动态内存分配器研究综述

刘 翔[1] 童 薇[1),2)] 刘景宁[1),2)] 冯 丹[1),2)] 陈劲龙[1)]

[1)](武汉光电国家实验室 武汉 430074)

[2)](信息存储系统教育部重点实验室(华中科技大学计算机科学与技术学院) 武汉 430074)

摘 要 通用动态内存分配器自出现以来一直是系统软件的基本组件.伴随着近些年来多核处理器的发展和新型非易失存储器的出现,关于动态内存分配器的研究也随之聚焦于不同的优化方向,比如多线程环境下的性能优化和针对新型非易失内存介质特性的优化.该文在归纳整理近三十年动态内存分配器的发展和研究状况的同时,对推动内存分配器发展的历史原因进行了分析.此外,作者整理了现有动态内存分配器测试可采用的工作负载和标准测试集,并提出了一套全面、多维度评价内存分配器的指标体系.最后,作者指出了现有工作的优势和面临的缺陷,并探讨了未来内存分配器相关的研究方向,为该领域在今后的发展提供了一定的参考.

关键词 动态内存管理;内存分配器;多线程;非易失存储器;存储技术

中图法分类号 TP393 **DOI 号** 10.11897/SP.J.1016.2018.02359

A Review of Dynamic Memory Allocator Research

LIU Xiang[1)] TONG Wei[1),2)] LIU Jing-Ning[1),2)] FENG Dan[1),2)] CHEN Jin-Long[1)]

[1)](*Wuhan National Laboratory for Optoelectronics*, *Wuhan* 430074)

[2)](*Key Laboratory of Information Storage System* (*School of Computer Science and Technology*, *Huazhong University of Science and Technology*), *Ministry of Education of China*, *Wuhan* 430074)

Abstract Versatile, general-purpose memory allocators have been a fundamental element of software for decades since they emerged. With the development of multicore processor in recent years and the advent of novel non-volatile memory, researches on dynamic memory allocator also focused on different optimization directions such as performance optimization in multithreading environment and optimization aimed at the characteristics of novel non-volatile memory. In this paper, we summarize the development and research status of the dynamic memory allocator in the past three decades and analyze the historical reasons behind the development of memory allocator simultaneously. In the computer history, the years from 1986 to 1994 are known as the golden age of PC(Personal Computer). With the prevalence of the PC, the demand for DRAM(Dynamic Radom Access Memory) is also rising, and the memory allocator has consequently undergone a process from scratch on such historical background. While since 2002, CPU(Central Processing Unit) chip processing technology has gradually approached the limit, and CPU also suffered from the heat problem with the increasing of frequency. In order to ensure the development of CPU performance to continue, CPU began to develop in the direction of multi-core. In order to take full advantage of CPU performance, top-level applications began to take advantage of the

收稿日期:2017-04-19;在线出版日期:2018-01-02.本课题得到国家"八六三"高技术研究发展计划项目基金(2015AA015301,2015AA016701)资助.**刘 翔**,男,1993 年生,硕士研究生,中国计算机学会(CCF)会员,主要研究方向为非易失存储器件、磨损均衡、内存管理、内存分配.E-mail:415228528@qq.com.**童 薇**(通信作者),女,1977 年生,博士,讲师,中国计算机学会(CCF)会员,主要研究方向为海量存储系统、固态存储、I/O 虚拟化.E-mail:tongwei@hust.edu.cn.**刘景宁**,女,1957 年生,博士,教授,中国计算机学会(CCF)会员,主要研究领域为计算机系统结构、计算机存储系统及高速通道接口技术.**冯 丹**,女,1970 年生,博士,教授,中国计算机学会(CCF)会员,主要研究领域为信息存储系统、网络存储、固态存储.**陈劲龙**,男,1995 年生,硕士研究生,主要研究方向为内存管理、内存分配.

multithreading ideas proposed in the 1960s, driving dynamic memory allocators to support multi-threaded memory allocations and ensure memory allocation performance of multi-core processor system. Then since 2010, storage academia has turned their research attention to finding new alternative memory media. Different non-volatile storage media first got rapid development in the field of materials, and then get the attention of the storage academia, bringing the new non-volatile memory research boom. Therefore, the research of dynamic memory allocator has been gradually combined with non-volatile memory technologies. Scholars began to study the challenges brought by the introduction of non-volatile memory to traditional allocators and provided their solutions from different perspectives. It could be seen that the development and research direction of the dynamic memory allocator is closely linked with the historical background at that time, so we can also look forward to the future development direction of the allocator based on this concept. In addition, we coordinate state-of-the-art workloads and benchmarks available for dynamic memory allocator, and propose a set of evaluation indicators on dynamic memory allocator. This set of metrics has 11 dimensions in total that include compatibility, memory footprint, allocation latency, scalability, predictability, non-volatile memory awareness, and so on. We hope to cover as much as possible aspects need to be considered of existing and future dynamic memory allocator design. The evaluation results of the main memory allocator (ptmalloc, TCmalloc, etc.) using the set of reference standards is also given. Besides, we pointed out advantages and disadvantages of current technologies and predicted the directions of further research, and thus could be reference work for the development of this field in future. With the rapid progress of non-volatile memory technologies and continuous improvement of capacity, performance and other aspects of 3D XPoint product, and in the meantime, with the continuous development of machine learning and artificial intelligence, it will bring new changes and revolutions to the field of dynamic memory allocator.

Keywords dynamic memory management; memory allocator; multi-thread; non-volatile memory; storage technology

1 引 言

内存管理是指软件运行时对计算机内存资源进行分配和使用的技术，其主要功能是高效、快速地分配内存资源，并且在适当的时候释放和回收内存资源. 内存管理对计算机系统有着深远的影响. 在《深入理解 Linux 内核》一书中就指出“内存管理是迄今为止 Unix 内核中最复杂的活动”. 内存分配器本质上是内存管理中最核心的部分，它为应用提供了真正实用的分配内存和释放内存的基本功能，是应用与内存交互的主要接口.

