软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(4):2076−2098 [doi: 10.13328/j.cnki.jos.006830]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

榫卯: 一种可组合的定制化内存分配框架*

欧阳湘臻, 朱怡安, 史先琛

(西北工业大学 计算机学院, 陕西 西安 710072)

通信作者: 欧阳湘臻, E-mail: matrixsukhoi@mail.nwpu.edu.cn

摘　要: 动态内存分配器是现代应用程序重要组成部分, 它负责管理空闲内存并处理用户内存请求. 现代通用动态内存分配器能够提供较为平衡的性能与内存利用率, 但考虑到不同应用场景的内存使用情况和优化目标不同, 使用通用内存分配器并非最优解. 针对应用场景定制的专用内存分配器通常能够更好地满足系统需要, 然而编写专用内存分配器较为费时, 也容易出错. 开发者通常使用内存分配框架搭建专用动态内存分配器. 然而, 现有的内存分配框架存在抽象能力较差, 组合性与定制性不足的问题. 为此, 从函数式编程视角审视动态内存分配过程, 基于函数可组合性提出了一种可组合的定制化动态内存分配器框架榫卯. 榫卯框架将系统内存分配抽象为多个互不耦合的内存分配层级函数的组合, 这些层级函数能够扩展出策略槽, 以提供更高的定制性和组合性. 榫卯框架基于标准 C 实现, 依赖 C 预处理器的元编程特性实现层级函数组合的零性能开销. 开发者能够通过组合与定制分配器的层级函数, 快速构建出适合应用场景的内存分配器. 为了证明榫卯框架的有效性, 使用榫卯框架构建了 3 种不同的内存分配器实例: tlsfcc, hslab 与 wfslab, 其中 tlsfcc 针对多核嵌入式应用场景, 通过替换同步策略优化并发吞吐率; hslab 是核心感知的 slab 式分配器, 通过定制线程缓存优化在异构硬件的性能; wfslab 是低延迟的无等待/无锁分配器. 为了评估这 3 种内存分配器实例, 通过运行基准测试对比现有内存分配器. 实验分别在 8 核 x86/64 平台和 8 核异构 aarch64 嵌入式平台进行. 实验表明 tlsfcc 与原始 tlsf 分配器相比, 在上述两个平台上分别取得了平均 1.76 和 1.59 的加速比; 对比 hslab 与类似架构的 tcmalloc, 它在两个平台的平均执行时间仅为 tcmalloc 的 69.6% 和 85.0%; wfslab 则取得了参与实验对比的内存分配器中最小的最差情况内存请求延迟, 其中包括目前最先进的无锁内存分配器 mimalloc 和 snmalloc.

关键词: 内存分配; 阻塞式同步; 异构系统; 操作系统; 函数式编程

中图法分类号: TP316

中文引用格式: 欧阳湘臻, 朱怡安, 史先琛. 榫卯: 一种可组合的定制化内存分配框架. 软件学报, 2024, 35(4): 2076–2098. http://www.jos.org.cn/1000-9825/6830.htm

英文引用格式: Ouyang XZ, Zhu YA, Shi XC. Mortise: Composable and Customizable Memory Allocator Framework. Ruan Jian Xue Bao/Journal of Software, 2024, 35(4): 2076–2098 (in Chinese). http://www.jos.org.cn/1000-9825/6830.htm

Mortise: Composable and Customizable Memory Allocator Framework

OUYANG Xiang-Zhen, ZHU Yi-An, SHI Xian-Chen

(School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China)

Abstract: Dynamic memory allocators are fundamental components of modern applications. They manage free memory and handle user memory requests. Modern general-purpose dynamic memory allocators ensure the balance of performance and memory footprint. However, in view of different memory footprints and optimization goals in application scenarios, a general-purpose memory allocator is not the optimal solution. Special-purpose memory allocators for specific application scenarios usually can better satisfy system requirements.

* 基金项目: 国家重点研发计划 (2021YFC2802503, 2020YFB1712201); 陕西省重点研发计划 (2021ZDLGY05-05)
收稿时间: 2022-04-17; 修改时间: 2022-08-02; 采用时间: 2022-11-01; jos 在线出版时间: 2023-06-28
CNKI 网络首发时间: 2023-06-29

However, they are time-consuming and error-prone to implement. Developers often use the memory allocation framework to build special-purpose dynamic memory allocators. However, the existing memory allocator framework has the problems of poor abstraction ability and insufficient composability and customizability. For this reason, this study proposes a composable and customizable dynamic memory allocator framework, namely mortise, based on function composability by reviewing the dynamic memory allocation process from the perspective of functional programming. The framework abstracts system memory allocation as a composition of hierarchical functions of several multiple decoupled memory allocations, and these functions can provide policies to ensure higher customizability and composability. Mortise is implemented by using standard C. To achieve zero performance overhead of hierarchical function composition, mortise uses the metaprogramming features offered by the C preprocessor. Developers can quickly build a memory allocator for targeted application scenarios by composing and customizing the hierarchical function of allocators. In order to prove the effectiveness of mortise, this study presents three different memory allocator instances, namely tlsfcc, hslab, and wfslab, by using mortise. Specifically, tlsfcc is designed for multi-core embedded application scenarios, which improves the parallel throughput by replacing the synchronization strategy; hslab is a core-aware slab-type allocator, which optimizes performance on heterogeneous hardware by customizing thread cache; wfslab is a low-latency and wait-free/lock-free allocator. This study runs benchmarks to compare these allocators with several existing memory allocators. The experiments are carried out on an 8-core x86/64 platform and an 8-core heterogeneous aarch64 embedded platform, and the experimental results show that tlsfcc achieves a mean speedup of 1.76 and 1.59 on the two platforms compared with the original tlsf allocator; hlsab achieves only 69.6% and 85.0% execution time compared with the tcmalloc with a similar architecture; the worst-case memory request latency of wfslab is the smallest among all memory allocators in the experiment, including the state-of-art lock-free memory allocators: mimalloc and snmalloc.

Key words: memory allocation; blocking synchronization; heterogeneous system; operating systems; functional programming

动态内存分配器负责管理空闲内存并在程序请求内存时返回满足需求的内存块, 它是高级编程语言, 操作系统以及现代应用程序中十分重要的组件. 例如, Java, Python, Haskell 等高级编程语言中对象的创建与释放依赖于运行时动态内存分配机制; Redis, V8 Javascript 引擎等应用程序运行过程中也伴随着大量内存对象的动态申请与释放. 一般而言, 通用的动态内存分配器, 如 GNU C 标准库 malloc (glibc malloc), tcmalloc, jemalloc, 能够提供低延时的内存分配与释放路径, 以及较低的内存占用. 然而, 不同应用场景下对内存分配器的要求并不完全相同, 采用通用内存分配器并不是最优解. 例如, 在软实时系统中使用动态内存分配器必须确保分配器的申请与释放路径的最差执行时间是确定的[1,2]. 而对于硬实时系统而言, 考虑到任务超时的风险, 应当避免内存资源的并发访问. 对于大型多核服务器, 程序访存开销与核间通信开销较为显著, 如数据仓库服务器中 CPU 等待周期主要是由 CPU data cache 与 TLB 未命中产生的[3]. 在这种应用场景下, 内存分配器应当具有良好的局部性, 以充分利用 TLB 和 cache. 在异构多核系统中, 一个 CPU 包含具有不同微体系结构的异构核心, 这些异构核心通常拥有不同的 cache 大小, 分支预测缓存, 指令发射窗口, 而通用动态分配器的设计无法感知到硬件结构, 从而不能充分利用硬件性能加速内存分配及访问过程. 此外, 对于一些可能存在内存漏洞的应用, 内存分配器还需要在其管理的内存块中增加冗余信息, 以缓解代码中存在的内存风险. 为满足不同应用场景的需求, 使用定制的专用动态内存分配器是更好的选择. 然而, 现代动态内存分配器的设计与实现较为复杂, 以 tcmalloc 为例, 该分配器包含每线程的 thread cache, 全局共享的 central cache 和 page heap 等复杂数据结构, 数据结构之间存在代码耦合. 手动实现与维护的代价较高, 且代码复用能力较差. 例如在 tcmalloc 基础上实现的巨页感知的加强组件 Temeraire[4]由于代码耦合无法简单地应用到其他内存分配器之上, 需要大量的移植和重新编码工作. 因此, 针对传统内存分配器组合性与定制性不足, 代码复用性较差的缺点, 内存分配框架被提出来.

内存分配框架一般将内存分配算法分为多个互不耦合的层级, 每一层都有各自的实现. Linux 内核内存分配框架分为页分配器与对象分配器两层. 内核通过 Kconfig 配置提供多种可选的分配器与优化选项. 其中页分配器是固定的伙伴算法, 而对象分配器有 3 种可选实现 slab, slub 和 slob, 分别针对通用系统, 大型多核系统以及嵌入式系统优化. 然而, Linux 内存分配框架的组合性不足, 不能定制更灵活的内存分配策略, 如页分配器默认为二分伙伴系统, 无法更换. 另一方面, Linux 内核分配框架与其他模块耦合程度较高, 例如对象分配器之一的 slub 分配器的实现需要修改表示物理页的数据结构, 影响代码的复用和模块的隔离. 可组合的高性能内存分配器框架 HeapLayer 解决了内存分配框架组合性不足的问题[5]. HeapLayer 基于 C++面向对象特性与元编程的模板功能, 将

内存分配器视为多个 Heap 类的组合, 并提供灵活, 解耦合的 Heap 类自定义. Heap 类主要实现 3 种接口, 分别为 malloc (sz)/free (ptr)/get_size (ptr). 在构造内存分配器时, 用户只需要通过 C++ mix-in 机制混合多个 Heap 类即可得到所需的内存分配器. 虽然 HeapLayer 能够定制出满足大部分应用场景的专用分配器, 但是它依然存在一些限制. 例如, Heap 类内存请求传递的只有内存请求的大小或者释放的内存指针. 尽管 HeapLayer 能够快速构建出 malloc(3) 定义的内存分配器实现, 却无法实现更灵活的用户接口. 例如操作系统内存分配常用的接口并非 malloc(3), 而是页对齐分配接口 page_alloc 与对象分配接口 kmalloc; 而在一些内存分区隔离的分配器中, 分配接口还需要传入额外的核心 ID 作为可选参数以区分特定的 DRAM bank 和 CPU cache 组[6−8]. 同时, HeapLayer 也没有提供同步机制的抽象, 这意味着 HeapLayer 无法依据应用场景的需求与并发竞争程度定制最佳的线程同步方式. 例如, 在存在大量竞争分散的锁时, CAS 自旋锁具有性能优势; 而在要求确定的最差情况执行时间的实时系统领域, 同步过程需要使用公平锁, 它能保证等待锁的线程不会陷入饥饿; 在存在高度并发的 NUMA 系统中, 基于委任的分层锁通常更具性能优势[9−15]. HeapLayer 依赖 C++标准库以及 C++模板提供的 mix-in[16], 因而难以在无操作系统的裸机环境或嵌入式实时操作系统环境下定制内存分配器. 此外, HeapLayer 在某些存在约束的编码场景下也无法应用, 如只有 C 编译器支持的嵌入式硬件平台以及禁止 C++编码的 Linux 内核编程.

针对当前内存分配框架存在的限制, 本文提出了一种新型可组合的定制化内存分配框架榫卯, 如图 1 所示. 榫卯框架基于可组合的函数式编程思想, 将内存分配器抽象为多个互不耦合的层级函数. 层级函数分为系统接口层, 基础层与用户接口层 3 大类. 开发人员能够通过组合已定义好的层级函数, 快速构建出所需要的专用动态内存分配器, 从而避免手动编写内存分配器时可能出现的错误. 此外, 开发人员可以使用该框架内提供的库函数, 策略函数与数据结构等基本构件编写并定制新的层级函数. 在编写过程中开发人员只需关注特定层函数的实现.

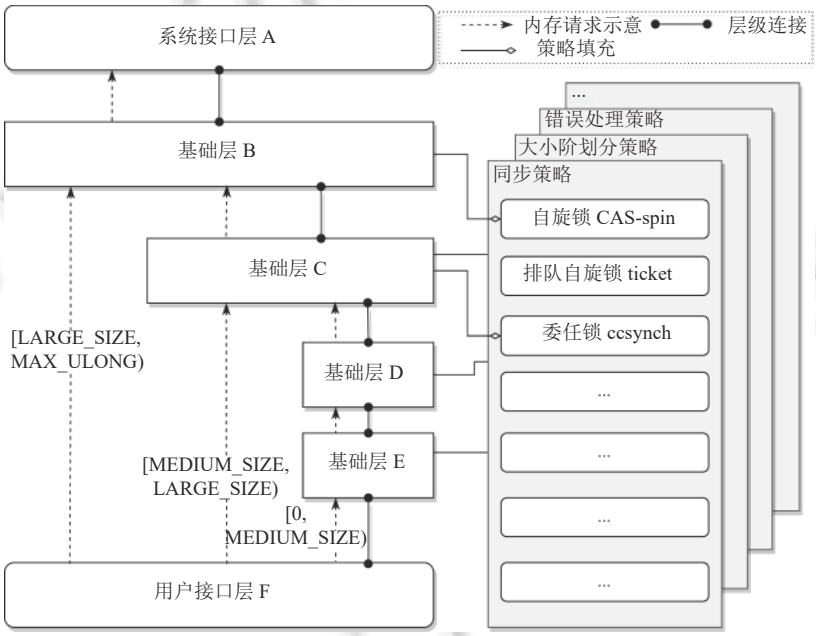

图 1 榫卯内存框架示意图

与现有的内存分配框架相比, 榫卯框架的特点在于:

(1) 组合性更好. 榫卯框架以函数式编程思想审视内存分配流程, 通过构造可扩展的内存请求上下文来在层级函数之间传递内存请求, 同时封装函数产生的副作用. 这使得榫卯框架能够自定义用户请求接口.

