分配器的释放间隔约为 242 μ 秒。当同时有 8 个工作线程工作时，释放间隔显著缩短，约为 1 微秒。不同工作线程配置下，总工作负载保持不变。然而，随着工作线程数增加，分配给每个工作线程的工作负载会减少。因此，每个工作线程释放的内存量也会发生变化。这些观察结果与释放后 RSS 的差异相对应。

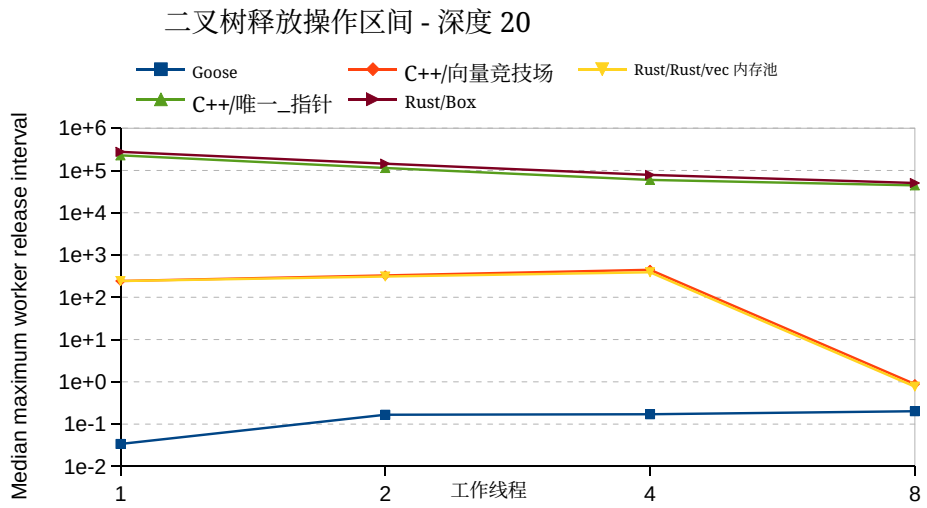

图 4。 深度为 20 的树的工作线程最大释放间隔中位数（单位：微秒）。纵轴采用对数刻度。

不同实现的释放操作各不相同。Goose 会恢复编译器维护的栈位置。C++ 和 Rust arena 变体会释放其向量存储。单独分配的变体会递归销毁树节点，并回

收其内存分配。Goose 的释放间隔极短。测量值可能受到时钟调用开销和分辨率的影响。我们没有单独校准这一开销。因此，不应将这些结果解读为物理内存回收的精确测量值。

这些实验表明，当对象的生命周期兼容时，集体释放的开销可能很低。不过，不同实现测得的释放操作耗时有所不同。Goose 最短的时间间隔也容易受到时钟测量开销的影响。研究结果表明，评估释放操作时必须辅以内存保留估计。

5.4. 弱扩展行为

测试使用深度为 18 的完全二叉树进行，每个工作线程负责创建一棵树。因此，总工作量随工作线程数线性增加。每项实验均进行五个周期，每个周期包含对已构建并回收的二叉树进行八次遍历。测试分别使用 1、2、4 和 8 个工作线程，每种配置独立重复 15 次。我们使用归一化执行时间 $T(p)/T(1)$ 评估弱扩展。比率为 1 表示达到理想的弱扩展，比率越小越好。我们还计算了弱扩展效率 $T(1)/T(p)$。图 5 展示了上述五种实现的归一化执行时间。基于 Goose 的实现，在所有工作线程数下的绝对执行时间均最短。使用 1 个工作线程时，执行时间为 28 ms；使用 2、4 和 8 个工作线程时，执行时间分别为 32 ms、48 ms 和 75 ms。以 C++ 实现的区域分配器会使执行时间从 32 ms 增至 82 ms，而以 Rust 实现的区域分配器在 8 个工作线程时的执行时间则达到 90 ms。使用 8 个工作线程时，逐个分配内存的 C++ 和 Rust 树实现分别需要 283 ms 和 287 ms。就相对性能下降而言，Goose 的相对增长幅度最大，为 2.68。其他情况下执行时间的相对增长幅度分别为 2.55、2.21、2.16 和 2.13，对应 C++ 区域分配器、Rust arena、C++ unique_ptr 和 Rust Box。弱扩展效率分别为 37.3%、39.2%、45.3%、46.4% 和 47.0%。这意味着，即使某些实现的绝对执行时间较短，其扩展效率也不一定优于其他实现。