鉴于几乎所有的计算机系统都采用了虚拟内存技术，内存在概念上首先可以被分为物理内存和虚拟内存，物理内存是指真实的机器上的内存条(即 DRAM 芯片)，而虚拟内存是指分配给应用程序的一大片虚拟地址空间. 虚拟内存通常比物理内存要大得多，即使程序运行时所需要的内存超过实际物理内存容量，也能通过虚拟内存技术正常运行. 相应地，内存分配器可以分为用户态内存分配器(User-level Memory Allocator，UMA)和内核态内存分配器(Kernel-level Memory Allocator，KMA)，如图 1 所示. 内核态内存分配器用于物理内存的管理，用户态内存分配器用于进程的虚拟地址空间的管理. 内核态内存分配器算法包括资源图分配算法、2 的

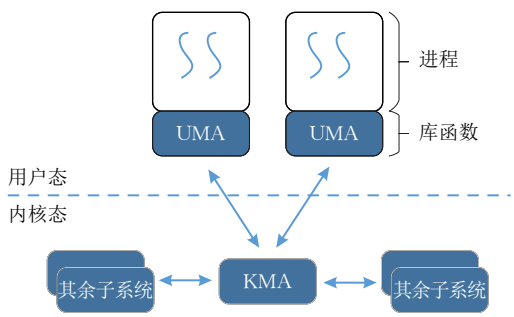

图 1 KMA 和 UMA 示意图

幂次方空闲链表、伙伴系统、区域(Zone)分配算法、Solaris 的 Slab 分配算法等；用户态内存分配器(包括本文后面会提到的大部分算法)通常是为了满足所有运行的应用程序的内存请求，与进程地址空间中的堆(heap)和运行时库(runtime library)耦合在一起，动态分配/回收进程的虚拟地址空间.

UMA 的层次是处于 KMA 之上的，也就是说，用户通过 UMA 分配进程地址空间，但当发生缺页中断需要分配物理页面时，需要由 KMA 来完成物理内存的分配. 一般而言，UMA 是应用开发人员能经常接触到的，在应用程序中会被频繁使用. 相反，KMA 工作在内核态，应用开发人员感受不到 KMA 的存在，其中的基本思想，比如伙伴系统算法自出现以来也基本没有发生变化，十分稳定. 同时 UMA 的设计直接影响了主存的使用效率和分配/释放的性能，从而影响了应用的整体性能，尤其是那些分配密集型(allocation-intensive)的应用.

根据分配的时间及空间不同，可以将内存分配分为静态内存分配(Static Memory Allocation)和动态内存分配(Dynamic Memory Allocation). 前者发生在程序编译和链接的时候，后者发生在程序调入和执行的时候；前者分配的是进程地址空间中的静态数据区，后者分配的则是堆空间. 静态内存分配存在一些缺陷，比如在为数组分配内存时，当难以预估所需的数组大小时，可能造成内存空间的浪费或不足，引发下标越界错误或更严重的后果. 而动态内存分配相对于静态分配更具灵活性，一方面不需要预先分配存储空间；另一方面分配的空间可以根据程序的需要扩大或者缩小，但在程序运行时的内存分配会给程序带来额外的性能开销.

本文中接下来提到的内存分配器，如果未特殊指明，均是指分配虚拟内存的用户态内存分配器.

2　问题与挑战

随着计算机技术的不断发展，动态内存分配器的需求及其面临的问题和挑战也在不断变化. 传统的动态内存分配器主要面临的问题和需要考虑的因素包括内存碎片、数据对齐和内存管理粒度以及局部性；当 CPU 主频提升遇到瓶颈，向着多核方向发展时，也为内存分配器带来了多线程的挑战；而当 DRAM 遭遇工艺尺寸和刷新能耗的瓶颈、新型非易失存储器快速发展时，新的存储器介质的特征又成为了动态内存分配器最新的挑战.

2.1　内存碎片

内存碎片(Fragmentation)指一个系统中存在大量不可用的空闲内存. 它会导致实际可用内存空间减少和应用程序性能下降. 内存分配算法必须能够处理好内存碎片，这是分配器需解决的核心问题之一[1].

内存碎片分为内碎片和外碎片. 内碎片发生在当分配的内存块大小比实际申请的内存要大时. 内碎片会导致空间的浪费，但是这种浪费在很多时候却能带来分配延迟的降低和分配器结构的简化. 外碎片是当空闲内存合计起来足够满足一个分配请求，但是没有一个单独的空闲块大到可以来处理这个请求时发生的. 外部碎片比内部碎片的确认要困难得多，因为它不仅取决于以前请求的模式和分配器的实现方式，还取决于将来请求的模式. 大多数分配器会有分割的操作，即将内存块分割成多个部分，分配其中的一部分，然后将剩下的作为一个更小的内存块；很多分配器也会有合并的操作，即将临近的空闲块合并，组合成一个更大的块，以满足未来的大对象请求；当然也有少数分配器，选择不处理内碎片，甚至在极端情况下每次都分配同样大小的内存块.

值得注意的是，内存碎片从某种意义上来说是无法完全消除的，内存分配器必须尽可能减少碎片问题可能会带来的影响，将之尽可能保持在可接受，不影响分配性能的范围内.

2.2　数据对齐和内存管理粒度

现代计算机访问物理内存时，都是以页面为单位进行的，也就是说操作系统管理内存的粒度是页面大小(在典型的使用虚拟内存系统的操作系统上页面通常是 4 KB). 内存数据对齐就是在放置数据时将数据放在偏移量为缓存行长度整数倍的地址上，这样就能充分利用 CPU 访问内存的方式，避免数据同时出现在不同数据块上，而减少了 CPU 访问内存的次数. 在动态内存分配时也需要考虑到数据对齐的问题. 为了将内存数据对齐，有时就需要跳过若干个字节大小的内存空间或者填充无意义的字节.

2.3　局部性

局部性原理是指 CPU 访问存储器时，无论是存取指令还是数据，所访问的存储单元都趋于聚集在一个较小的连续区域中. 内存分配也具有局部性，同样包括时间和空间局部性. 时间局部性是指动态内存分配时，相近时间内分配的内存大多会在相

近的时间内释放.空间局部性则是指在相近时间内分配的内存,其中的某个内存区域被访问之后,虚拟地址空间上相邻近的区域也会在相近的时间内被访问.

动态内存分配器需要尽可能利用分配器里面的局部性原理,处理得当将会提升用户进程访问内存的速度.

2.4 多线程

随着硬件技术的飞速发展,功耗等问题导致处理器主频不能进一步增加,处理器转而向多核发展.为了充分利用多核的潜能,多线程应用蓬勃发展,内存分配器也需要随之进化.

多线程情况下,内存分配将面临很多在单线程模式下不存在的挑战:

(1)效率及可扩展性.对于一个具备可扩展性的理想分配器而言,随着系统中处理器数量的增加,分配器的性能应该随之线性增加.速度方面,支持多线程的分配器至少应该保证当一个多线程应用在单处理器系统中运行时,多线程分配器提供的性能要好于一个优秀的不支持多线程的分配器,而不是带来程序的性能退化.