(2) 定制性更好. 榫卯框架为每一个层级函数扩展出策略槽, 用于插入一些可复用的策略函数, 例如大小阶 (size class) 的映射函数与保证线程安全的同步算法函数. 这些策略函数为榫卯框架提供了更高的自由度, 可以产生更多的定制组合.

(3) 可移植性更好. 榫卯框架基于标准 C 实现, 在一些只允许 C 编程的场合也能使用, 同时框架本身无外部库依赖, 所有代码都对开发者都是可见的.

(4) 用户代码简洁性更好. 榫卯框架将实现的复杂性尽可能转移到框架本身的实现当中, 而呈现给用户的接口是相当精简和直观的. 用户只需要数十行代码就能组合出一个复杂结构的内存分配器. 同时, 榫卯框架允许使用一种抽象的模型语言描述分配器的架构.

为了说明榫卯框架的组合性和定制性, 本文给出了使用榫卯框架快速定制出的 3 种针对不同应用场景的动态内存分配器的实例, 分别为针对多核嵌入式实时系统优化的 tlsfcc; 针对异构多核系统优化的 hslab; 针对高并发和延迟敏感场景优化的 wfslab. 本文通过运行内存分配器基准测试, 收集相关性能统计信息, 从平均执行时间, 内存占用, 最差情况内存请求延迟 3 个角度评估这 3 种内存分配器. 为了证明榫卯框架的可移植性, 实验分别在 8 核 16 线程的 x86/64 AMD Ryzen 3700x 平台和 8 核异构 aarch64 Qualcomm Snapdragon 888+嵌入式平台进行, 参与实验对比的通用或专用内存分配器包括:

glibcmalloc (https://www.gnu.org/software/libc/sources.html), tlsf[17], tcmalloc (https://github.com/gperftools/gperftools), jemalloc (https://github.com/jemalloc/jemalloc), Hoard[18], tbbmalloc[19], mimalloc/smimalloc[20], snmalloc[21], Scudo (https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/scudo), mng (https://github.com/richfelker/mallocng-draft), Diehard[22].

实验结果显示使用榫卯框架实现的 3 种内存分配器实例满足其应用场景要求的性能优化目标, 从而证明了榫卯框架的有效性.

本文第 1 节介绍内存分配器与内存分配框架的相关研究. 第 2 节介绍榫卯内存分配框架的设计与实现细节. 第 3 节介绍使用榫卯框架构建专用内存分配器的 3 个实例. 第 4 节介绍实验设置并给出实验结果和对比分析. 第 5 节总结全文.

1 相关工作

内存分配器作为系统的关键组件, 其相关算法与高性能实现已有大量的研究, 这些研究主要集中在优化内存分配器执行时间, 内存利用率以及容错性. 本节主要介绍通用内存分配器, 专用内存分配器以及内存分配框架的相关工作.

1.1 通用内存分配器

随着多核处理器得到普及, 为了解决线程安全和多核可扩展性问题, 分配器必须使用合适的同步方式处理共享数据的并发访问. 本文按照通用内存分配器使用的同步方式, 将这些内存分配器分为基于锁的内存分配器与无锁内存分配器两大类.

基于锁的通用内存分配器包括 ptmalloc, jemalloc, tcmalloc, Hoard[18]. ptmalloc 是 GNU C 库默认的分配器, 下文中称为 glibc malloc. 它是一种分割-合并式分配器, 使用带标记的 chunk 作为分割-合并的基本单位. glibc malloc 使用分级管理策略维护 chunk, 依据 chunk 大小实行不同的分配策略. 同时, glibc 分配器通过将锁的竞争分散到不同的共享数据结构 arena 中以提供多核可扩展性. jemalloc 是 FreeBSD 使用的内存分配器, 它是一种 slab 式的分配器. 连续的内存被划分为 run, 每个 run 会被划分成特定大小的小内存块, 小内存块之间不会分割或合并, 每个 run 通过 bitmap 标记内存的使用情况. jemalloc 的多核同步策略与 glibc 相似, 它使用 4 倍于最大核心数的 arena. 线程被指派到一组 arena 中, 以降低对同一 arena 的并发竞争. tcmalloc 是 Google Chrome 浏览器中使用的内存分配器, 它通过线程本地分配缓存 thread cache 与中心式的 central cache 来管理空闲内存. 同时, 为了防止单一线程持有大量空闲内存, tcmalloc 实现了 thread cache 窃取机制. Hoard 为了缓解并发瓶颈, 将用户堆分为全局堆和每 CPU 核心堆两种, 以分担并发压力. 另外, Hoard 通过确保相同 CPU 核心尽可能使用相同 cache 行来降低跨 CPU 核心的 cache 行并发访问对性能的影响, 以一定内存消耗为代价提升 CPU 访存性能. 在使用的同步技术方面, glibc malloc, jemalloc, tcmalloc 和 Hoard 都使用锁来保证线程安全性, 其中 glibc malloc, jemalloc 使用 POSIX 互斥

锁, tcmalloc 与 Hoard 使用用户态 CAS 自旋锁.

基于无锁数据结构的通用内存分配器包括 lfmalloc[23,24], scalloc[25], supermalloc[26], mimalloc[20]与 snmalloc[21]. 在现代处理器平台上, 诸如 LL/SC, CAS, SWAP, FAA 等硬件原子原语的引入使得无锁编程成为可能[27−34]. 无锁编程消除了死锁与优先级反转问题, 在提供高性能的同时保证系统的高健壮性. lfmalloc[23]基于 MP-RCS (multi-processor restartable critical section, 多处理器可重新开始的临界区) 实现了无锁, 而另一个 lfmalloc[24]是在 Hoard 分配器架构引入无锁算法实现的无锁分配器. scalloc 同时使用锁和无锁数据结构的混合架构, 并利用操作系统透明巨页 (transparent huge page) 来降低系统延迟. supermalloc 使用 x86 HTM (hardware transaction memory, 硬件事务性内存) 机制代替了锁, 在实现无锁的同时相比其他分配器大大简化了代码实现. snmalloc 和 mimalloc 是目前最先进的无锁分配器. snmalloc 主要针对存在大量非本线程释放的多生产者/多消费者应用场景优化. snmalloc 维护着每线程私有的堆和分离适配的全局堆. 它使用基于 SWAP 原语的多生产者-单消费者队列处理消费者线程的远程内存释放操作. 在全局堆的内存分配中, snmalloc 使用基于 CAS 原语的无锁栈. mimalloc 在每个连续内存页 span 中维护了不同用途的自由链表, 其中本地释放使用本地链表, 远程释放使用远程链表. 该远程链表是基于 SWAP 原语的多生产者-单消费者栈. 然而, 无锁数据的结构的引入带来了内存回收与 ABA 问题[35−37]. 为了避免并发编程中的 ABA 问题, mimalloc 与 snmalloc 使用带版本戳的 CAS2 原语或 LL/SC 原语. 同时, 该类分配器不允许向操作系统释放存放元数据 (metadata) 的 span 头部页, 从而避免出现较慢的并发线程访问已释放的页的情况. 这里的元数据指描述内存块属性 (如大小) 的数据.

1.2 专用内存分配器

与通用内存分配器不同, 专用内存分配器则是针对特定硬件或目标应用场景优化的内存分配器, 需要结合系统硬件及软件协同设计, 以满足系统整体需求.

高性能计算系统中的内存分配算法设计通常需要考虑整个系统的内存层次结构和 CPU 核心间通信开销. tbbmalloc 是 Intel Thread Building Block 库中默认的内存分配器[19], 它是针对多 socket 的高性能服务器设计的 slab 式内存分配器. 为了尽可能减少系统调用开销与并发竞争导致的核间通信开销, tbbmalloc 使用线程私有堆. tbbmalloc 将私有的连续内存页 block 划分成许多小内存对象, 并在 block 中维护私有自由链表处理持有线程的内存申请与释放. 非本线程的申请释放则由公开的自由链表处理. 这种设计能够防止多个跨 socket 的 CPU 核心并发访问同一个 cache 行而导致的延迟.

TLSF 是针对实时应用设计的 O(1) 时间复杂度的动态内存分配算法[17]. 它采用二级大小阶划分, 利用硬件 CLZ (count leading zero, 计算前导零) 指令查找二级位图实现 O(1) 的分配与释放流程. TLSF 的内存块采用分割-合并方式管理, 通过边界标记 (boundary-tag) 维护内存块的大小与上一个内存块指针等元数据. 内存块在分配时分割出合适的大小, 在释放时尝试与邻接内存块合并. TLSF 的问题在于它在多核并行系统中必须使用一个全局锁来保证线程安全, 因此缺乏多核可扩展性.

内存隔离的内存分配器: Palloc[6]与 CAMA[7]. 对于共享内存的多核系统, cache 组相连 (set-associative) 和 DRAM bank 可能会导致 CPU 访存时间的较大变化[8]. 在时间敏感的硬实时系统中, 访存时间的变化可能会导致系统任务超时, 因此更高程度的内存隔离对硬实时系统尤为重要. cache 组与 DRAM bank 通常由内存地址的特定比特位划分, 内存隔离的分配算法依据这些内存地址比特位以分区管理策略维护空闲内存, 令每个 CPU 核心分配得到的内存所在的内存区域处于不同的 cache 组或 DRAM bank, 由此降低多核 CPU 共享物理内存对访存时间的影响. Palloc 是 DRAM bank 感知的页式分配器, 其中空闲物理页依据所在的 DRAM bank 分区管理, 避免同一 DRAM bank 在不同的 CPU 核心间共享. 对于现代 DDR 内存而言, DRAM bank 是接收内存读写命令的基本单位, 不同 DRAM bank 的读写命令是可并行的, 因此受其他 CPU 核心访存影响较小. CAMA 是 cache 感知的内存分配器, 它按照 CPU cache 组管理内存, 避免同一 cache 组在多核间的共享.

优化内存访问开销的内存分配器 Temeraire. 数据仓库服务器的性能统计信息显示 50%–60% 的 CPU 等待周期是 data cache 和 TLB 未命中导致的, 为此, Hunter 等人在 tcmalloc 基础上实现了一种巨页感知的加强组件 Temeraire[4]. 它在 tcmalloc 之上增加了 HugeAllocator 层, 该层使用启发式算法尽可能填充巨页中的碎片, 并使用

巨页 cache 机制减少向操作系统归还巨页的开销. Temeraire 以一定的执行时间开销为代价来换取 CPU 访存等待周期的降低.

Mesh[38]是针对 C/C++程序的运行时动态内存碎片整理机制, 它旨在通过替换 malloc 实现, 尽可能消除 C/C++程序中的内存碎片. Mesh 结合了一种随机算法和共享内存机制减少内存碎片, 它在保证性能的情况下减少内存碎片, 降低应用内存占用.

内存安全防护的分配器: Scudo, smimalloc, Diehard[22]与 mallocng. Scudo 分配器是 Android 11 的原生代码 (native code) 内存分配器, 它针对常见的内存漏洞 (如内存溢出, 释放后使用和重复释放) 提供额外的检查和缓解措施, 同时确保良好的性能. Scudo 在用户内存块的头部插入额外的 CRC32 校验字用于检测重复释放和缓冲区溢出, 当头部校验字出现错误意味着数据被改写. 同时, Scudo 采用延迟释放策略, 能够允许一定程度的释放后使用行为. 延迟释放策略提供固定大小的释放隔离区. 用户释放的内存首先会被移入该隔离区, 在被换出隔离区之后才会真正释放. Scudo 在检测出内存异常使用之后, 会报告错误并立即中止用户进程. smimalloc 是 mimalloc 的安全版本, 它使用异或校验的方式防止页头部元数据中的自由内存块链表被缓冲区溢出所改写. Diehard 将内存块元数据存放在用户堆外避免元数据被改写, 同时应用随机方法检测未初始化的内存读取问题. 考虑到元数据与内存块分离不利于访问元数据时的内存局部性, mallocng 采用了折中的方式, 一部分内存元数据存放在内存块头部以提高局部性, 另一部分则存放在用户堆外防止被改写.

1.3 内存分配框架

现代内存分配器通常是多种内存分配算法的结合, 而内存分配框架可以帮助开发者根据应用场景, 利用已有的内存分配算法搭建出满足需求的内存分配器, 并进行一定程度的定制与性能优化.

Linux 内核内存分配框架由页分配器和对象分配器构成, 其中页分配器主要使用分割-合并式的二分伙伴系统算法, 对象分配器是可配置的, 可选择 slab/slob/slub 其中一种实现. slab 将物理页预先分割成不可合并的小内存对象, 结合每 CPU 核心的 cache 机制以提供快速的对象分配. slob 为了降低对象分配器本身的内存开销, 使用简单的自由链表维护空闲内存, 以首次适配策略分配内存. slub 针对大型系统优化, 它将 slab 的页空闲对象数组和每核心 cache 数组改为链表以降低额外的储存开销.