我们使用独立进程级测量的 20000 次自助法重采样评估抽样的不确定性，并采用百分位数法估计置信区间。所得的 95% 置信区间描述了测试条件下归一化执行时间估计值的精确度。图 5 显示了使用 1 至 8 个工作线程时的归一化执行时间。对于 Goose，归一化执行时间的 95% 置信区间为 2.65 至 2.78；对于 C++arena，该区间为 2.46 至 2.63；对于 Rust arena，该区间为 2.14 至 2.30。图中展示的区间反映了抽样变异所造成的影响。

在弱扩展实验中，从 4 个工作线程增加到 8 个工作线程时，观察到执行时间显著增加。考虑到可用的 4 个核心以及 CPU 提供的 8 个逻辑处理器，最后一种情况意味着采用了同时多线程模式。目前的分析尚未给出同时多线程、内存带宽限制和同步开销各自影响的区分信息。

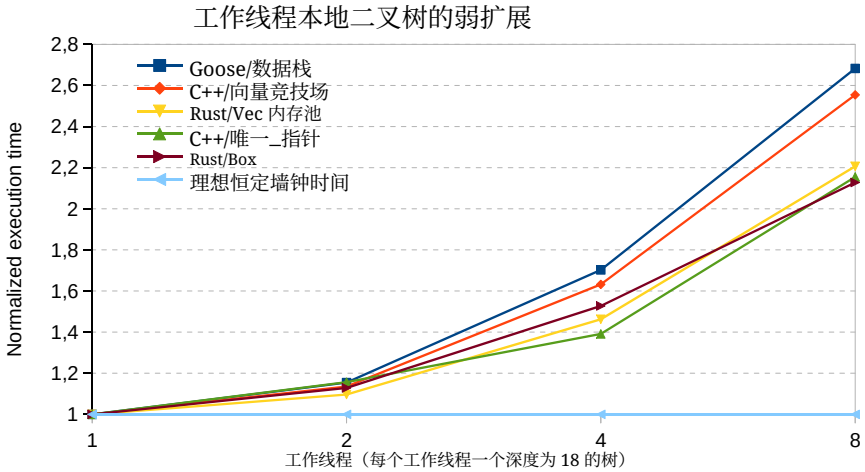

图 5。 深度为 18 的二叉树在弱扩展下的归一化中位执行时间，每个工作线程对应一棵树。每种配置包含五轮，每轮执行八次遍历。结果表示 15 次独立进程执行的中位数。理想弱扩展对应于恒定的归一化执行时间 1。

表 2 汇总了弱扩展执行时间以及归一化的八比一工作线程比值。

表 2各实现的弱扩展性能。执行时间以毫秒（ms）为单位报告。T(8)/T(1) 比值为无量纲。

实现	1 个工作线程	2 个工作线程	4 个工作线程	8 个工作线程	T(8)/T(1)
Goose	27.99	32.31	47.64	75.08	2.68
C++向量竞技场	32.13	36.48	52.45	82.07	2.55
Rust Vec 分配区	40.71	44.63	59.53	89.82	2.21
C++ 唯一 _ptr	131.19	151.72	182.52	282.72	2.16
Rust Box	135.02	152.37	206.20	287.33	2.13

所有五种实现的执行时间都随着工作线程数的增加而延长。没有一种实现达到了理想的弱扩展。

6. 讨论

6.1. 主要发现讨论

根据在二叉树上获得的实验结果，研究表明，托管存储的性能高度依赖于工作负载参数、存储表示形式和并行计算特性。就执行时间而言，与 C++ 和 Rust arena 实现相比，Goose 达到的数值最低。不过，与单独分配的节点相比，它展现出了显著的性能优势。此前的研究还表明，分配策略和数据结构的布局会影响性能 [2] [9]。这支持将优化的竞技场实现用作比较基线。