(2)伪共享(false sharing)问题.如图2所示,当多个处理器共享同一个缓存行(cache line)里的一部分,但是并不共享数据时,称两个线程发生了伪共享.分配器可能主动引入伪共享问题,例如分配器可能给多个线程分配缓存行的一部分(比如将缓存行分割成8字节的chunk),如果多个线程请求8字节对象,分配器会轮流为每个线程分配处于同一个缓存行的8字节对象给每个线程一个缓存行的8字节对象;分配器也有可能被动引入伪共享问题,比如假设一个程序在处理器间传递缓存行片段,分配器可能会在缓存行片段释放后,让每个处理器重用各自释放的缓存行片段,进而带来伪共享问题.如果有两个线程正运行在不同的处理器上,并且都在操作相同缓存行上的不同对象,那么这两个处理器必须争抢属于自己的缓存行,将带来严重的缓存行不命中和性能下降的问题.

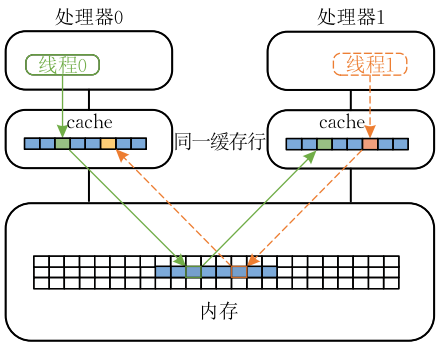

图2 伪共享示意图

(3)堆爆炸(heap blowup)问题.堆爆炸问题是分配器内存占用的一个特殊情况.一个典型的堆爆炸来源就是"生产者-消费者"模式,也就是说生产者线程不断地分配内存块并传递给消费者线程,然后由消费者线程释放.如果消费者线程释放的内存不能及时被生产者线程重新用以下一次的分配的话,就会导致分配器向操作系统申请越来越多的内存,堆爆炸问题随之发生.

此外,还有由于错误指针操作带来的多次释放(double free)、指针悬挂(dangling pointer)和内存泄露(memory leak)问题.这些问题原本就存在,只是在多线程环境下更容易出现.

(1)多次释放.第一个线程申请释放某个被多个线程共享的内存区域,然后在第二个线程中再释放一次,这将使得所申请释放的内存区域被释放两次,在大多数操作系统上,这将导致内存段错误,进程将被终止.

(2)指针悬挂.如果一个线程释放了一个对象,但没有对该对象的指针作任何修改的话,称这个指针为悬挂指针.随后倘若内存分配器将这部分已经释放的内存重新分配给另一个线程,而原来的线程又重新引用悬挂指针的话,将会导致和原来线程不相关的数据被破坏,进而导致不可预料的错误.

(3)内存泄露.线程属性可根据父进程与子线程之间的关系分为分离式和非分离式.当分离式线程退出时,内存空间会自动释放,而非分离式需要外部线程主动调用内存释放函数进行内存释放,否则就会导致内存泄露.设计错误的多线程程序可能会在线程结束时不主动释放为线程分配的内存空间,进而导致内存泄露,造成内存空间的浪费.对于长时间运行的进程,比如服务器后台进程,内存泄露将导致严重的后果.它会使得进程所占用的空间不断增长,最终导致内存耗尽.

2.5 内存介质

多核技术日趋成熟及多线程技术的广泛应用,使得处理器的性能得到了极大提升.同时DRAM由于自身工艺尺寸的限制和越来越不可忽视的刷新能耗问题遇到了发展的瓶颈.与此同时,相变存储器(Phase Change Memory,PCM)[2-3]、阻变式存储器ReRAM[4]以及自旋转移力矩存储器STT-RAM[5]等新型非易失性存储器(Non-Volatile Radom Access

Memory，NVRAM）和已经初步商业化的 Intel 3D XPoint① 技术的出现得到了学术界和工业界的广泛关注，并为计算机存储技术提供了新的解决方案. 由于 NVRAM 具备能耗低、密度高、非易失性、可按位寻址等优良特性，很有可能成为 DRAM 主存的替代品. 但当 NVRAM 作为主存时，就内存管理而言，传统的动态内存分配器是不适用的（甚至是对 NVRAM 不友好的），需要针对新型 NVRAM 的特点优化或改变其算法，尽可能在利用 NVRAM 优点的同时保留动态内存分配器高效、灵活的特征.

对于新型非易失存储介质而言，内存分配也将面临很多在传统 DRAM 介质中不存在的挑战：

（1）磨损均衡. 和DRAM不同，很多新型NVRAM都存在着写次数有限的问题，而传统的内存分配器并未考虑内存介质的寿命问题，比如数据和元数据耦合在一起，在内存块拆分/合并操作时会带来大量的小元数据写，使得局部地区被磨损得很快；又如传统内存分配器会采用类似于后进先出（Last-In-First-Out，LIFO）这样的算法来加快对刚释放内存块的重用，对于 NVRAM 而言同样可能会导致局部内存块被频繁使用，进而被磨穿. 如何将磨损均衡方法与处于软件层次的内存分配器相结合将是挑战之一.

（2）非易失特性. NVRAM 具有非易失的特性，即像磁盘和闪存一样，在掉电后不会丢失数据. 因此用户在将数据写入到 NVRAM 中时，数据就已经被保存了. 然而这些数据在程序重新运行时不能直接使用：内存分配器和应用都“不记得”上次运行时的内存分配信息了，并且虚拟地址和物理地址之间的映射关系也已经改变了. 这就意味着基于 NVRAM 的内存分配器应承担额外的任务：〈1〉为用户提供非易失性接口，分配给用户非易失内存并负责回收；〈2〉支持数据恢复，分配器需要提供一个指向之前分配内存区的指针，这样当程序重新启动后就可以访问之前的分配区域；〈3〉保证指针的正确性，在重建映射关系时，虚拟地址可能会改变，导致保存的指向之前虚拟地址空间某一位置的指针指向错误位置；〈4〉保证失败原子性（Failure-atomicity），分配器要保证所有分配的原子性，这样一个应用的崩溃不会让部分内存区域是未初始化的或者不可访问的；〈5〉垃圾回收，避免产生“无主”内存，即使产生也要注意回收，避免内存的永久浪费. 值得一提的是，对于 DRAM 这样的介质内存泄露可能只是暂时的，因为进程一旦终止其占有的所有内存空间都将还给操作系统；但对于 NVRAM 内存而言，为了便于数据恢复，进程的终止并不会引发内存回收，也就是说 NVRAM 中的内存泄露将是永久和不可逆的，所以必须要有合理的垃圾回收操作.