针对通用内存分配器可定制性差的缺点以及现有内存分配器框架灵活性有限的问题, Berger 等人提出了一种可组合的高性能内存分配器框架 HeapLayer[5]. HeapLayer 借助面向对象编程中的继承, 封装的特性, 将内存分配框架视为 Heap 类的复合, 而通过 Heap 类的继承可以实现层级的定制. HeapLayer 内存分配器框架基于 C++模板元编程特性实现, 它利用 mix-in 组合层级. mix-in[16]是一种 C++的抽象类, 它将一个类中使用的其他类的类型抽象, 作为该类模板的一个输入参数. 前文提到的 Hoard, Mesh 以及 Diehard 内存分配器均使用 HeapLayer 框架搭建.

2 榫卯内存分配框架

内存分配框架 HeapLayer 从面向对象的编程角度考虑内存分配框架的思路启发了本文研究. 本文将从函数式编程视角考虑内存分配流程, 提出一种更直观, 同时更具组合性和定制性的分配器框架.

2.1 榫卯框架的总体设计

本文将处理内存请求的分配器层级视为多个具备可组合性的函数, 内存分配框架中的分配算法的组合即为层级函数的组合. 函数的可组合性定义源自数学函数的复合关系 $compose(f,g) := fun\ x : A \to\ f(g(x))$. 其中 $g : A \to B$, $f : B \to C$. 根据定义, 高阶函数 *compose* 接受两个函数 g 和 f 作为输入, 其中 g 的输出类型是 f 的输入类型. 它将函数 g 和函数 f 复合成一个新函数 $compose(f,g)$, 这个新函数接受一个参数 $x : A$, 返回结果的类型为 C. 通常使用算子 $f \circ g$ 表示 $compose(f,g)$. 该高阶函数满足结合律, 即 $(f \circ g) \circ k = f \circ (g \circ k)$. 本文针对内存分配流程给出抽象的函数组合模型. 模型相关的函数定义如表 1 所示.

首先, 本文定义一个内存请求上下文 mctx, 用于传递用户请求与分配器的状态. 内存请求上下文能够记录各级函数产生的副作用 (side effect), 从而保证各层函数的无状态 (stateless) 属性. 可组合的层级函数为 L, 它会修改

mctx 中的状态存储运行结果. 层级函数 L 的组合依靠高阶函数 *LCompose*, 它接受两个层级函数 L_1 和 L_2, 得到一个新的复合层级函数. 每个层级函数可以扩展出多个策略抽象, 这些策略函数 p 可以进一步提升层级的可定制性. 例如为了得到一个可组合的层级函数 L_k, 必须通过柯里化 (Currying) 的方式不断为未完成的层级函数 l_k 填充策略函数, 直到所有的策略输入被定义, 该层级函数的类型才允许进行函数的组合. 即完整的层级函数. 本文使用 $L_1::L_2$ 表示 $LCompose(L_1, L_2)$.

表 1 椁卯内存分配框架函数组合模型的定义

定义	说明
$mctx$	内存请求上下文
$L: mctx \to mctx$	可组合的层级函数定义
$LCompose(L_1, L_2) := fun\ x: mctx \to L_1(L_2(x))$	用于组合层级函数L的高阶函数定义, 其中 $L_1, L_2: L$
$p := \{p_0, p_1, \ldots, p_n\}$	策略函数定义
$l: p_i \to p_k \to \ldots \to p_m \to L$	未填入策略函数的层级函数定义
$L_k := fun\ x: mctx \to l_k(p_i, p_j, \ldots, p_m)(x)$	填满策略函数的层级函数定义
$C_m: A \to mctx$	用户内存分配封装函数
$C_f: mobj \to mctx$	用户内存释放封装函数
$D: mctx \to mobj$	内存上下文解析函数
$S: B \to mobj$	产生*mobj*的函数
$V: mobj \to mctx \to mctx$	消除*mobj*的函数

为了能够通过用户内存请求构造内存请求上下文, 我们需要定义用户内存申请封装函数 C_m 以及它的解析函数 D, 其中 A 是抽象数据类型, 可以由用户自定义. 最简单的 A 是用户请求的内存大小. *mobj* 是内存块指针及其大小. 相似地, 对于内存释放请求同样需要定义 C_f. 通常各层函数并不能凭空产生内存块. *mobj* 必须由已存在的内存块通过分割函数产生. 不过, 最顶层至少有一个能产生 *mobj* 类型的函数 S 和消除 *mobj* 的函数 V. 其中, B 为抽象数据类型, 描述可用于产生 *mobj* 的对象. 这里的 S 与 V 函数可以视为系统接口函数的抽象, 它可以通过向操作系统请求或归还内存来产生或消除 *mobj*. 由此本文得到分配器的整体 3 层函数架构: (1) 用户接口层函数实现 C_m, C_f 和 D, 该层定义用户调用的内存请求接口, 使用抽象数据类型帮助用户根据应用需求定制更为灵活的内存请求接口, 例如可以将抽象数据类型 A 定义为 size×cpu_id , 要求用户同时输入请求的大小以及 CPU 核心 ID, 以此可构建根据 DRAM bank 或者 cache 分区策略的内存分配器. (2) 基础层函数实现未填入策略的层级函数 l 与策略函数 p. 该层定义具体的内存分配机制, 同时可扩展出多种可替换的策略, 用户可将 p 填入 l 中得到可组合的层级函数 L. 对于一个可组合的层级函数, 若本层无法处理用户内存请求则将请求转至上一层处理. (3) 系统接口层函数实现 S 和 V, 该层定义最终的内存请求处理. 在有操作系统环境下, 该层通常定义如何与操作系统交互, 如处理新的内存页的映射以及归还; 若在无操作系统的裸机环境下, 该层定义如何管理可用物理页.

至此, 本文提出了一种可组合的内存分配框架的设计. 该内存分配框架基于函数式编程思想将内存分配抽象为互不耦合的层级函数, 通过层级函数的组合与定制实现分配器的搭建. 如图 2 所示, 在椁卯框架中, 每一个已定义的层级函数都是一个可组合的构件, 而一个层级函数可以定义策略槽用于填充额外的策略函数以提供额外的扩展性和定制性. 若策略槽已被填满, 层级函数完成定义, 能够连接到主构件中. 而内存分配从用户接口层对用户暴露的接口函数 C_m 调用开始, 接口会构造一个内存请求上下文 *mctx*, 用于封装内存请求信息, 各层的状态信息以及分配的结果. 内存请求上下文会从底层函数向上逐级传递, 直到最顶层为止. 一旦满足要求的内存被分配, 内存块 *mobj* 会被记录到内存请求上下文中并逐级向下返回, 最后用户接口的解析函数 D 将 *mctx* 中记录的 *mobj* 提取出来. 对于内存释放请求, 则通过 C_f 函数包装为内存请求上下文, 与状态信息一并向上传递直到封装的 *mobj* 被某一层级函数回收.

同时, 本文能够用一种简单的抽象模型语言来描述复杂的内存分配器架构, 如图 1 所示的分配器, 可描述为 A::B(sync = spinlock)::C(sync = ccsynch)::D(…)::E(…)::F. 其中每个层级函数括号中的为策略槽位, 即可自定义的策略函数.

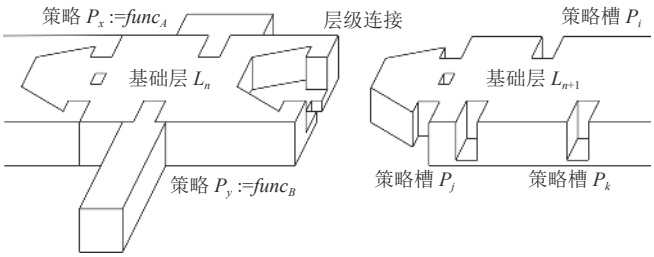

图 2 榫卯内存框架层级组合示意图

综上所述, 本文从函数式编程视角出发定义内存分配器框架主要出于以下考量. 可组合性: 函数式编程允许使用高阶复合函数组合现有的函数来产生新函数, 与面向对象编程中对类进行组合相比, 其组合粒度更低. 简洁性: 函数式编程允许使用更简洁, 抽象的函数模型语言描述或构造复杂的内存分配器. 验证友好: 层级函数的输出只与其输入参数有关, 状态和副作用均被封装到内存请求上下文中, 因而更易于测试与验证.

2.2 榫卯框架的实现细节

本文基于标准 C 实现了榫卯内存分配框架. 榫卯框架尽可能将实现的复杂性转移到框架本身, 对于开发者而言只需先引入已有层级, 使用策略填充该层, 并最终将各层连接起来即可, 如图 3 所示的内存分配器构建示例. 在榫卯内存分配框架下, 开发者仅需要数十行代码便可组合出一个复杂的, 带有尾部校验和检测及错误处理功能的 3 层架构的 malloc(3) 高性能内存分配器. 图 3 中构建的内存分配器使用抽象模型语言可以表示为: mmap_heap(faa_heap_growth=1, threadlocal_heap_size=1GB)::buddy_glk(sync=spinlock, size_handle=[65536, SIZE_MAX))::slab_wf(sizeclass=prime, size_handle=[0, 65536))::tail_chksum(chksum=xor)::malloc_socket.

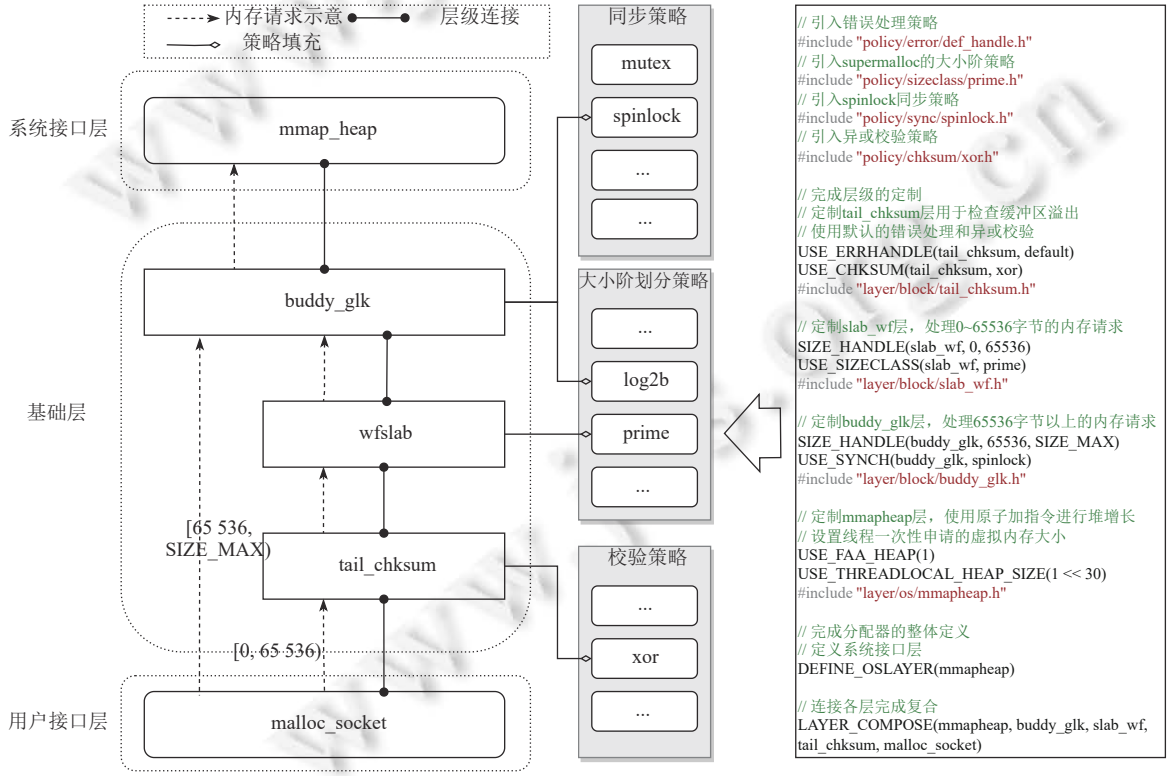

图 3 使用榫卯框架构建动态内存分配器

榫卯框架中使用高阶函数复合各层级函数, 其 C 宏实现如图 4 所示. 对于一个层级函数而言, 它至少需要实现 4 个层级函数接口 create_obj, alloc_obj, free_obj 和 getmeta_obj, 分别用于线程创建时初始化内存请求上下文中的层级对象, 处理分配和释放内存请求, 以及获取内存对象的元数据. 层级函数接口的输入参数包含内存请求上下文 mctx_t 和已完成连接的层级函数 lfunc_t 的接口. 图 4 给出了一个简单层级函数的实现示例, 该层级函数用于统计指定大小内存申请和释放次数. 对开发者而言, 编写一个层级函数只需要关注本层提供的功能以及相关数据结构, 而无需考虑其上层或下层代码. 一个完整的内存分配器构建过程首先需要使用 DEFINE_OSLAYER 完成系统接口层的定义, 然后通过 LAYER_COMPOSE 高阶函数将多个层级函数连接.

```
// 可连接的层级函数定义
typedef mctx_t * lfunc_t (mctx_t  *ctx);

#define __DECLARE_OBJ_FUNC  (func_name) \
static inline mctx_t * func_name##_obj (mctx_t  *ctx, lfunc_t  *ufunc);

// 每一层至少需要实现的层级函数
#define DECLARE_LA YER (layer_name) \
MAP (__DECLARE_OBJ_FUNC  , layer_name##_create,
layer_name##_alloc , layer_name##_free , layer_name##_getmeta )

#define __LFUNC_COMPOSABLE  (unlinkedf, linkedf) \
static inline mctx_t *__linked_##unlinkedf (mctx_t  *ctx)
{ return unlinkedf (ctx, linkedf);}

// 定义系统接口使用的辅助函数
#define __OS_LA YER (layer_type) \
__LFUNC_COMPOSABLE  (layer_type##_obj , NULL )

// 用于复合层级函数的辅助函数
#define __LCOMPOSE  (linkedl1_type , l2_type) \
__LFUNC_COMPOSABLE  (l2_type##_obj , __linked_##linkedl1_type##_obj )

#define __LCOMPOSE_TYPE  (l1, l2, type) \
__LCOMPOSE  (l1##_##type , l2##_##type )

// 定义系统接口层函数
#define DEFINE_OSLA YER (alayer) \
MAP (__OS_LA YER , alayer##_create,
alayer##_alloc , alayer##_free , alayer##_getmeta )

// 层级连接,进行三种函数的复合
#define __LAYER_CONNECT  (linkedl1 , l2) \
__LCOMPOSE_TYPE  (linkedl1 , l2, create) \
__LCOMPOSE_TYPE  (linkedl1 , l2, alloc) \
__LCOMPOSE_TYPE  (linkedl1 , l2, free) \
__LCOMPOSE_TYPE  (linkedl1 , l2, getmeta)

// 使用FOLD LEFT连接整个层级函数链表
#define LAYER_COMPOSE  (...) \
FOLDL (__LAYER_CONNECT  , __VA_ARGS__)
```