关于包含八个工作线程的深度为 20 的测试用例，Goose 将基准测试的中位执行时间相较于 C++ arena 降低了 5.62%，相较于 Rust arena 降低了 10.95%。考虑到所获得的 95% 自助法百分位数置信区间，可以得出结论：Goose 具有正向差异，尤其是考虑到进程执行次数较少所导致的变异性。不过，应谨慎解读这些发现，因为这些优势取决于特定的工作负载。在单一硬件架构上开展的实验无法揭示这些优势是否能够推广到其他架构。

实验结果表明，由编译器按每个工作线程管理的存储方式相较于数据结构具有竞争力。具体而言，在二叉树实验中，Goose 的速度快于高度优化的 C++和 Rust arena。然而，在动态数组基准测试中并未观察到这种优势。

关于内存消耗指标，Goose 的常驻集大小与连续内存区的数值相当，并且显著小于单独分配的节点情况下的数值。值得注意的是，观察到的差异主要源于所采用的数据存储策略，不能完全归因于分配方法。具体而言，紧凑的基于竞技场的节点避免了与单独内存分配相关的开销。Goose 的进程虚拟内存大小明显大于其他实现。使用八个工作线程时，虚拟内存大小约为 24.52 GiB。该数值在树构建之前就已存在。虚拟地址的预留应从资源使用的角度加以考虑。

回收实验显示，释放操作的耗时存在显著差异。Goose 的测量间隔最短。C++ 和 Rust arena 实现也支持批量释放，但采用了不同的机制。解读这些计时结果时应谨慎，因为最短的时间间隔容易受到时钟测量开销的影响。此外，释放的内存往往仍驻留在内存中。

并行扩展引入了另一个影响实验中基准测试整体性能的因素。Goose 达到了最短的绝对执行时间。然而，在工作负载的

强并行扩展过程中，它并未表现出始终如一的最高效率。关于弱扩展，Goose 在五个比值中显示出最高的归一化执行时间（2.68），而较低的数值更优。C++arena 基准测试的比值为 2.55，Rust arena 的归一化执行时间为 2.21。八工作线程配置使得能够在四核处理器上使用同时多线程。

本研究在以往的 Goose 基准测试 [12] 基础上，增加了对并行性能和内存回收的测量。结果表明，对于所测试的工作负载，编译器管理的工作线程本地存储具有竞争力。然而，评估时还应考虑驻留内存、虚拟地址空间和扩展效率。

6.2. 局限性

这些实现执行了等效的计算，并通过节点数和校验和进行了验证。然而，它们的数据布局、分配方法和同步机制各不相同。C++ 和 Rust 区域分配器使用带有 32 位子节点索引的紧凑节点。其他 C++ 和 Rust 树变体则使用指针，并为节点单独分配内存。因此，不能仅将计时差异归因于语言设计或分配开销。

我们使用完全二叉树来考察单独分配节点的成本。这些树并不能代表所有应用。B 树和 B+ 树可能会产生不同的结果。我们没有测量缓存未命中、缓存命中率或内存访问延迟。因此，局部性差异可能是造成观测到的执行时间差异的因素之一。

整个进程执行时间包括初始化、计算、同步和终止。RSS 和 VMS 是在检查点处采样的，而非持续监控。测得的释放间隔对应于特定实现的操作，而非物理内存归还给操作系统的过程。Goose 极短的作用域退出间隔可能受到时钟测量开销和分辨率的影响，这些因素并未经过校准。

进程总执行时间还包括收集内存检查点的开销。这项开销未单独测量，可能影响结果，尤其是对于较短的工作负载。

在 Goose 的最后一轮中，工作线程可能会在 ROUND 内存快照期间终止。因此，释放后驻留集大小比较仅纳入前四轮。

所有实验均在配备 Intel Core i3-14100 处理器的工作站上进行。进程亲和性限制在选定的逻辑 CPU 上，但各个工作线程未绑定到特定核心。调度、同时多线程、CPU 频率变化以及资源争用都可能影响结果。

主要实验针对每种配置进行了 15 次独立进程执行。独立的深度为 18 的试点实验进行了五次执行。这些测量支持描述性统计和自助法置信区间，但并不能证明在其他系统上的性能表现。基准测试使用了合成工作线程本地数组