（3）安全性. 类似 PCM 等 NVRAM 这样有写入次数限制的存储器带来了一种新的安全威胁，即寿命安全问题. 比如恶意攻击——一般地，攻击者可以通过恶意磨穿局部区域的方式来使得 NVRAM 系统在短时间内不可用、导致系统崩溃等；保修欺骗——NVRAM 设备的购买者可以在它们的设备保修期满之前为了得到一个全新的设备而使用类似的攻击代码使 NVRAM 无效等等. 在真正应用 NVRAM 到主存前解决这些安全漏洞非常有必要. 分配器由于其特殊性会经常与应用交互，当 NVRAM 作为内存时极有可能面对一种十分简单的攻击——不断分配-写入-释放相同大小的对象，这种攻击模式就可以让分配器所映射的物理内存磨损明显加快. 首先，这样的攻击模式会频繁地使用同类内存块，使得同一个内存块被多次重用；其次，频繁的分配和合并操作也带来大量元数据写入，加快存储元数据内存块的局部磨损.

此外，计算机体系结构对内存访问模式有着直接的影响，也进一步影响到了内存分配器的工作方式，比如 DRAM 内 bank 的并行性、UMA（Uniform Memory Access）和 NUMA（Non Uniform Memory Access）架构等.

3　研究现状分析

3.1　关于动态内存分配器的分类方法

对于动态内存分配器，研究人员根据不同的出发点和视角提出了多种分类方法.

早在 1995 年关于动态分配器的综述[1] 中，作者以十年为一个阶段，按时间对动态分配器相关的工作进行了划分. 发展到近二十年，众多内存分配器虽各有聚焦但仍然保留了分配器的一些基本特征（如首部信息与对齐、边界标记、分配块的指针域和对小对象的特殊处理等等），十分相似.

内存分配器可以采用多种数据结构来组织，比如空闲分配链表、对象池、位图等等基本算法. 现有的一些分配器往往不只采用了一种算法而是混合的

① Merritt R. 3D XPoint steps into the light. http://www.eetimes.com/document.asp, 2016

算法,以 glibc malloc 为例,它对于小于 64 B 的对象采用类似对象池的算法,对于大于 512 B 的对象采用最佳适配算法的空闲分配链表,对于 64 B 到 512 B 的对象采取的是对象池和最佳适配算法的折中策略(即分配的对象既有可能来自对象池,也有可能来自采用了最佳适配算法的空闲分配链表),对于大于 128 KB 的请求则是通过操作系统的 mmap 系统调用解决.

内存分配器也可按照专用(定制)分配器和通用分配器进行分类,通用分配器可以适用于各种复杂环境但是比较复杂,需考虑多种情况,分配/释放关键路径较长;专用(定制)分配器针对特定应用环境,设计简单而且性能高效,但不具备通用性.

内存分配器还可以根据内存是否需要手动回收而分为手动分配器和自动分配器两类.诸如基于 malloc 这一类的分配器属于前者,也就是说,编写程序的工程师必须自己跟踪已分配的内存,主动释放;而诸如 java 垃圾收集器这一类的属于后者,垃圾收集器会自动跟踪已分配的内存来消除类似内存泄露、悬挂指针等问题.

近三十年文献有记载的对内存分配器的分类方法主要有三个,第一个来自 1995 年动态内存分配器的综述[1].该分类实际上并不是对内存分配器的分类,而是对内存分配器采用的不同机制/算法(为了记录哪些内存区是空闲的,哪些在使用中以及合并内存块为更大的块而采用的方法)的分类.具体如下:(1)顺序适配(Sequential Fits);(2)分离空闲链表(Segreated Free List);(3)伙伴系统(Buddy System);(4)索引适配(Index Fits);(5)位图适配(Bitmapped Fits).作者在综述中也提到这样的分类并不是最理想的[1].因为基于机制/算法的细分可能会模糊更重要的策略.

第二个来自 2000 年 Hoard[6] 的作者,他结合当时多处理器内存分配器的发展现状将内存分配器算法分为了以下五类:(1)串行单堆(Serial Single Heap),如 Windows NT/2000[7];(2)并行单堆(Concurrent Single Heap);(3)纯私有堆(Pure Private Heaps),比如 C++ 的 STL;(4)带所有权的私有堆(Private Heaps with Ownership),比如 ptmalloc① 和 LKmalloc[8];(5)带阈值的私有堆(Private Heaps with Thresholds),比如 Hoard 和 Vee[9].

最后一个则是 2002 年对定制内存分配器的分类[10].共有三类:(1)Per-class 的内存分配器;(2)基于 region 的内存分配器;(3)用于特定内存模式的内存分配器.

一方面,之前的分类方法要不就是已经过时,要不就是不够全面,只能包含部分动态内存分配器.另一方面,就像前面提到的,一些比较新的相对复杂的分配器会选择混合上述分类中的多种机制/算法以获得理想的性能,因此无法沿用之前的分类方法.综上,本文并没有在描述文献的时候就将之分类.此外,分配器领域的发展方向与特定的计算机重大事件(如 CPU 多核的出现,NVRAM 的兴起)是紧密联系在一起的,通过时间分割法能详细阐述推动分配器发展的背后历史原因.因此本文沿用上一篇相关综述的时间分割法,填补其从 1995 年到现在动态内存分配器相关工作的空白,并将在总结与回顾这一小节中利用结合出发点与具体采用策略的多层次分类方法对本文所包含的分配器优化方向进行了总结与分析.该分类方法相比较之前的分类更为全面,能更好地帮助理解动态内存分配器的优化方向.

值得注意的是,因时间和篇幅有限,并不能涵盖本时间段内所有相关的论文、期刊或软件成果.

3.2 1990 年到 2000 年

计算机历史上将 1986 年到 1994 年称为 PC(Personal Computer)的黄金时代.伴随着 PC 的盛行,DRAM 的需求也在不断提升,内存分配器也在这样的历史背景下经历了从无到有的过程.

第一个十年是内存管理和内存分配思想、概念、机制以及算法发展的十年.也有少部分学者在世纪末率先开始了对多线程环境下分配器优化的研究.