一个简单基础层级函数实现示例
该层级函数用于统计指定大小内存申请和释放的次数

```
// 声明层级
DECLARE_LA YER (count)
// 层级对象结构体定义
typedef struct {
   unsigned long alloc_count;
   unsigned long free_count;
} count_obj_t ;

count_obj_t  count_obj_global ;
// 将储存层级数据的对象指针记录到线程的内存上下文中
static inline mctx_t * count_cr eate_obj (mctx_t  *ctx, lfunc_t  *ufunc){
   // 此处所有线程共享一个全局对象指针
   mctx_setlayer_obj  (ctx, &count_obj_global );
   // 若线程私有对象指针则使用 os_alloc接口分配线程私有的空间
   return ufunc (ctx);
}

static inline mctx_t * count_alloc_obj  (mctx_t  *ctx, lfunc_t  *ufunc){
   // 使用分级管理策略判断是否直接跳过本级
   MATCH_HANDLE_SIZE  (count, ctx->rsize, ctx, ufunc);
   // 获取当前层级对象
   count_obj_t  *l = (count_obj_t  *)mctx_getlayer_obj  (ctx);
   debug_printf  ("count alloc size %llu\n", ctx->rsize);
   // 增加统计计数
   FAA(&l->alloc_count, 1);
   // 内存请求转交给上级处理
   return ufunc (ctx);
}

static inline mctx_t * count_fr ee_obj (mctx_t  *ctx, lfunc_t  *ufunc){
   // ...
}

static inline mctx_t * count_getmeta_obj  (mctx_t  *ctx, lfunc_t  *ufunc){
   // 该层不负责管理内存对象的元数据 ,直接交给上层处理
   return ufunc (ctx);
}
```

图 4　榫卯框架的实现细节与一个简单的层级函数示例

由于 C 语言并不支持函数式编程, 因此实现函数的复合还需要用到 C 预处理的元编程特性. 层级函数的复合主要依靠高阶函数 LAYER_CONNECT 实现. 该高阶函数要求输入一个上级已完成连接的层级函数和未完成连接的层级函数, 输出一个完成连接的层级函数__linked_layer_type_obj. 在编译预处理阶段, C 预处理器会展开相关宏, 替换宏参数从而实现中间函数的生成与函数的复合. 此外, 图 4 中所使用的函数式编程中常用的高阶函数 MAP 和 FOLDL 依赖 C99 标准可变参数宏机制以及 C++20 标准可递归宏机制. 其中 MAP 接受一个函数 f 和一串输入, 将每个输入传入 f 执行; FOLDL 则将前一次输出与下一次输入一同传入函数 f, 递归产生输出. 由于这两个高阶宏函数的 C 实现较为复杂且有相应开源代码库 macrofun 可供参考, 本文在此不再赘述其具体实现.

在榫卯框架中, 可复合的层级函数分为系统接口层函数, 基础层函数和用户接口层函数 3 种, 其中基础层函数允许存在多个, 而系统接口层函数和用户接口层函数只允许最多 1 个, 分别位于层级顺序的第 1 个和最后 1 个. 榫

卯内存框架当前提供的可用层级函数如表 2 所示. 其中系统接口层抽象了管理用户堆的方式, 它能将 mmap, brk, virtualAlloc 等操作系统相关的接口抽象, 以提供更好的跨平台移植性. 同时, 系统接口层也允许在无操作系统的嵌入式平台上使用预定义的静态内存池来管理现有内存. 基础层分为主干和附加两类, 其中主干部分是构成内存分配器的主体, 它抽象了内存分配所使用的算法, 如 slab(slab_lk), 伙伴系统 (buddy_glk) 和 TLSF(tlsf_glk). 基础层还包括一些实现较为简单的附加层, 其中有提供额外分配约束的层级, 如内存对齐分配的 aligned 层, 负责将用户传入的内存请求上下文中的大小对齐到指定大小; 也有改善性能的附加层, 如改善 CPU cache 局部性的线程缓存 thread_cache 与核心感知的缓存的 carrcache_lk, 以及改善 TLB 局部性的 slab_arrcache. 附加层也可以在非内存安全的应用场景下提升系统健壮性, 例如检测缓冲区溢出漏洞的 tail_chksum, 以及检测释放后使用行为的 delayed_free; 还有额外的附加层提供条件绕过机制, 允许线程在满足条件的情况下绕过上一层进行内存分配, 以提供更灵活的组合.

表 2　榫卯内存分配框架的层级函数

分类	层级函数	说明
系统接口层	mmap_heap	使用UNIX mmap接口
	brk_heap	使用C库brk接口
	virtalloc_heap	使用virtualAlloc接口
	sheap_glk	可配置的静态内存堆
	malloc_heap	使用C库malloc接口
基础层 (主干)	slab_lk	slab式, 每大小阶锁
	buddy_glk	伙伴系统, 全局锁
	tlsf_glk	TLSF, 全局锁
	segfit_lk	分离适配, 每大小阶锁
	segfit_lf	分离适配, 无锁同步
	slab_wf	slab式, 无等待同步
基础层 (附加)	aligned	分配大小对齐
	batched_free	批量释放缓存层
	thread_cache	每线程缓存, 使用LIFO链表管理
	carrcache_lk	每CPU核心缓存, 使用缓存数组管理
	tarrcache	每线程缓存, 使用缓存数组管理
	szcls_stat	统计该层以上内存请求大小阶分布
	wcet_stat	统计该层以上内存请求延迟分布
	perf_stat	统计该层以上内存请求平均时间
	tail_chksum	内存块末尾增加校验和
	delayed_free	内存块头部增加校验与延迟释放
	cond_bypass	可自定义判断条件, 绕过上层进行分配
用户接口层	malloc_socket	malloc.h中定义的malloc(3)接口
	kernel_socket	Linux内核内存分配接口
	cmalloc_socket	额外传入核心ID的malloc(3)接口
	custom_socket	可自定义入参的malloc(3)接口

此外, 一些附加层也可以记录额外的性能信息或者调试信息, 如统计内存请求大小阶分布的 szclas_stat 和最差情况请求时间分布的 wcet_stat, 能够在运行时统计并记录信息以便系统的后续优化. 基础层使用榫卯框架提供的数据结构实现搭建, 以提高代码的可复用性. 用户接口层定义用户如何与内存分配器交互, 并向上连接基础层. 用户接口层的意义在于实现更多样化的用户接口以适配不同的应用场景. 例如在操作系统内核中定制 page_alloc/kalloc 接口或者在用户态定制 malloc(3) 接口. 在 CPU 核心感知的应用场景下, 用户可以通过传入额外的 CPU 核

心 ID 来提示内存分配器使用不同的管理方式.

榫卯框架通过为层级函数扩展出策略槽, 以提供更好的可定制性和代码复用能力. 策略槽的实现实际上是一组遵循相同调用规则的接口, 利用 C 预处理的元编程特性可以将策略函数作为层级函数的输入参数填入. 榫卯框架目前包含以下类型的策略函数.

(1) 分级管理策略. 在现代内存分配器中通常使用分级管理, 对不同大小范围的内存请求采用不同的内存分配算法. 为了实现这种分级管理, 榫卯内存分配框架将其抽象为分级管理策略, 由 SIZE_HANDLE 宏定义其管理的内存大小范围, 若超过范围则直接转交给下一层处理.

(2) 大小阶策略. 分离适配算法中通过大小阶区分不同大小的内存块请求, 大小阶的划分粒度通常会影响到内存碎片率. 对于使用分离适配的基础层, 本框架将大小阶划分抽象为统一的接口, 然后使用 USE_SIZECLASS 宏为每层定制大小阶划分策略. 每一个大小阶划分策略必须需要实现 size_to_szcls 和 szcls_to_size 接口. 榫卯框架提供 tcmalloc, supermalloc 等分配器中的大小阶策略实现, 同时还提供配置间隔的自定义大小阶策略.

(3) 同步策略. 线程共享的层级存在临界访问的问题, 必须依赖线程间同步机制保证并发访问共享数据结构的线程安全. 本文提出的框架将同步机制抽象为同步策略, 将临界区执行封装为同步过程调用[14], 使用统一 atomic_exec 接口执行. 开发者能够使用 USE_SYNC 宏为层级函数指定同步方式. 榫卯框架提供多种类型的阻塞式同步算法实现, 如传统的竞争自旋锁 spinlock, 排队锁 mcslock, 基于合并同步的 ccsynch, 或是基于委任的 RCL, 这些同步算法适用于不同的硬件和应用场景.

(4) 校验策略. 使用非类型安全的编程语言编写的用户应用程序可能存在非法使用内存的行为, 包括缓冲区溢出, 释放后使用, 空指针, 重复释放等. 针对这些内存漏洞, 内存分配器可为用户内存块增加额外字段用于记录校验和, 检验内存块是否被非法使用. 校验策略需要定义校验字 chksum_t 以及 chksum_calculate 接口, 该接口依据内存指针以及大小生成校验字. 通常, 为了减少性能开销, 开发者可使用简单的魔数 (magic number) 校验或是异或校验, 而在要求更高检出率的场景下, 开发者可以进一步扩展校验字大小, 使用更复杂的 CRC 冗余校验.

(5) 错误处理策略. 错误处理策略定义在检测出内存非法使用时, 如何进行相应的处理. 榫卯框架将错误处理抽象为 raise_err 接口, 统一编码错误类型. 用户可以通过定制该接口实现更灵活的错误处理方式, 如进行容错处理 (如丢弃被改写的内存块) 或是执行用户注册的错误处理函数.

2.3 榫卯框架小结

榫卯内存分配框架允许用户利用现有的层级函数和策略函数快速构建和定制动态内存分配器, 同时也支持用户编写新的层级函数. 框架本身也包含大量基础数据结构与函数组件用于编写层级函数. 与现有的内存分配框架 HeapLayer 相比, 榫卯框架是以函数式编程的方式处理内存分配过程, 其定制性, 组合性和简洁性更佳. 一方面, 榫卯框架提供了策略抽象以提供更好的可定制性, 如同步策略, 错误处理策略抽象等; 另一方面, 这些遵循同一调用规则的可复合的层级函数允许更灵活的分配器组合, 例如条件绕过上层函数向指定层传递内存块, 以及对内存请求上下文 *mctx* 的扩展. 这些都是 HeapLayer 框架无法做到的. 同时, 榫卯框架将复杂性转移到框架本身的实现上, 呈现给用户的接口也更为简洁. 例如榫卯框架复合层级函数使用的 LAYER_COMPOSE 传入的参数是从高层到低层的层级列表, 避免了在 HeapLayer 中复合多个层级时容易陷入括号嵌套的情况, 因此代码更为直观, 可读性更好. 基于函数式编程思想构建内存分配器也能够简化分配器模型的表示, 例如分配器状态全部封装在内存请求上下文中, 层级函数的副作用也只会修改内存请求上下文. 这些特性都有助于内存分配器抽象模型与实现的形式化验证.

在性能开销方面, HeapLayer 框架利用 C++ mix-in 元编程机制实现零开销的层级组合, 而榫卯框架基于 C 宏元编程, 以传入上层函数回调, 生成中间函数的形式来实现层级函数的复合. 这种方式可能会造成不必要的函数调用过程. 不过, 此性能开销可由现代编译器优化消除. 主流的 C 编译器会将 inline 函数的调用过程强制展开, 从而避免额外的函数调用过程. 此外, 内存请求上下文的构造和在层级间的传递也存在一定的栈开销和重复的逻辑判断. 不过由于该过程发生在该框架内部, 且框架全部以.h 头文件实现, 因而现代 C 编译器能够对框架内所有可见代码进行自动推导, 利用常量折叠, 死代码消除等优化方式消除这部分开销.

在可移植性方面, 榫卯框架除系统接口层外, 整体框架实现不依赖于任何外部代码. 相比只支持在操作系统环境下定制内存分配器的 HeapLayer, 榫卯框架还允许在无操作系统的裸机上定制内存分配器. 不过, 榫卯框架依赖 C/C++预处理器的可变参数宏机制 (C99 标准) 以及递归宏机制 (C++20 标准). 考虑到一些嵌入式 C 编译器仅支持 C89 标准, 榫卯框架构建的内存分配器可能存在无法通过编译的问题. 然而, 由于榫卯框架依赖的是预处理器而非编译器, 该问题可以通过混合使用编译工具链来解决: 使用支持目前最新 C2x/C++20 标准的 GCC/Clang 编译器产生预处理过的.c 文件, 然后再交给嵌入式 C 编译器编译.