和二叉树。未对共享数据结构、不规则工作负载以及其他硬件架构进行评估。

7. 结论

本研究考察了 Goose v2026.3 在并行工作负载中的性能和内存行为。我们将 Goose 与采用单个对象分配和优化连续内存区的 C++ 和 Rust 实现进行了比较。实验涵盖动态数组和二叉树。我们评估了执行时间、内存消耗、释放操作时长和扩展性。

在深度为 20、包含八个工作线程的重复回收基准测试中，Goose 的执行时间中位数比 C++ 区域分配器低 5.62%，比 Rust arena 低 10.95%。在 LIVE 检查点，Goose 增加的常驻集大小与两种区域分配器相当，且显著低于单独分配的树。不过，Goose 的进程虚拟内存大小要大得多，约达 24.52 GiB。Goose 的作用域退出间隔也非常短，但时钟测量开销限制了测量精度。逻辑回收并不总能降低进程常驻集大小。

尽管 Goose 在重复回收实验中的执行时间较短，但其弱扩展效率低于 C++ 和 Rust 区域分配器。动态数组测试的结果也喜忧参半。这些发现表明，编译器管理的工作线程局部存储所带来的优势取决于工作负载和性能指标。它们也凸显了区分逻辑内存回收与驻留内存变化的重要性。

实验仅限于单台工作站上的合成工作线程负载。数据表示形式和同步方面的差异也会影响比较结果。未来的工作应考察更大规模的多核系统，以及包含共享或不规则数据结构的应用程序。还应研究 B 树、B+ 树和缓存行为。另一个有价值的实验是比较 C++ 和 Rust 区域分配器，观察它们在轮次之间保留容量的情况。这有助于评估重复使用已分配存储的影响。总体而言，评估并行内存管理需要综合考量执行时间、驻留内存、虚拟地址空间和内存回收行为。

参考文献

[1]Berger E.D., McKinley K.S., Blumofe R.D., Wilson P.R.：《Hoard：面向多线程应用程序的可扩展内存分配器》。载于：L. Rudolph、A. Gupta 编，第九届编程语言与操作系统体系结构支持国际会议 *ASPLOS-IX* 会议录，美国马萨诸塞州剑桥，*2000*年*11*月*12*日至*15*日，第117–128页。ACM Press，2000年。网址：http://dx.doi.org/10.1145/378993.379232。

[2]Berger E.D., Zorn B.G., McKinley K.S.：重新思考定制内存分配。载于：M. Ibrahim、S. Matsuoka 编，*2002*年 *ACM SIGPLAN* 面向对象编程系统、语言与应用会议录，*OOPSLA 2002*，美国华盛顿州西雅图，*2002*年*11*月*4*日至*8*日，第1–12页。ACM，2002年。网址：http://dx.doi.org/10.1145/582419.582421。[3] Evans J.：面向 FreeBSD 的可扩展并发 malloc(3) 实现。载于：*BSDCan 2006*。2006年。网址：https://papers.freebsd.org/2006/bsdcan/ evans-jemalloc/。[4] Frączek R.：测试源代码，2026年。网址：https://bitbucket.org/rafalfr/goosetests/commits/e3454aa9df57492371c89561a46e596821c3c805。提交 e3454aa；在线访问；访问日期：2026年11月2日。[5] Grossman D., Morrisett J.G., Jim T., Hicks M.W., Wang Y., Cheney J.：Cyclone 中的基于区域的内存管理。载于：J. Knoop、L.J. Hendren 编，*2002*年 *ACM SIGPLAN* 编程语言设计与实现会议（*PLDI*）会议录，德国柏林，*2002*年*6*月*17*日至*19*日，第282–293页。ACM，2002年。网址：http://dx.doi.org/10.1145/512529.512563。[6]He S., Jia S., Bao Y., Rompf T.：当生命周期获得释放：具有高阶可达性跟踪的面向区域的类型系统。载于：*ACM* 程序语言会议录，第10卷（OOPSLA1），第1486–1513页，2026年。网址：http://dx.doi.org/10. 1145/3798254。[7]Hughes J., Tratt L.：Rust 的垃圾回收：终结器前沿。载于：*ACM* 程序语言会议录，第9卷（OOPSLA2），第3588–3614页，2025年。网址：http: //dx.doi.org/10.1145/3763179。[8] Jung R., Jourdan J., Krebbers R., Dreyer D.：RustBelt：保障 Rust 编程语言基础。载于：*ACM* 程序语言会议录，第2卷（POPL），第66:1–66:34页，2018年。网址：http://dx.doi.org/10.1145/3158154。[9]Lattner C., Adve V.S.：自动池分配：通过控制堆中的数据结构布局来提升性能。载于：V. Sarkar、M.W. Hall 编，*ACM SIGPLAN 2005* 年编程语言设计与实现会议录，美国伊利诺伊州芝加哥，*2005*年*6*月*12*日至*15*日，第129–142页。ACM，2005年。网址：http://dx.doi.org/10.1145/1065010.1065027。[10]Leijen D., Zorn B., de Moura L.：Mimalloc：空闲链表分片的实践。载于：A.W. Lin 编，编程语言与系统——第十七届亚洲研讨会，*APLAS 2019*，印度尼西亚巴厘岛努沙杜瓦，*2019*年*12*月*1*日至*4*日，会议录，计算机科学讲义，第11893卷，第244–265页。Springer，2019年。网址：http://dx.doi.org/10.1007/978-3-030-34175-6_13.[11]Michael M.M.：危险指针：无锁对象的安全内存回收。载于：*IEEE* 并行与分布式系统汇刊，第15卷（第6期），第491–504页，2004年。网址：http://dx.doi.org/10.1109/TPDS.2004.8。[12]van Oortmerssen W.：Goose 基准测试结果，2026年。网址：https://aardappel.github.io/goose/bench/results.html。在线访问；访问日期：2026年11月2日。