1992 年至 1994 年,Zorn 和 Grunwald 及其合作者进行了大量相关的工作,首先从空间、时间以及局部性方面进行了分配器和垃圾收集器的大量实验评估[11].这也是第一次使用来自不同程序的真实 trace 进行测试的大规模实验;继而他们发现循环首次适应算法(Next Fit)会带来最差的局部性[12].此外他们也尝试去建立基于传统内存分配器的评估模型,使得分配器可以被人工合成的 trace 评估,但得到的结论是:评估分配器的唯一可靠方法是利用真实 trace 的 trace 驱动模拟[13].

1995 年的文献[1]是被广泛引用的综述文献,Wilson 等人全面地描述了动态内存分配相关的问题,包括内存碎片、分配器所使用的策略和机制等.

① Gloger W. Wolfram gloger's malloc homepage, May 2006. http://www.malloc.de/en/

后面提到的很多分配器中的基本思想和算法都可以在本文中找到前身.遗憾的是,后续再也没有出现类似的同领域综述文章.

第一个被广泛使用的通用动态内存分配器——dlmalloc① 于 1996 年问世.dlmalloc 由 Lea 于 1987 年开始研发,一度被认为是速度最快且最节省内存的堆分配器[8].dlmalloc 根据内存块大小使用最佳适配算法分配内存,相邻的空闲内存块可以合并以减少碎片,对于小块内存的管理使用双向链表,大块内存的管理使用树结构,这些数据结构仅用来缓存空闲的内存块.dlmalloc 的思想被很多后来的分配器采用并发展,在分配器历史上具有深远的影响.不过,dlmalloc 使用一个单一的锁来保护整个分配的数据结构,没有考虑页面边界问题,更不支持多线程.

1997 年的 ptmalloc 第一代版本是基于 dlmalloc 的,由 Wolfram Gloger 开发,到 2.7 版本(2006 年发布)之后就整合进了 glibc2.3 中,之后一直成为 Linux 中的默认内存分配器,即前面提到的 glibc malloc.关于为什么要使用 ptmalloc,作者在代码的注释中也提到"这并不是有史以来最快的、最节省空间的、最轻便的、最具可扩展性的 malloc 分配器,但它是其中之一".ptmalloc 主要的改进部分就是提供了对 SMP(Symmetric Multi-Processor)环境下多线程的支持.在 dlmalloc 中只有一个主分配区,每次分配内存都必须对主分配区进行加锁,分配完成后释放锁,在 SMP 多线程环境下,对主分配区的锁的争用很激烈,严重影响了分配效率.于是 ptmalloc 增加了非主分配区,主分配区与非主分配区用环形链表进行管理.每一个分配区利用互斥锁使线程对于该分配区的访问互斥.缺陷在于申请小块内存会产生很多内存碎片,ptmalloc 在整理时也需要对分配区做加锁操作.每个加锁操作大概需要 5 到 10 个 CPU 时钟周期,在线程很多的情况下锁的等待时间就会延长,使得分配性能下降.

1998 年,一个针对多线程并发应用的多线程内存分配器——LKmalloc[8] 被提出,作者从"之前动态内存分配器的工作很大程度上忽视了长期运行的服务器应用"这一观察出发,分析了服务器应用和一次性应用对内存分配的不同需求并提出了自己的分配器.LKmalloc 使用了很多子堆,一个线程总是从同一个子堆分配内存块,但可以释放其它子堆的内存块;还提供了一个映射线程到子堆集合的映射函数.值得注意的是,LKmalloc 将大块内存永久分给处理器,并将释放的块立即返回这些区域,这在某种程度上缓解了伪共享问题.

1999 年,Vee 和 Hsu[9] 认为绝大多数程序员将应用程序从单处理器转移到多处理器的主要原因是为了获取应用的加速,而不是为了寻求更大的内存容量,所以多线程内存分配器的时间效率要比空间效率更重要.基于这一点,作者提出了一个针对固定块大小的共享内存多处理器的多线程内存分配器.算法包含两个部分:一个全局池,对全局池的并发访问通过一个锁控制;与处理器数量相同的多个局部池,每个处理器可以使用自己的局部池和全局池.在局部池中保存了两个带阈值控制的内存块堆栈以及一个用于记录局部池使用状况的计数器,这种方法有效限制了分配器私有池所占用的内存数量.

其它的相关工作还有:1996 年的 Vmalloc[14] 从严格意义上来说是一个分配器框架,内存被划分为若干区域,对于每个区域可以使用不同的策略管理,具体使用哪一个分配策略可以在短时间内被选择;1996 年 Richard[15] 仔细论述了各种垃圾收集器算法,包括引用计数(Reference Counting)、标记和清除(Mark&Sweep)、拷贝(Coping)算法等.在详细阐述这些算法的同时,列举了它们的优缺点以及研究人员所做的不同改进.

3.3　2000 年到 2010 年

自 2002 年起,CPU 芯片加工工艺逐渐逼近极限,同时遭遇主频提高后无法解决的散热问题.2004 年 Intel 在发布 3.8GHz 的产品之后只好宣布停止 4GHz 的产品计划,为了保证 CPU 的性能发展不停步,CPU 开始向多核方向发展.为了充分利用 CPU 的性能,顶层的应用程序开始利用早在 60 年代就已被提出的多线程思想,带动了动态内存分配器向支持多线程内存分配、保证多核处理器系统内存分配性能的方向发展.

这一个十年,建立在之前对分配器(dlmalloc,ptmalloc 等)的研究之上,为配合多核 CPU 和多线程应用的发展,更多针对多线程分配器的研究被提出.将无锁数据结构应用在分配器中以减少锁竞争开销的研究也初见端倪.

2000 年,来自大学实验室的 Hoard[6] 被开发出来,支持 Linux、Window、Solaris 等操作系统.基于"传统内存分配器通常成为多处理器系统中严重限

① Lea D. A Memory Allocator Called Doug Lea's Malloc or dlmalloc for Short. Available online [March 26, 2010]: http://gee.cs.oswego.edu/dl/html/malloc.html, 1996

制程序可扩展性的瓶颈”这一观察，Hoard 主要针对多处理器并行环境设计. Hoard 是第一个同时解决了多处理器共享内存场合的缓存伪共享问题[16-17]和堆爆炸问题的分配器，其维护了 per-处理器堆和全局堆，当一个 per-处理器堆的使用率低于某个设定的阈值，Hoard 就从 per-处理器堆转移一个固定大小的 chunk 到全局堆，使得空间可以被其它处理器使用. 但是，由于 Hoard 是针对多处理器共享内存的环境而设计的，在单处理器系统中应用则开销较大.