3 榫卯内存分配框架实例

在实际应用的系统中, 定制专用内存分配器需要根据应用场景的内存分配统计数据进行分析, 判断系统瓶颈所在, 然后修改造成瓶颈的部分. 在传统的内存分配器基础上开发定制专用内存分配器, 需要对整体代码有深入了解, 因此定制的时间成本较高且代码复用性也较差. 相比之下, 使用内存分配框架定制专用内存分配器有显著的开发效率优势. 由于榫卯内存分配框架的层级和策略都是可替换的, 定制专用内存分配器可以通过组合现有层级, 更换或编写新的层级和策略函数来实现. 而这些重新编写的函数也能够得到复用. 本节主要以实例说明如何根据应用场景和优化目标, 使用榫卯内存分配框架定制 3 种不同类型的专用内存分配器.

(1) 通过定制同步策略优化多核性能: tlsfcc 分配器. TLSF 是嵌入式实时操作系统中常用的内存分配算法, 它具有内存碎片率低, 内存请求延迟低的优点. 然而在多核环境下, TLSF 需要一个全局锁来保证线程安全, 因此该全局锁是 TLSF 分配器的并发性能瓶颈所在. 如今多核嵌入式系统已得到广泛应用, 为了优化 TLSF 在多核以及异构多核系统上的性能和内存请求延迟, 本文使用榫卯框架实现了一种改进的 tlsfcc 分配器. 该分配器主要将 TLSF 全局锁抽象为同步策略, 并将同步策略指定为公平排队的合并同步 ccsynch[11]. 在存在多个等待的竞争线程时, ccsynch 同步算法的性能更好, 同时, 其同步算法的公平性也能确保最差情况执行时间有界. 整个定制过程无需改动 TLSF 层函数的实现, 只需要通过 USE_SYNC 宏指定同步策略. 此外, 由于 TLSF 使用分割合并式的内存块管理方式, 如果地址空间不连续可能会造成较多的内存碎片, 因此本文使用 brk_heap 作为 tlsfcc 的系统接口层, 该层使用 brk 系统调用确保申请的地址空间是连续的. tlsfcc 内存分配器结构使用抽象模型可描述为:

brk_heap::tlsf_glk(sync=ccsynch, size_handle=[0, SIZE_MAX))::malloc_socket.

(2) 通过核心感知的分配缓存优化性能: hslab 分配器. 传统的 slab 式分配器通常假设所有 CPU 核心是相同的, 因此在分配缓存层的配置上采用大小相同的设计. 如今异构多核 CPU 已经在许多功耗敏感的系统上得到应用, 而这些异构处理器核心的 cache 大小并不是相同的. 例如, 8 核异构处理器 Snapdragon 888 集成的 3 种异构核心拥有不同的 L1/L2 cache 大小. 为了能够充分利用异构处理器的硬件 cache, 本文依据异构 CPU 不同核心的 cache 大小对线程分配缓存进行定制, 实现了一种异构核心感知的 hslab 内存分配器. hslab 内存分配器整体结构与 tcmalloc 相似, 它采用一个全局的伙伴系统管理大于 32 KB 的内存, 使用 slab 式分配算法管理小内存, 使用一个 thread_cache 附加层作为线程本地分配缓存. hslab 在 thread_cache 层启用了核心感知的动态缓存. 该层函数会定期查询当前线程所在的 CPU 核心 ID, 并根据预先配置的核心信息动态调整缓存大小. 此外, 为了方便对比, hslab 分配器采用与 tcmalloc 相同的大小阶划分策略. 相比于只使用自旋锁同步的 tcmalloc, hslab 还为各层精细化定制同步方式. 其中, 全局伙伴系统是集中式竞争, 采用 flat-combine[10]同步策略能够最大化该层并发吞吐率. slab 层是每大小阶的竞争, 由于 hslab 的大小阶划分较多, 并发竞争较为分散, 因此使用 POSIX 自旋锁较为合适. POSIX 自旋锁的获取开销较低, 同时可以确保长时间等不到锁的线程会主动出让 CPU 给其他线程. hslab 分配器结构使用抽象模型可表示为:

mmap_heap(faa_heap_growth=1, threadlocal_heap_size=2MB)::buddy_glk(sync=flatcombine, sizeclass=tc, size_handle=[32768, SIZE_MAX))::slab_lk(sync=posix_spin, slab_size=64KB, sizeclass=tc, size_handle=[0, 32768))::thread_cache(sync, sizeclass=tc, size_handle=[0, 32768), cache_size=2MB, coreaware_cachesize=1)::malloc_socket.

(3) 通过系统设计降低内存请求延迟: wfslab 分配器. 在时间敏感的系统中, 大量的并发内存请求需要在尽可能短的时间内完成. 为了尽可能降低内存请求延迟, 每个层级都应该有最小的内存请求延迟, 同时也应该将内存分

配器运行的系统环境考虑在内. 本文通过系统设计实现了一种低延迟的无锁/无等待内存分配器 wfslab. 该分配器使用两种方式减少内存请求延迟.

wfslab 基础层使用无锁/无等待数据结构和算法, 在处理小内存分配时使用无等待分配算法, 而在处理大于 64 KB 的大内存时使用分离适配的无锁分配算法. 在高并发系统中, 核间同步方式往往会成为系统瓶颈. 为了降低同步开销, 开发者通常在竞争激烈的数据结构中使用无锁技术. 针对高并发系统的内存分配器的设计同样可以使用无锁数据结构优化并发性能, 降低内存请求延迟. 无锁技术能够保证所有竞争线程中, 至少有一个线程能够取得进展. 然而, 使用无锁数据结构有可能会导致线程饥饿, 因此在时间敏感的应用场合通常要求能够保证无饥饿的无锁, 即无等待[39,40]. 使用无等待数据结构好处是并发执行过程中所有线程的执行时间是有上界的, 这有助于减少内存请求的延迟.

wfslab 系统接口层使用线程私有堆和透明巨页机制. 向操作系统申请内存页必须通过系统调用. 一方面, 系统调用可能会触发操作系统内核的调度行为; 另一方面, 现代操作系统分配给用户的内存页通常是虚拟的, 当进程读写该内存页时, 操作系统产生缺页中断, 在缺页中断的处理过程中操作系统才会为用户真正分配物理页. 这些操作系统的行为可能会造成内存请求延迟的增加. 为此, wfslab 使用线程本地的堆, 尽可能减少向系统请求内存页的次数, 同时使用透明巨页机制降低 minor 缺页中断发生次数. 此外, 使用透明巨页也有助于减少硬件 TLB 未命中的概率, 降低访存延迟.

wfslab 分配器结构的抽象模型可表示为:

mmap_heap(faa_heap_growth=1, threadlocal_heap_size=256MB, huge_page=1)::segfit_lf(sizeclass=log2b, size_handle=[65536, SIZE_MAX))::slab_wf(sizeclass=prime, slab_size=64KB, tlslabs=40, size_handle=[0, 65536))::malloc_socket.

4 实验分析

与 HeapLayer 相比, 榫卯内存分配框架一大改进是提供额外的策略组合与定制. 这为构造内存分配器提供了更多的自由度. 本节的实验将使用 tlsfcc, hslab 与 wfslab 运行内存分配器基准测试, 通过与多个通用和专用内存分配器对比来说明榫卯框架有效性.

4.1 实验环境与设置

实验使用的硬件平台包括一台 8 核 x86/64 设备和一台异构 8 核 aarch64 嵌入式设备, 其硬件规格与软件环境的具体说明如表 3 所示. 在两个不同体系结构的平台进行实验是为了说明榫卯框架的可移植性和兼容性. 在实验平台的系统参数设置上, aarch64 平台的 Linux 内核启用了 EAS (energy-aware scheduling, 能量感知的调度) 和 DVFS (dynamic voltage and frequency scaling, 动态电压与频率调整), 其中 DVFS 功能会根据工作负载动态调整核心频率, EAS 会根据当前负载将任务动态迁移到不同的异构核心. 为了减少动态频率调整对实验结果稳定性的影响, 本文在两个实验平台均开启了 performance 频率策略, 强制操作系统内核使用最大核心频率运行负载. 此外, 由于 aarch64 平台的 Linux 内核没有启用透明巨页支持, wfslab 使用透明巨页请求实际上获得到只是 4 KB 的内存页.

表 3 实验平台说明

配置	x86/64实验平台	aarch64实验平台
CPU型号	AMD Ryzen 3700x	Qualcomm Snapdragon 888+
CPU核心数	8核16线程 (SMT开启)	异构8核8线程
CPU组成	分为两个CCX簇, 每簇4核4.2 GHz	1x Cortex-X1 3.0 GHz, 3x Cortex-A78 2.4 GHz, 4x Cortex-A55 1.8 GHz
Cache	每簇共享16 MB L3 cache	共享4 MB L3 cache
内存	16 GB DDR4-3800 MHz	12 GB LPDDR5-3200 MHz
操作系统	Linux 5.15.12 / Fedora 35	Linux 5.4.86 / Android 12 / Ubuntu 21.10运行在container中
编译器	GCC 11.2.1 / Clang 13.0.0	GCC 11.2.0 / Clang 13.0.0

参与对比的其他内存分配器及其代码版本在表 4 中会呈现, 其中分配器简称的对应关系如下: sys (glibcmalloc), tc (tcmalloc), je (jemalloc), hd (Hoard), tbb (tbbmalloc), sn (snmalloc), mi (mimalloc), smi (smimalloc), dh (Diehard). 这些内存分配器均采用 mimalloc-bench (https://github.com/daanx/mimalloc-bench) 开源库的编译脚本进行编译. 相关研究中提到的 Mesh 和 supermalloc 分配器无法在 aarch64 平台编译运行, 因此无法参与实验对比. 对于通过榫卯框架构造的内存分配器 tlsfcc, hslab 和 wfslab, 本文使用 gcc -fPIC 命令编译为动态链接库, 启用-O3 优化参数. 其中 x86/64 平台编译使用-mno-see 参数禁止使用 SSE 寄存器, 避免操作系统上下文切换时保存 SSE 寄存器的额外开销对实验结果造成影响. 同时, 由于 wfslab 中使用的 treiber 无锁栈的 x86 实现依赖 CAS2 原子操作, 必须增加-mcx16 参数允许使用 CMPXCHG16b 指令. aarch64 平台编译需要增加-march=armv8-a+lse 参数, 允许编译器使用 armv8.1a 新增的 CAS 和 LDADD 原子操作指令, 确保部分层级函数中使用的原子操作原语被正确地编译. 此外, 编译 TLSF 依赖计算前导零函数, x86 上有两种指令都能实现该函数, 其中 LZCNT 花费的 CPU 周期更短, 因此需要增加-mlzcnt 参数启用 LZCNT 指令.

表 4　平均执行时间与最大 RSS 的归一化几何平均数

内存分配器	版本	AMD Ryzen 3700x		Snapdragon 888+	
		执行时间	RSS	执行时间	RSS
sys	2.34	1.292	0.978	1.566	1.060
tlsf	—	16.125	0.694	9.284	0.824
tc	gperftools-2.9.1	1.545	1.095	1.362	1.322
je	5.2.1	1.170	1.029	1.259	1.174
hd	5afe855	1.280	1.302	1.340	1.445
tbb	883c2e5	1.313	0.994	1.570	1.076
sn	0.5.3	0.869	1.121	0.889	1.244
mi	v1.7.3	0.827	1.028	0.910	1.162
tlsfcc	—	9.119	0.757	5.854	0.830
hslab	—	1.075	1.035	1.157	1.148
wfslab	—	1.000	1.000	1.000	1.000
wfslabfl	—	1.005	0.993	0.994	1.004
wfslabm	—	1.002	0.993	1.006	1.002
scudo	ed56007	2.463	0.967	2.856	1.091
smi	v1.7.3	1.057	1.271	1.294	1.334
mng	2ed5881	11.206	0.688	7.649	0.773
dh	1d08836	30.294	1.228	30.063	1.485
wfslabs	—	1.135	1.106	1.340	1.102
hslabd	—	1.200	1.162	1.283	1.266
tlsfccsd	—	10.066	0.841	7.736	1.068

为验证榫卯框架本身的开销, 本文选取 wfslab 作为基准, 将该分配器算法手动编码实现为 wfslabm 分配器, wfslabm 各项参数均与 wfslab 保持一致, 通过对比 wfslab 和 wfslabm 性能可得出使用榫卯框架搭建内存分配器的开销. 同时, 为了进一步证明榫卯框架生成中间函数的开销可以被编译器优化消除, 本文编写了一个 fake 附加层进行验证. 该附加层在判断内存请求大小后, 无论判断条件是否成立均将请求转交给下一层. 对于这种实质上不改变分配器行为的额外层级, C 编译器优化机制会消除无意义的判断和额外的函数调用. 为了验证这一点, 本文使用该附加层搭建了内存分配器 wfslabfl 进行对比, 该分配器在 wfslab 的每一个非系统接口层之上额外连接一个 fake 附加层.