[13]van Oortmerssen W.：《Goose 编程语言》。https://github.com/aardappel/goose，2026年。[14]Stroustrup B.：《*C++* 程序设计语言——特别版（第 *3* 版）。Addison-Wesley，2000。ISBN 978-0-201-70073-2。[15]Tofte M.、Talpin J.：基于区域的内存管理。载于：《信息与计算》，第 132 卷（第 2 期），第 109–176 页，1997 年。网址：http://dx.doi.org/10.1006/INCO.1996.2613。[16]Zhou Z.、Gogte V.、Vaish N.、Kennelly C.、Xia P.、Kanev S.、Moseley T.、De- limitrou C.、Ranganathan P.：仓库规模下内存分配器的特征分析。载于：R. Gupta、N.B. Abu-Ghazaleh、M. Musuvathi、D. Tsafrir 编，《第 29 届 ACM 编程语言与操作系统的体系结构支持国际会议论文集，第 3 卷，ASPLOS 2024，美国加利福尼亚州拉霍亚，2024年4月27日—2024年5月1日》，第 192–206 页。ACM，2024 年。网址：http://dx.doi.org/10.1145/3620666.3651350。

A. 测量细节

外部启动器在调用 fork() 前立即调用了函数 clock_gettime(CLOCK_MONOTONIC)。它在子进程中启动基准测试，并使用 wait4() 函数检测完成情况。它从子进程的资源统计信息中读取 ru_maxrss。该值记录的是进程最大常驻集大小，而非内存分配次数。

在重复回收基准测试中，工作线程先完成构建和遍历，然后由主线程获取每个 LIVE 快照。快照期间，树仍保持分配状态。同步完成后，每个工作线程分别为自己的存储释放操作计时。只有在所有工作线程都完成释放后，才会获取下一个 ROUND 快照。工作线程计时器不包括同步等待时间，但进程总时间包括这部分时间。

共享辅助程序从 /proc/self/smaps_rollup 中读取 Rss: 字段，也从 /proc/self/statm 中读取虚拟内存大小。这两项测量都描述整个进程。逻辑释放后，驻留内存可能仍分配给该进程，并可供重复使用。我们没有执行显式的内存修剪。我们将 wait4() 报告的最大常驻集大小与采样的常驻集大小测量值分开记录。第五个 ROUND 样本未纳入释放后比较，因为 Goose 工作线程可能会在此检查点附近终止。

本文已被《计算机科学》期刊接收发表：*http:// journals. agh. edu. pl/ csci*

所属机构

Rafał Frączek
克拉科夫 AGH 大学，Mickiewicza 大道 30 号，30-059 克拉科夫，rafalfr@agh.edu.pl

收稿：???**修订**：???
录用：???