2002 年，Hoard 的作者 Berger 等人[10]在调研中发现，使用通用内存分配器（作者称之为 heaps，heap 也就是用于动态内存分配的堆空间）代替应用中的定制内存分配器会取得更好的性能，基于 region 的定制内存分配器（作者称之为 regions，region 指的是可以一次性释放所有已分配对象的集合）除外. 但是 regions 只能释放所有内存，而无法释放单独的对象，这可能要消耗更多的内存，并且导致不支持常用的应用场景，如动态数组和“生产者-消费者”模式等. 于是作者提出了 reaps，一个结合了 regions 和 heaps 的通用内存分配器. reaps 通过提供额外的接口，使得应用可以释放单独对象，并将之托管在 heaps 中以待重用，从而既提供了 regions 原本的特性，又支持了对单个对象的删除.

2004 年，Michael's 分配器[18]问世，该分配器是一个具备可扩展性且无锁的动态内存分配器. 作者发现当时的内存分配器为了实现线程安全（MT-safe，safe under multithreading），会使用不同形式的互斥锁，从而对内存分配器包括性能、可用性、健壮性和编程灵活性在内的多个方面造成负面影响. 为了利用无锁同步的优势，作者提出了一种完全无锁的内存分配器. 为了实现通用性，他们仅使用了广为各操作系统及各主流处理器支持的原子指令；为了实现无锁，他们将 malloc 和 free 拆解成几个原子步骤，并组织其数据结构，使得任一线程在其它线程处于任何状态时都可以使用内存分配器. 此外他们还沿用了 Hoard 中的上层数据结构，从而实现了多核间的并发内存分配操作，也避免了伪共享，并使得内存占用只能增长常数倍. 该内存分配器中的堆由称之为 superblocks 的大块内存构成，每个 superblock 又划分为等大小的 blocks，并根据其中 blocks 的大小划分为不同 size class. 每个 size class 维护着相当于处理器核心数的堆. 在分配和释放时通过识别请求内存的大小与线程的标识符，在相应堆中进行原子操作.

jemalloc[19]由 Jason Evans 于 2006 年开发出来，也是一个针对多处理器共享内存场景下的需求而开发的内存分配器，同样解决了多处理器系统多线程环境下缓存伪共享问题，与 Hoard 不同的是它采用轮转的方式让不同的处理器线程开辟空间，每个处理器线程的初始空间为 2 MB. jemalloc 使用了 3 个大小分类：small，large，huge，并以不同的方式实现之. 对于 small 和 large 对象，jemalloc 使用一个红黑树来跟踪任意连续的页面并在 chunk 的首部维护元数据信息；对于 huge 对象直接采用 mmap() 映射. jemalloc 有效地减少了长期运行进程比如浏览器和数据库服务器的内存占用. jemalloc 原本是为 FreeBSD 开发的，后来 Firefox 浏览器、NetBSD 和 FaceBook 的服务器端都加以应用，jemalloc 自身也在这些应用中得到了大幅改进与提升. 可惜的是 jemalloc 没有应用于 Windows. 最新版本为 2016 年 5 月的 v4.2.0.

同样在 2006 年，Li 等人[20]注意到当时的操作系统中的内存分配器完全由软件实现，而如果使用硬件内存分配器，则可以实现空闲内存的并发搜索，并可以在应用执行的同时执行内存释放，还可以在后台合并空闲内存. 但同时硬件内存分配器可能具有很高的硬件复杂度. 基于这样的观察，他们折中了硬件的高性能与软件的低复杂度，开创性地提出了一种硬件层/软件层混合设计的内存分配器. 该内存分配器的软件部分沿用了 FreeBSD 中的算法，负责建立页索引并初始化页面首部，而硬件部分用于提高软件部分的性能. 当请求大对象（大于半页）时，内存管理完全由软件实现；当请求小对象时，软件部分会定位有空闲页的位图，并向硬件发出搜索申请. 硬件部分会并行地搜索页索引（或位图），以找到空闲 chunk，并标记位图为已分配. 他们提出的硬件设计具有固定的硬件复杂度，并不会随着内存页的尺寸改变而改变.

同年，Streamflow[21]被提出，Streamflow 是一个一个高性能、低开销、线程安全的内存分配器，并且在缓存层、TLB（Translation Lookaside Buffer）层以及页面层优化了局部性. 作者发现当时的多处理器内存分配器已经具有可扩展性，传统内存分配器则重点关注了局部性，然而却从来没有同时结合可扩展性及局部性考虑的设计出现. 基于这样的观察，Streamflow 分离了本地（local）和远程（remote）操作，在最常见的本地操作中采用了无需同步的设计，并在远程操作中采用了新的非阻塞算法，从而减少

了同步开销，提高了可扩展性，并避免了堆爆炸及伪共享问题. Streamflow 也通过精心布局内存中的堆，重用空闲内存，利用 superpages，提高了缓存层、TLB 层和页面层的局部性.

同年，McRT-Malloc[22] 的作者基于自己之前在软件事务内存（Software Transcational Memory，STM）方面的工作[23]，指出事务内存可以显著减少并发编程的复杂度，进而使得应用可以高度并发，充分利用多核处理器的潜能. 作者提出了一个在事务代码块内支持内存分配和回收的非阻塞式内存分配器. McRT-Malloc 基于 Hoard 修改，据作者称是第一个将软件事务内存和基于 malloc/free 的内存分配器结合在一起的工作. 不同于之前的无锁设计，McRT-Malloc 的算法避免了传统代码路径上的原子操作，带来了效率的提升. 相似的，McRT-Malloc 也使用了线程私有堆的结构. 遗憾的是由于 McRT-Malloc 缺乏具体的实现，无法对其进行评价比较.

2007 年，tbbmalloc[24] 被提出并用于 intel 的线程构建块（Threading Building Blocks，TBB）. Tbbmalloc 基于 McRT-malloc 的思想，使用了线程私有堆（thread-private heap），并且从来不会将小对象所占的空间还给操作系统. tbbmalloc 总是无锁地从一个线程私有堆分配内存. 如果同一个线程也像分配时一样将对象返回给私有堆，那么就不需要加锁；否则对象将返回给 foreign 块并需要轻量级的同步来将该对象放入 foreign 块所有者的空闲分配链表. 与 jemalloc 和 Hoard 相同，tbbmalloc 使用的 chunks 内包含相同大小的对象，并将元数据放在每个 chunk 的首部. 遗憾的是在某些特定的场景下，tbbmalloc 可能会达到无限的内存占用[25].