此外, 本文为 wfslab, hslab 和 tlsfcc 增加了额外的安全附加层函数, 分别命名为 wfslabs, hslabd 和 tlsfccsd. 其中 wfslabs 增加了一个使用异或校验策略的内存块尾部校验层 tail_chksum(chk_sum=xor, err_handle=default), 用于检测缓冲区溢出, 异或校验和大小为 8 字节, 使用默认的错误处理策略: 只输出错误但不终止程序. hslabd 增加了

一个使用内存块头部魔数检验的延迟释放层 delayed(chk_sum=magic, node_xor=1, delayed_size=64KB, err_handle=fatal), 用于检测重复释放和释放后使用行为, 该层级使用线程私有链表管理延迟释放的内存, 最大管理的内存大小为 64 KB, 按 FIFO 顺序换出. 链表节点记录的指针和内存块大小使用异或编码, 以一定概率避免被用户非法内存修改覆盖. 同时, 该层使用致命错误处理策略, 一旦检测出错误立刻终止程序. tlsfccsd 则同时增加了上述两个层级.

4.2 基准测试集与评价指标

由于内存分配器的设计通常是执行时间与内存占用的取舍, 评估通用内存分配器通常使用平均执行时间和内存利用率两个性能指标. 不过, 软实时系统应用动态内存分配器也较为常见, 如监控系统与移动通信基站软件. 在这些系统中, 内存分配器不仅要求平均执行时间低, 内存利用率高, 还要求有确定的最差情况内存请求延迟, 以避免实时任务执行超时. 因此本文还将对分配器的最差内存请求延迟进行评估.

本文通过运行基准测试对比内存分配器的平均执行时间与内存利用率, 实验中使用的基准测试主要来自开源库 mimalloc-bench, 其中包含多组实际应用程序基准测试程序与不同场景的压力测试负载. 本文用于对比的基准测试负载有:

(1) cfrac: 单线程科学计算应用, 用于计算整数的质因数分解, 存在大量的小内存申请与释放.

(2) barnes[41]: 来自 SPLASH-2 并行计算基准测试集, 用于模拟 163 840 个粒子的引力.

(3) espresso[42]: 单线程可编程逻辑数组分析器, 存在一些 realloc 内存分配场景.

(4) redis: 内存数据库服务应用, 使用最大硬件线程并发连接数进行 10^6 次插入 10 个元素并且请求 10 个元素.

(5) lean[20]: 调用 lean 编译器编译 lean 标准库.

(6) cxqueue[43]: 并发队列数据结构的负载, 分别使用 msqueue[28], wfqueue[30]与 crtqueue[44]进行 10^6 次并发入队和出队操作, 每批入队 16 384 个节点.

(7) larson[45]: 模拟真实服务器上观察到的大量线程的并发申请和释放工作负载, 其中许多动态内存对象并非由申请线程释放.

(8) cscratch[18]: 主要用于测量被动 cache 伪共享 (false-sharing) 的微基准测试, 其中线程分配小对象并交给其他线程访问对象, 如果不同线程拥有的对象处于一个 cache 行则会导致 cache 行竞争.

(9) xmalloc-test: 该微基准测试使用 100 个分配线程和 100 个释放线程进行内存请求与释放.

(10) rptest: 来自 rpmalloc 的测试程序, 存在大量内存的并发申请和释放, 部分内存请求使用对齐分配.

(11) glibc-simple/glibc-thread: glibc malloc 使用的基准测试程序, 前者为单线程, 后者为多线程.

(12) sh6bench/sh8bench: 来自 SmartHeap 的基准测试, 其释放模式有一部分是后进先出顺序, 一部分则是先进先出顺序释放. 其中还有一部分内存是由非申请线程释放的.

(13) alloc-test: 内存分配压力测试, 随机分配大小服从帕累托分布.

(14) mstress[20]: mimalloc 中用于模拟真实服务器释放模式的负载, 分配对象会在多个线程间迁移, 其中线程在结束前不会将所有对象释放, 换言之, 有一些对象会在线程结束后依然存活.

此外, 本文使用一种微基准测试 wcet_test 来测量最差情况内存请求延时. 参与实验对比的所有内存分配器在单线程的情况下的最差内存请求延迟理论上都是有界的. 然而, 在多核并发竞争过程中由于部分分配器使用了非公平自旋锁或是无锁同步技术, 这些同步方式不能保证线程无饥饿, 因此并发场景下理论最差情况执行时间是不确定的. 从实际系统的应用场景角度来看, 等待锁的过程中也不可避免会触发上下文切换, 调度和核心迁移, 这些操作系统调度行为会对内存请求延迟产生不利影响. 评估各分配器的最差内存请求执行时间需要测量内存分配器线程间并发竞争最紧张的情况下的执行时间, 这在大多数实际应用中都不会有这种极端的分配释放场景. 即使是存在大量小对象分配与释放的无等待/无锁数据结构测试负载 cxqueue, 其最大吞吐率也仅为数百万请求每秒. 为了模拟这种极端情况下的分配释放, 本文综合 xmalloc-test, larson 和 cscratch 测试基准中的设计思想, 设计了一种并发压力微基准测试, 该微基准测试以最大硬件线程数执行, 每个线程执行一千万次相同大小阶的分配与释放. 每个线程的申请释放由多个批次构成, 每批次负载申请内存大小是不对称的, 单个线程最多一次性执行总大小 8 MB

的内存申请. 所有分配的内存对象被打乱顺序后发布到一个全局链表中由其他线程远程释放, 这样能尽可能耗尽内存分配器的线程本地缓存. 测试总共 7 组, 每组对应的分配大小阶以 2 倍增长, 依次为 (16, 32, 64,…, 2048) 字节. 由于各线程分配内存大小是非对称的, 分配都是针对一个大小阶的, 且绝大部分释放都是非申请线程处理的远程释放, 线程间竞争得以最大化.

4.3 实验方法

● 平均执行时间和最大 RSS 的测量方法. 本文运行基准测试负载时使用 LD_PRELOAD 命令加载不同的内存分配器动态库, 通过/usr/bin/time 命令测量程序执行时间以及执行过程中的最大 RSS (resident set size, 驻留集大小). 基准测试程序执行时间越短说明内存分配器越快. 而 RSS 可认为是进程实际使用物理内存大小. 由于测试负载中内存请求次数和大小通常是确定的, RSS 可视为分配器内存利用率的指标. RSS 越小说明分配器的内存利用率越高.

● 最差情况内存请求延迟的测量方法. 由于测量内存请求延迟的行为本身会产生延迟, 本文无法精确测量每一个内存分配器的最差情况内存请求延迟. 因此, 本文使用最差情况内存请求延迟时间分布的百分位数统计值作为其指标. 具体的测量方法是: 使用榫卯框架构建出一个测量内存请求时间的库 wcet_stat_malloc, 该库为 malloc 增加一个 wcet_stat 统计层函数, 其结构为: malloc_heap::wcet_stat. 该库与测试程序静态链接后, 再加载动态内存分配器库即可测量并统计内存请求延迟的百分位时间分布.wcet_stat 层函数通过硬件指令插桩测量内存申请和释放的 CPU 周期, 考虑到测量函数本身的执行可能影响到内存请求的速度从而降低并发竞争程度, 例如在 x86 上读取硬件周期计数器会产生读内存屏障 (read memory barrier) 阻止 CPU 的乱序执行, 本文使用每 8 次内存请求采样 1 次的方式降低测量对结果的影响. 同时, 为了尽可能排除内核调度器的影响因素, wcet_stat 层通过配置线程的核心亲和性, 强制将创建的线程固定在每个 CPU 逻辑核心上. 此外, 为了简化测量时的结果存储, wcet_stat 使用了一个周期数组记录结果, 测量数组的最大值的周期对应的时间为约为 1 048 576 ns, 超过最大值的请求时间被视为最大值记录, 测量完成后将统计的所有 CPU 执行周期转换为微秒.

4.4 实验结果与分析

本文分别在 x86/64 和 aarch64 实验平台加载内存分配器动态库运行基准测试, 总共执行 5 轮, 由此计算平均值与标准差, 得到的结果如图 5–图 8 所示. 其中图 5 和图 6 是执行时间的对比, 图中 x 轴是基准测试项, y 轴是归一化的执行时间, 即各分配器执行时间与 wfslab 执行时间的比值, 比值越小意味着更好的性能. 图 7 和图 8 是最大 RSS 的对比, 图中 y 轴是归一化的最大 RSS, 即各分配器 RSS 与 wfslab 最大 RSS 的比值, 比值越小说明内存利用率越高. 本文计算了实验结果的平均执行时间与最大 RSS 的几何平均数, 结果同样以 wfslab 的数据为基础归一化, 用于综合衡量各分配器的平均执行时间和内存利用率, 如表 4 所示.

为了对比采用内存漏洞缓解措施的安全分配器, 本文使用 scudo, smi, mhg, dh, wfslabs, hslabd 和 tlsfccsd 同样运行了上述实验, 计算其实验结果的几何平均数, 如表 4 所示.

本文在 x86/64 与 aarch64 硬件平台分别运行了 8 次最差情况内存请求延迟微基准测试负载, 收集测量结果并对结果求平均值和标准差, 结果如图 9、图 10 所示. 图 9、图 10 的横坐标为百分位数, 纵坐标是百分位数对应的内存请求延迟, 单位为 μs. 图中描点的数据从左到右依次是最耗时的 10%, 1%,…, 0.000 1% 的内存请求延迟. 需要注意的是, 由于 aarch64 平台运行 Android 系统, 系统中存在大量的内核线程和传感器产生的中断, 导致线程运行过程中出现较多的抢占和上下文切换, 因此在 aarch64 平台上测得的部分分配器内存请求延迟的大小与标准差比 x86/64 硬件平台更高.

本文将依据实验结果, 围绕以下 5 个问题进行分析, 以说明榫卯框架的有效性.

● 问题 1: 对比 wfslab 分配器, 手动编码实现的 wfslabm 分配器与插入额外 fake 附加层的 wfslabfl 分配器, 说明榫卯框架中函数组合的性能开销.

● 问题 2: 对比 tlsfcc 分配器与原始 TLSF 分配器, 说明定制同步策略能否带来性能提升.

● 问题 3: 对比 hslab 分配器与相似架构的通用内存分配器, 说明定制核心感知的线程缓存能否带来性能提升.

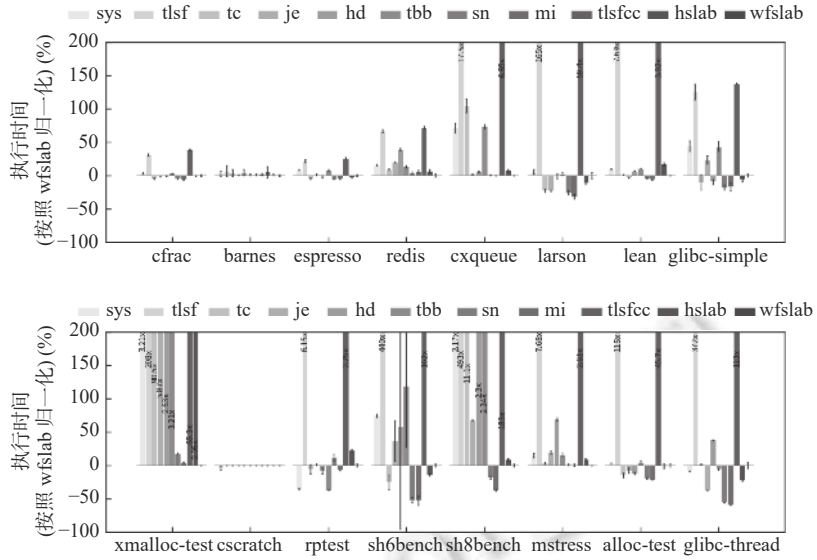

图 5 x86/64 Ryzen 平台运行基准测试的执行时间

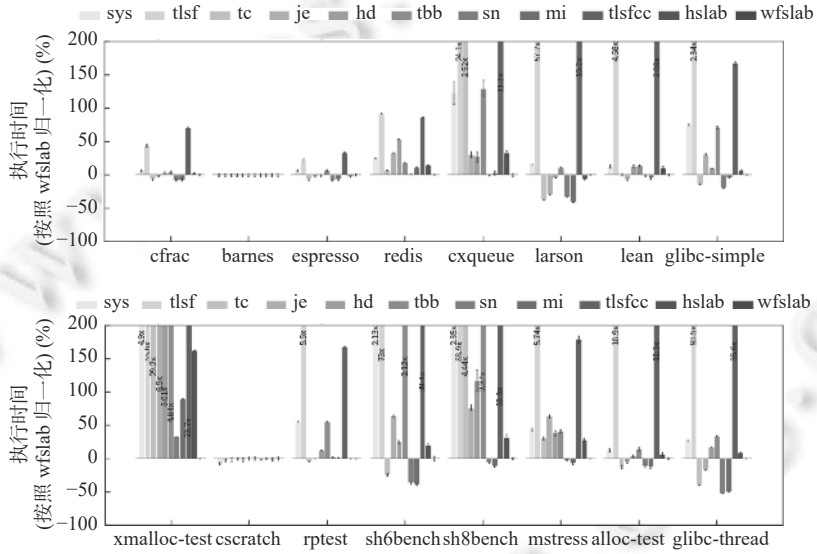

图 6 aarch64 Snapdragon 平台运行基准测试的执行时间

● 问题 4: 对比 wfslab 分配器与通用内存分配器, 说明通过对分配器层级函数和策略函数的系统设计与定制能否减少内存请求延迟.

● 问题 5: 通过为 tlsfcc, hslab, wfslab 增加内存漏洞缓解措施的附加层, 进一步说明榫卯框架的组合性以及低组合开销.