同年，Google 公司公布了 TCmalloc①，TCmalloc 是一款专为高并发而优化的内存分配器，为 Google gperftools 里的组件之一. TCmalloc 拥有一个中心堆和多个每个线程私有的线程缓存（Thread-Cache，这也是名字的由来）. 处理分配请求时，对于小对象优先使用线程堆中多级链表里的空闲块分配，若链表中空间不足则需要向中心堆申请一个合适大小的空闲块，并切分成多个大小相同的小对象加入线程堆链表中以满足请求；对于大对象则直接使用中心堆，通过自旋锁保证并发控制. TCmalloc 主要运用了“中心堆-线程堆”模型，重点优化了小对象的分配和释放，使得对小对象的分配无需加锁. 由于在实际程序中，小对象的分配远远多于大对象分配，因此可以极大地提升程序性能. 最新版本为 2016 年 5 月的 gperftools-2.5.

2010 年，Gidenstam 等人[26] 提出了 NBmalloc，一个新的无锁、细粒度同步的，用以增强并行性、容错性和可扩展性的内存分配器. NBmalloc 的架构也来自于 Hoard，并引入了一种新的，被称为 flat-set 的数据结构. Flat-set 是一种容器，其操作是常规集合操作（交集∩，并集∪等）的子集，同时提供“inter-object”操作，用于将一个 flat-set 中的对象迁移到另一个中. 此外 NBmalloc 还设计了新的无锁算法，它利用了多处理器系统提供的标准硬件同步原语（即 CAS（Compare-And-Swap），或等效的 LL（Load-Link）和 SC（Store-Conditional）等），并提供 flat-set 操作的线性化实现.

其它的相关工作还有 2005 年试图改进 cache 局部性，特别是针对长时间运行作业的 Vam allocator[27]；2005 年同样借鉴 Hoard、与 Micheal’s allocator 类似的，利用多处理器系统提供的标准同步原语实现的无锁分配器 nbmalloc[28]，即 NBmalloc 的前身.

3.4　2010 年到现在

2011 年 IBM 研究人员实现了 PCM 的重大突破——多位封装技术，该技术使得一个存储单元能够存储多位数据，意味着 PCM 的高存储密度已取得实际突破. 三星公司在一年后的国际固态电路研讨会上推出了 20 nm 工艺的 8 GB PCM 芯片，并已经应用到旗下一款手机设备中[29]. 早在 1996 年，就有学者从理论上预测了一种被称为自旋转移矩（Spin Transfer Torque，STT）的纯电学的磁隧道结写入方式，2005 年索尼公司首次制备了 4 KB 的 STT-RAM 测试片，此后，STT-RAM 的研发吸引了越来越多的关注，从 2010 年一直到 2016 年，都有各大厂商比如高通、东芝、TDK、三星、Everspin 等推出了自己从 4 KB 到 64 MB 不等的芯片[30]. 同样的，忆阻器（memristor）的概念 1971 年就被提出，到 2000 年实验室用钙钛矿氧化物制作出了 RRAM，再到 2010 年，美国惠普实验室科学家在《自然》杂志上撰文表示，他们在忆阻器设计上取得重大突破可用于数据处理和存储[31]. 此后，各种 RRAM 芯片层出不穷，工艺越来越先进，容量也越来越大. 此外，2015 年，Intel 和 Micron 推出了号称是 25 年以来内存芯片市场上首次出现的突破性新技术——3D XPoint. 其包含了 cross-point 结构以及新型介质特

① TCmalloc：Thread-Caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

征.与传统 3D NAND 不同的是,3D XPoint 是通过材料的晶态/非晶态来存储数据.2017 年 3 月,Intel 推出了应用该技术的真实产品——傲腾(Optane),傲腾内存是自适应的智能系统加速器,搭建了 DRAM 和存储之间的桥梁,从系统引导到应用启动全方面优化电脑相应能力①.

可以看到,随着 DRAM 面临制造工艺和刷新能耗双方面的瓶颈,传统 DRAM 介质的发展面临困境,迫使学术界将注意力转到寻找新的可替代介质上.不同的非易失存储介质首先在材料领域获得了快速发展,然后获得存储领域的广泛关注,进而带来了新型非易失存储的研究热潮.

最近几年,动态内存分配器的研究开始逐渐跟非易失存储结合,学者开始研究非易失内存的引入对传统分配器带来的挑战并从不同的角度给出了自己的解决方案.

2011 年,SFMalloc[32] 实现了一个针对多线程应用的基本无同步操作的动态内存分配器.SFMalloc 利用了很多之前工作的思想,比如分离块分配机制、来自 streamflow 的远程释放思想、来自 michael 的风险指针来实现无锁对象的安全内存回收等来解决锁争用、伪共享、堆爆炸等问题.类似的,SFMalloc 也使用了私有堆机制,但是在无锁和无同步方面做了更多优化工作.作者提到了很多无锁分配器里使用的原子同步指令或者细粒度的锁往往开销很大,应该尽量减少同步指令的出现,因此 SFMalloc 使用了实现更为简单的无锁栈.另外,SFMalloc 通过在内存分配器中添加多重缓存,有效地减少了内存管理带来的延迟.最后,此前的分配器聚焦于小内存块的高效内存分配,SFMalloc 则考虑了应用间歇性分配大内存块的特殊情况.

同年,学者 Herter 等人从实时系统对动态内存分配器的特殊需求出发,提出了 CAMA[33]——一个可预测的、cache 可感知的内存分配器.他们注意到通用内存分配算法往往致力于提供良好的平均响应时间,而不能保证最坏情况下的响应时间,这使得它们不适用于要求每个操作的执行时间都有着严格上界的实时程序.这些内存分配器在两个方面引入不可预测性:(1)无法保证分配的内存块映射到的 cache set;(2)通过遍历分配器中保存的空闲内存块分配内存,而遍历的时间复杂度是 $O(n)$.因此,与其它内存分配器提供的接口相比,CAMA 需要额外提供一个参数,用于指定内存所分配到的 cache set. CAMA 采用互相隔离的空闲链表管理空闲内存块.此外他们还采用 cache 可感知的分割、合并空闲内存的技术来减少外碎片;采用多层隔离链表(muti-layered segregated-list),类似于 TLFS[34] 中的方法减少内碎片.值得注意的是,在文献中并没有提到对多线程的支持,我们也未找到其开源实现.