● 关于问题 1 的分析. 榫卯框架依赖 C 宏预处理器的元编程特性, 通过生成中间函数的方式完成层级函数的复合, 但该方式可能会造成额外的函数调用过程. 函数调用过程中的 prologue 和 epilogue 步骤通常包含多组产生访存操作的栈操作指令. 对于频繁调用的内存请求接口而言, 额外的操作如访存或者条件判断都有可能会造成明显的性能下降. 如 supermalloc 与 snmalloc 通过消除大小阶映射函数中的条件分支就能显著改善单线程瓶颈. 不过, 现代 C 编译器优化通常可以将函数调用过程展开并消除不必要的函数调用与重复条件判断. 根据表 4 中的实验结果, 比较基于榫卯框架实现的 wfslab, 手动编码实现的 wfslabm 以及额外插入 fake 附加层的 wfslabfl 可以得

到: 三者运行基准测试的执行时间和最大 RSS 的几何平均数无显著差别, 其最大差距不超过 0.7%. 同时, 在 aarch64 平台的实验中 wfslabfl 甚至取得了比 wfslab 执行时间几何平均数低 0.6% 的结果. 因此, 三者的差距可认为是 5 次实验运行过程中受到操作系统以及其他进程运行影响而出现的波动. wfslabfl 和 wfslab 分配器的区别在于其额外插入的多个 fake 层函数调用以及为复合这些函数而生成的中间函数, 实验结果说明生成中间函数的开销和无意义的分支判断对性能的影响可被编译器优化消除. 同时, 对 wfslab 与 wfslabfl 编译生成的.so 动态库文件执行 objdump 命令进行反汇编分析发现两者所有的中间函数符号均被消除, 且最终生成 malloc(3) 接口函数长度以及汇编代码都保持一致. 这也从另一个角度证明不必要的函数调用开销与分支判断可以被编译器优化完全消除. 另外, wfslab 和 wfslabm 算法和参数均相同, 而后者通过手动编写实现. 两者运行时的区别在于 wfslab 需要在栈上构造内存请求上下文 mctx 并在层级函数间传递. 实验结果说明该开销基本可被编译器优化为使用寄存器消除. 综上所述, 实验结果说明榫卯框架实现了层级函数组合的零性能开销.

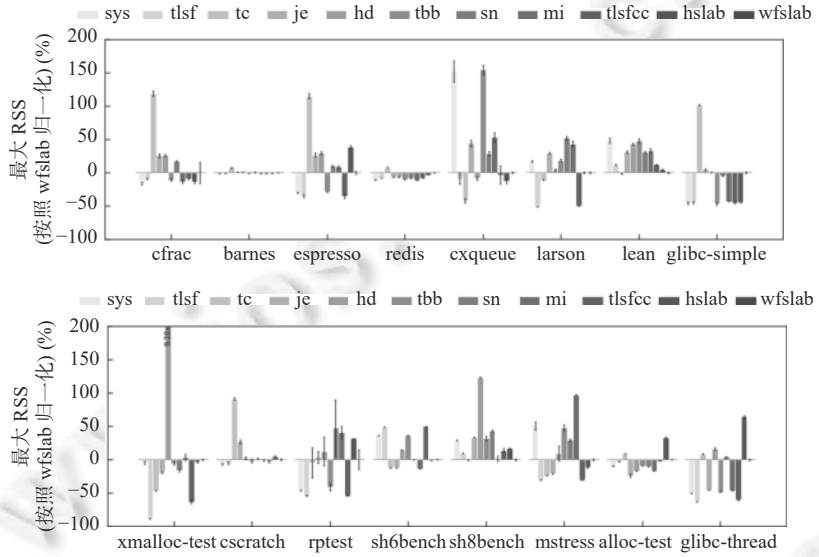

图 7 x86/64 Ryzen 平台运行基准测试的最大 RSS

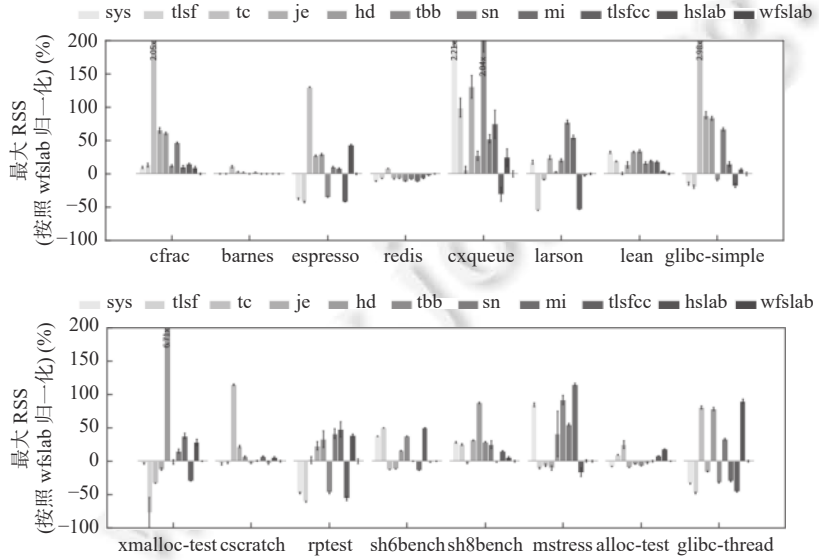

图 8 aarch64 Snapdragon 平台运行基准测试的最大 RSS

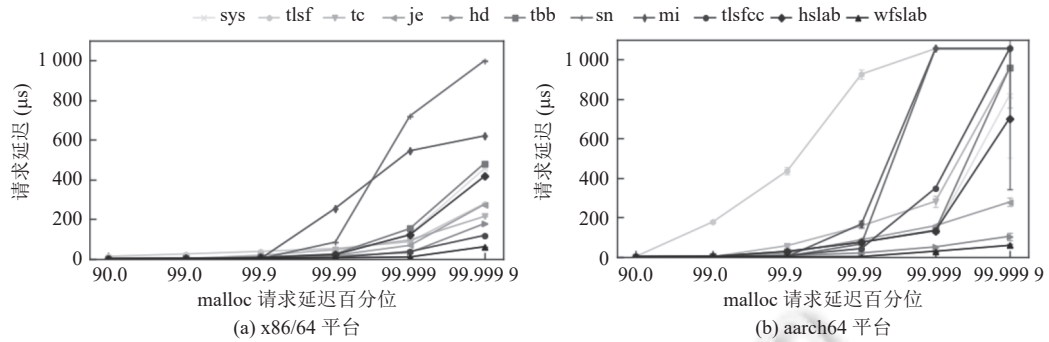

图 9 malloc 请求延迟百分位数

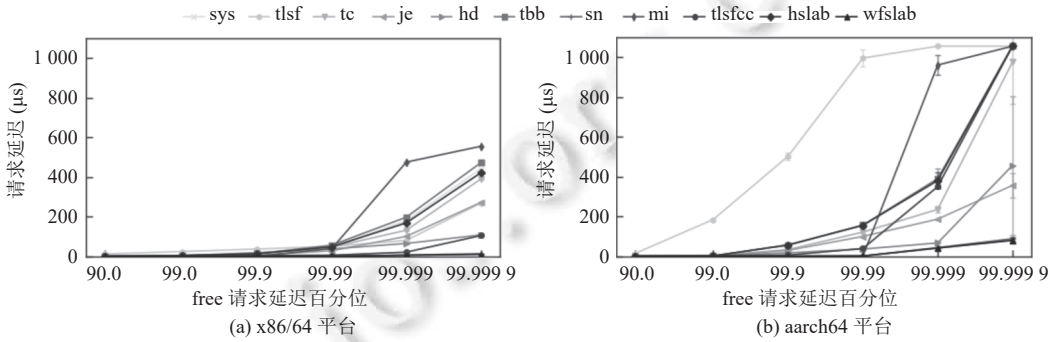

图 10 free 请求延迟百分位数

● 关于问题 2 的分析. TLSF 算法采用一个全局共享的单一堆, 它的性能主要受竞争线程数影响, 竞争线程越多, 性能越差. 但另一方面, 单一堆的内存请求是可序列化的, 由并发访问导致的额外内存占用较少, 因此内存占用也较其他采用多堆设计的通用内存分配器低. 对比使用 ccsynch 同步策略的 tlsfcc 与使用 POSIX 互斥锁的原始 TLSF 实现可以发现: 根据表 4 中所有基准测试项的平均执行时间的几何平均数, tlsfcc 在 x86/64 和 aarch64 实验平台上分别取得了 1.76 和 1.59 的加速比. 从图 5 和图 6 中各基准测试项的详细结果来看, tlsfcc 在存在大量多线程并发请求内存的基准测试负载中表现出较大优势, 如 cxqueue, larson, mstress, xmalloc-test 等基准测试负载的执行时间远小于原始 TLSF 实现, 而在单线程基准测试负载中的表现几乎与原始 TLSF 实现相同. 造成性能差异的原因主要是 ccsynch 同步算法能在全局锁竞争激烈时合并同步请求, 从而大大提升并发吞吐率. 同时, 从图 9, 图 10 的内存请求延迟来看, 由于多核并发吞吐率的提升, tlsfcc 最差情况内存请求延迟的百分位数分布相比原始 TLSF 实现也有明显的改善. 不过, 使用 ccsynch 同步算法也是有代价的, 由表 4 的最大 RSS 几何平均数可以看到: 由于该同步算法的每线程同步请求结构体占据了额外的内存空间, 因此分别在 x86/64 和 aarch64 实验平台上产生 9.0% 和 0.7% 的额外最大 RSS. 然而, 与多核性能的显著提升相比, tlsfcc 产生轻微的内存占用提升是可以接受的. 根据 tlsfcc 的实验结果, 本文可以得出: 将内存分配算法使用的同步方式抽象为同步策略, 依据应用场景对同步策略进行定制, 可以有效提升分配器在多核场景下的性能.

● 关于问题 3 的分析. hslab 的整体架构与 tcmalloc 相似, 但在各层级的同步策略以及线程缓存上做了更精细的定制. 由表 4 中的数据可以得出, hslab 在 x86/64 和 aarch64 实验平台上的基准测试的执行时间几何平均值仅为 tcmalloc 的 69.6% 和 85.0%, 也优于参与实验的所有基于锁的内存分配器. 由图 5 和图 6 可以看到, hslab 在大多数基准测试项中与 tcmalloc 表现接近, 但在存在较高并发压力的 xmalloc-test 和 sh8bench 基准测试项, hslab 取得了显著优势, 这主要是因为 hslab 依据并发压力在各层级使用合适的同步策略. 同时, 由于 hslab 对线程缓存大小进行核心感知的定制, 其内存利用率也有较大改善. 在 x86/64 实验平台上 hslab 使用所有核心相等份额的线程缓存, 其最大 RSS 的几何平均值为 tcmalloc 的 94.5%; 而在 aarch64 实验平台上, hslab 依据异构核心信息动态定制线程

缓存, 其最大 RSS 的几何平均值为 tcmalloc 的 86.8%. 两者对比减少了 7.7% 的内存占用, 说明核心感知的线程缓存能够改善内存分配器的内存利用率.

● 关于问题 4 的分析. wfslab 是高性能, 低延迟的无锁/无等待分配器, 它通过组合低延迟的层级函数以及每个层级的定制, 尽可能降低最差情况内存请求延迟. 由图 9, 图 10 可知, wfslab 取得了所有内存分配器中最低的最差情况内存请求延迟. 其 99.9999% 内存 malloc/free 请求的延迟分布百分位对应的延迟仅为 61.1 μs/10.3 μs (x86/64 平台) 和 58.3 μs/80.3 μs (aarch64 平台). 同时, 由表 4 可以得到, wfslab 的基准测试执行时间的几何平均值小于所有基于锁的分配器, 但与目前最先进的无锁分配器 mimalloc 和 snmalloc 还存在差距. 具体地, 从图 5 和图 6 可以看出, 在存在线程远程释放情形的 larson, sh6bench, sh8bench 以及 glibc-thread 基准测试项上, wfslab 的执行时间要比 snmalloc 和 mimalloc 略高. 这是因为 wfslab 为降低最差情况内存请求延迟, 舍弃了远程释放缓存批量释放的设计. 虽然批量释放有助于提升平均性能, 但释放过程中线程可能会频繁访问大量空闲内存块, 造成较多的 cache 未命中和 TLB 未命中, 从而导致最差情况内存请求延迟增加. 不过, 同样因为 wfslab 未采用远程释放缓存的设计, 其内存利用率也有所改善, 其最大 RSS 的几何平均值相较于 mimalloc/snmalloc 低 2.8%/12.1% (x86/64 平台) 和 16.2%/24.4% (aarch64 平台). 此外, 由图 5 和图 6 可以看出 wfslab 在单线程的基准测试项的表现稍差, 例如 cfrac, espresso, glibc-simple. 这主要是因为 wfslab 即使在单线程场景下也使用相对耗时的原子操作. wfslab 的实验结果表明立足于榫卯框架的组合性和定制性, 通过系统的设计取舍, 能够有效降低分配器的最差情况内存请求延时.

● 关于问题 5 的分析. 参与实验对比的内存安全分配器各自采用了不同的内存漏洞缓解措施, 这些内存漏洞缓解措施可能对分配器执行时间和内存占用造成较大的影响, 因此本文无法直接对分配器进行对比. 不过, 通过比较增加附加层的 wfslabs, hslabd 与 tlsfccsd 与原始的 wfslab, hslab 和 tlsfcc 可以得到这些附加层的开销. 根据表 4 中的执行时间和最大 RSS 几何平均值, 比较 wfslab 和 wfslabs 可以发现, 内存块尾部增加异或校验字的 tail_chksum 附加层的执行时间/内存占用开销为 13.5%/10.6% (x86/64 平台) 和 34.0%/10.2% (aarch64 平台); 比较 hslab 和 hslabd 可以发现, 采用 64 KB 释放隔离区和内存块头部魔数校验的 delayed 附加层的执行时间/内存占用开销为 11.6%/12.3% (x86/64 平台) 和 10.9%/10.3% (aarch64 平台); 比较 tlsfcc 和 tlsfccsd 可以发现, 采用以上两种附加层的执行时间/内存占用开销为 10.4%/11.1% (x86/64 平台) 和 32.1%/28.7% (aarch64 平台). 实验结果显示, 叠加两个附加层的开销并非线性的. 同时, 附加层的开销在 x86/64 和 aarch64 两个实验平台也显示出了显著的差别. 这可能是受 CPU 指令流水线, 硬件 cache 大小与分配算法的影响. 为了比较附加层的组合开销, 本文选取 mimalloc 与 smimalloc 作为对比, smimalloc 在空闲链表中使用异或校验, 其执行时间/内存占用开销为 27.8%/23.6% (x86/64 平台) 和 42.2%/14.8% (aarch64 平台), 远高于 wfslabs 和 hslabd 的开销. 至此实验结果显示, 榫卯框架能在不改变原始算法的情况下, 通过组合附加层级, 以较低的开销为内存分配器增加可定制的内存漏洞缓解措施和错误处理策略.

5 总 结

本文从函数式编程视角审视内存分配流程, 基于函数可组合性提出了一种可组合的定制化内存分配框架榫卯. 榫卯框架将系统内存分配抽象为多个互不耦合的内存分配函数的组合, 这些内存分配函数可以扩展出策略槽提供更高的组合性和定制性. 该内存分配框架基于标准 C 实现, 依赖 C 预处理器提供的元编程特性实现层级函数组合的零性能开销. 开发者能够通过组合与定制分配器的层级函数, 快速构建出适合应用场景的内存分配器. 本文使用榫卯框架构建了 3 种为不同应用场景优化的内存分配器实例, 通过运行基准测试与现有内存分配器进行对比. 实验结果证明了榫卯框架的有效性.

References:

[1] Donahue SM, Hampton MP, Deters M, Nye JM, Cytron R, Kavi KM. Storage allocation for real-time, embedded systems. In: Proc. of the 1st Int'l Workshop on Embedded Software. Tahoe City: Springer, 2001. 131–147. [doi: 10.1007/3-540-45449-7_10]

[2] Puaut I. Real-time performance of dynamic memory allocation algorithms. In: Proc. of the 14th Euromicro Conf. on Real-time Systems. Euromicro RTS 2002. New York: IEEE, 2002. 41–49. [doi: 10.1109/EMRTS.2002.1019184]

[3] Kanev S, Darago JP, Hazelwood K, Ranganathan P, Moseley T, Wei GY, Brooks D. Profiling a warehouse-scale computer. In: Proc. of

the 42nd Annual Int'l Symp. on Computer Architecture. Oregon: Association for Computing Machinery, 2015. 158–169. [doi: 10.1145/2749469.2750392]

[4] Hunter AH, Kennelly C, Turner P, Gove D, Moseley T, Ranganathan P. Beyond malloc efficiency to fleet efficiency: A hugepage-aware memory allocator. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation. New York: USENIX Association, 2021. 257–273.

[5] Berger ED, Zorn BG, McKinley KS. Composing high-performance memory allocators. In: Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Utah: Association for Computing Machinery, 2001. 114–124. [doi: 10.1145/378795.378821]

[6] Yun H, Mancuso R, Wu ZP, Pellizzoni R. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In: Proc. of the 19th IEEE Real-time and Embedded Technology and Applications Symp. Berlin: IEEE, 2014. 155–166. [doi: 10.1109/RTAS.2014.6925999]

[7] Herter J, Backes P, Haupenthal F, Reineke J. CAMA: A predictable cache-aware memory allocator. In: Proc. of the 23rd Euromicro Conf. on Real-time Systems. Porto: IEEE, 2011. 23–32. [doi: 10.1109/ECRTS.2011.11]

[8] Qiu JF, Hua ZH, Fan J, Liu L. Evolution of memory partitioning technologies: Case study through page coloring. Ruan Jian Xue Bao/Journal of Software, 2022, 33(2): 751−769 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6370.htm [doi: 10.13328/j.cnki.jos.006370]

[9] Roghanchi S, Eriksson J, Basu N. Ffwd: Delegation is (much) faster than you think. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: Association for Computing Machinery, 2017. 342–358. [doi: 10.1145/3132747.3132771]

[10] Hendler D, Incze I, Shavit N, Tzafrir M. Flat combining and the synchronization-parallelism tradeoff. In: Proc. of the 32nd Annual ACM Symp. on Parallelism in Algorithms and Architectures. Santorini: Association for Computing Machinery, 2010. 355–364. [doi: 10.1145/1810479.1810540]

[11] Fatourou P, Kallimanis ND. Revisiting the combining synchronization technique. In: Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Louisiana: Association for Computing Machinery, 2012. 257–266. [doi: 10.1145/2145816.2145849]

[12] Dice D, Marathe VJ, Shavit N. Flat-combining NUMA locks. In: Proc. of the 33rd Annual ACM Symp. on Parallelism in Algorithms and Architectures. California: Association for Computing Machinery, 2011. 65–74. [doi: 10.1145/1989493.1989502]

[13] Luchangco V, Nussbaum D, Shavit N. A hierarchical CLH queue lock. In: Proc of the 12th Int'l Conf. on Parallel Processing. Dresden: Springer, 2006. 801–810. [doi: 10.1007/11823285_84]

[14] Lozi JP, David F, Thomas G, Lawall JL, Muller G. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In: Proc. of the 2012 USENIX Conf on Annual Technical Conf. Boston: USENIX Association, 2012. 65–76.

[15] Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems, 1991, 9(1): 21–65. [doi: 10.1145/103727.103729]

[16] Bracha G, Cook W. Mixin-based inheritance. In: Proc. of the 1990 European Conf. on Object-oriented Programming Systems, Languages, and Applications. Ottawa: ACM, 1990. 303–311. [doi: 10.1145/97945.97982]

[17] Masmano M, Ripoll I, Crespo A, Real J. TLSF: A new dynamic memory allocator for real-time systems. In: Proc. of the 16th Euromicro Conf. on Real-time Systems, 2004. ECRTS 2004. Catania: IEEE, 2004. 79–88. [doi: 10.1109/EMRTS.2004.1311009]

[18] Berger ED, McKinley KS, Blumofe RD, Wilson PR. Hoard: A scalable memory allocator for multithreaded applications. In: Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Cambridge: Association for Computing Machinery, 2000. 117–128. [doi: 10.1145/378993.379232]

[19] Kukanov A, Voss MJ. The foundations for scalable multi-core software in Intel threading building blocks. Intel Technology Journal, 2007, 11(4): 309–322.

[20] Leijen D, Zorn BG, de Moura L. Mimalloc: Free list sharding in action. In: Proc. of the 17th Asian Symp. on Programming Languages and Systems. Nusa Dua: Springer, 2019. 244–265. [doi: 10.1007/978-3-030-34175-6_13]

[21] Liétar P, Butler T, Clebsch S, Drossopoulou S, Franco J, Parkinson MJ, Shamis A, Wintersteiger CM, Chisnall D. Snmalloc: A message passing allocator. In: Proc. of the 2019 ACM SIGPLAN Int'l Symp. on Memory Management. Phoenix: Association for Computing Machinery, 2019. 122–135. [doi: 10.1145/3315573.3329980]

[22] Berger ED, Zorn BG. Diehard: Probabilistic memory safety for unsafe languages. In: Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Ottawa: Association for Computing Machinery, 2006. 158–168. [doi: 10.1145/1133981.1134000]

[23] Dice D, Garthwaite A. Mostly lock-free malloc. In: Proc. of the 3rd Int'l Symp. on Memory Management. Berlin: Association for Computing Machinery, 2002. 163–174. [doi: 10.1145/512429.512451]

[24] Michael MM. Scalable lock-free dynamic memory allocation. In: Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Washington: Association for Computing Machinery, 2004. 35–46. [doi: 10.1145/996841.996848]

[25] Aigner M, Kirsch CM, Lippautz M, Sokolova A. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In: Proc. of the 2015 ACM SIGPLAN Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. Pittsburgh: Association for Computing Machinery, 2015. 451–469. [doi: 10.1145/2814270.2814294]

[26] Kuszmaul BC. SuperMalloc: A super fast multithreaded malloc for 64-bit machines. In: Proc. of the 2015 Int'l Symp. on Memory Management. Portland: ACM, 2015. 41–55. [doi: 10.1145/2754169.2754178]

[27] Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing, 2010, 70(1): 1–12. [doi: 10.1016/j.jpdc.2009.08.011]

[28] Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing. Philadelphia: Association for Computing Machinery, 1996. 267–275. [doi: 10.1145/248052.248106]

[29] Timnat S, Petrank E. A practical wait-free simulation for lock-free data structures. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programmings. Orlando: ACM, 2014. 357–368. [doi: 10.1145/2555243.2555261]

[30] Kogan A, Petrank E. Wait-free queues with multiple enqueuers and dequeuers. In: Proc. of the 16th ACM Symp. on Principles and Practice of Parallel Programming. San Antonio: Association for Computing Machinery, 2011. 223–234. [doi: 10.1145/1941553.1941585]

[31] Fatourou P, Kallimanis ND. Highly-efficient wait-free synchronization. Theory of Computing Systems, 2014, 55(3): 475–520. [doi: 10.1007/s00224-013-9491-y]

[32] Kogan A, Petrank E. A methodology for creating fast wait-free data structures. In: Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New Orleans: Association for Computing Machinery, 2014. 141–150. [doi: 10.1145/2145816.2145835]

[33] Peng YQ, Hao ZY. FA-stack: A fast array-based stack with wait-free progress guarantee. IEEE Trans. on Parallel and Distributed Systems, 2018, 29(4): 843–857. [doi: 10.1109/TPDS.2017.2770121]

[34] Yang CR, Mellor-Crummey J. A wait-free queue as fast as fetch-and-add. In: Proc. of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Barcelona: Association for Computing Machinery, 2016. 16. [doi: 10.1145/2851141.2851168]

[35] Wen HS, Izraelevitz J, Cai WT, Beadle HA, Scott ML. Interval-based memory reclamation. In: Proc. of the 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Vienna: Association for Computing Machinery, 2018. 1–13. [doi: 10.1145/3178487.3178488]

[36] Michael MM. Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. on Parallel and Distributed Systems, 2004, 15(6): 491–504. [doi: 10.1109/TPDS.2004.8]

[37] Nikolaev R, Ravindran B. Universal wait-free memory reclamation. In: Proc. of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. San Diego: Association for Computing Machinery, 2020. 130–143. [doi: 10.1145/3332466.3374540]

[38] Powers B, Tench D, Berger ED, McGregor A. Mesh: Compacting memory management for C/C++ applications. In: Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Phoenix: Association for Computing Machinery, 2019. 333–346. [doi: 10.1145/3314221.3314582]

[39] Herlihy M. Wait-free synchronization. ACM Trans. on Programming Languages and Systems, 1991, 13(1): 124–149. [doi: 10.1145/114005.102808]

[40] Petrank E, Musuvathi M, Steesngaard B. Progress guarantee for parallel programs via bounded lock-freedom. In: Proc. of the 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Dublin: Association for Computing Machinery, 2009. 144–154. [doi: 10.1145/1542476.1542493]

[41] Woo SC, Ohara M, Torrie E, Pal Singh J, Gupta A. The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. of the 22nd Annual Int'l Symp. on Computer Architecture. New York: Association for Computing Machinery, 1995. 24–36. [doi: 10.1145/223982.223990]

[42] Grunwald D, Zorn B, Henderson R. Improving the cache locality of memory allocation. In: Proc. of the 1993 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Albuquerque: Association for Computing Machinery, 1993. 177–186. [doi: 10.1145/155090.155107]

[43] Correia A, Ramalhete P, Felber P. A wait-free universal construction for large objects. In: Proc. of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. San Diego: Association for Computing Machinery, 2020. 102–116. [doi: 10.1145/

3332466.3374523]

[44] Ramalhete P, Correia A. POSTER: A wait-free queue with wait-free memory reclamation. In: Proc. of the 22nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Austin: Association for Computing Machinery, 2017. 453–454. [doi: 10.1145/3018743.3019022]

[45] Larson PA, Krishnan M. Memory allocation for long-running server applications. In: Proc. of the 1st Int'l Symp. on Memory Management. Vancouver: Association for Computing Machinery, 1998. 176–185. [doi: 10.1145/286860.286880]

附中文参考文献:

[8] 邱杰凡, 华宗汉, 范菁, 刘磊. 内存体系划分技术的研究与发展. 软件学报, 2022, 33(2): 751–769. http://www.jos.org.cn/1000-9825/6370.htm [doi: 10.13328/j.cnki.jos.006370]

欧阳湘臻(1994－), 男, 博士生, 主要研究领域为操作系统, 嵌入式实时系统.

史先琛(1993－), 男, 博士生, 主要研究领域为嵌入式实时系统, 工业互联网.

朱怡安(1961－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为嵌入式实时系统, 工业互联网, 并行计算.