2011 年中国科学院的学者从数据局部性角度出发,提出了一种对用户透明的、轻量级动态数据分配的方案[35],在用户态直接优化二进制代码.其在运行时识别对象之间的亲缘性(程序在访问数据时,一些对象容易被邻近访问到,亲缘性(affinity)正是用来识别数据的这种性质),把具有亲缘性的对象从同一内存池里分配.依据亲缘性把对象分到不同的对象组里,可以更好地适应它们的访存特点,提高数据局部性,增加它们所在缓存行的重用.另外,作者实现了动态池分配原型系统 DigitalBridge-dopt,通过提升数据的局部性得到了性能的提升.

2012 年 SSMalloc[36] 指出影响内存分配器的三个主要因素分别是:分配延迟、访问局部性和可扩展性.他们的工作着重考虑了性能可扩展性,利用到了一个有趣的观察——同时执行虚拟内存管理操作(比如 mmap 和 munmap)的多个线程会在内核里产生竞争.SSMalloc 通过最短化关键路径、严格限制虚拟内存管理系统调用、采用无锁结构和无等待算法有效降低了内存分配器的访问延迟,提升了访问局部性和可扩展性.所谓的无等待算法指的是,在无锁同步中严格限制循环的使用,并使得算法中的其余部分都能在有限的步骤中完成.值得注意的是,SSMalloc 的设计也采用了私有堆的概念并将全局堆和私有堆交换的基本对象(chunk)保持大小一致.据作者称,此举可以使得私有堆中不同 size class 的内存对象之间可以更容易地相互转换.SSMalloc 是一个开源的项目.

2013 年的 NVMalloc[37] 可能是第一篇关于非易失内存分配器的文献.文中提出了一个针对 NVRAM 的磨损可感知的分配器,其限制内存块的分配频率不高于 1/T.首先,它给每个被释放内存块加上一个 T 的时间戳限制然后将之添加到一个叫“不分配列表(don't allocate list)”的 FIFO(First-In-First-Out)空闲块队列.每一次内存分配和释放时,NVMalloc 会检查该 FIFO 的队列头,如果它在队列中的驻留时间达到 T,就将其移除队列,并标记为可

① Intel Optane Memory — Revolutionary Memory http://www.intel.cn/content/www/cn/zh/architecture-and-technology/optane-memory.html

重新分配.此外,作者还出于安全性和健壮性的考虑提出了防止错误写的虚拟内存保护机制,比如在每个内存块的头部新添加一个校验和来检测对持久化数据的错误写入;最后,作者出于性能考虑提出了自己的缓存一致性方案.

2014 年 PALLOC[38]的工作聚焦于多核计算机系统中内存子系统的 bank 共享问题及性能隔离,作者提出了一个 DRAM bank 可感知的内存分配器,通过利用现代操作系统基于页的虚拟内存系统,可以把内存块分配到指定的 DRAM banks 中.这是一个基于伙伴算法修改的内核态分配器,只处理单个物理页框的分配,多个物理页框的分配仍然由原始的伙伴系统来处理.这样做是出于如下的考虑:用户级的内存分配最终都将由页面中断处理程序以页框大小(4 KB)来处理.PALLOC 可以为用户级应用正确地控制 bank 的分配.最常见的内核内部分配请求(获得一个页框)也由 PALLOC 处理.通过动态分割 banks 来避免多核之间的 bank 共享,从而在不需要任何特殊的硬件支持的前提下提升 COTS (Commercial Off-The-Shelf)多核平台的隔离性.此外,PALLOC 没有处理大请求,而是交由操作系统的 mmap 系统调用来完成.

2015 年,SuperMalloc[25]——一种最初为 64 位 X86 硬件事务内存(Hardware Transactional Memory, HTM)设计的内存分配器由 MIT 的研究人员提出.作者观察到,虽然物理内存是宝贵的,但 64 位机器上的虚拟地址空间却是巨大的,达到了 256 TB 之多.与其它面向 chunk 的分配器不同,SuperMalloc 并不将 chunk 分割成更小的单位,而是直接以 2 MB 为单位对齐,正好与 64 位 X86 系统上的大页(huge page)相同.其它分配器"分割 chunk"的方法有效节省了虚拟地址空间,因为需要更少的 chunk 来满足分配.由于操作系统请求调页的特性(即虚拟地址对应的页面在实际写入时才会真正分配物理内存),在 SuperMalloc 中,一个 block 大于分配请求大小的部分是不占用物理内存的.作者反复在文中提及 64 位软件设计的原则之一——可以适当浪费虚拟地址空间.其它的设计方法诸如对象按大小分类,数据结构等与 jemalloc 类似,只是它采用的是数组而不是树结构来实现查找表,尽管查找表需要占用 512 MB 的虚拟地址空间,但同样由于操作系统请求调页的特性,实际并没有占用多少物理内存.另外,作者从速度、内存占用、代码复杂度三个方面评估了多个分配器,代码也在 github 上开源.

同年,scalloc[39]的作者也注意到了虚拟内存具有请求调页的特性,而且 64 位地址空间十分庞大(256 TB).基于这样的观察,他们设计了一款基于虚拟 span 技术的多线程内存分配器,scalloc.他们通过 mmap 系统调用一次性申请 32 TB 的虚拟内存,称之为 arena,并将之划分为大小为 2 MB 且关于 2 MB 对齐的内存块,称之为虚拟 span.scalloc 中使用虚拟 span 统一地处理 1 MB 及以下的内存申请.在实现中,他们沿用了"前后端"技术作为整体框架,基于线程本地分配缓冲器(Thread-Local Allocation Buffers,TLABs)[6]的前端用于快速分配,基于最新的全局并发数据结构——Treiber 栈[40]的后端用于内存重用.此外,scalloc 中在内存释放后立即处理空闲内存,而非在等待一段时间后统一处理,从而使得内存回收可在常数时间内完成,但可能存在释放效率不高的问题.

同年,WAlloc[41]聚焦于如何设计一个损耗可感知的分配器来改进 NVRAM 的寿命问题.其设计思想有三个:(1)首先,元数据管理和数据管理去耦,并且将元数据分类为易失和非易失的,分离易失元数据管理和数据管理,也就是将空闲内存空间的易失元数据保存在 DRAM 中,崩溃后由 WAlloc 重建;(2)提出了一种新的最少分配优先(Less Allocated First Out,LAFO)策略,选择一个最少被写入的空闲内存块来分配,实现了分配器内空间的尽可能均匀分配;(3)管理一个由 mmap 创建的堆外内存,支持更加弹性的分配方式.值得注意的是,分配器建立在 DRAM 于 NVRM 混合的物理内存架构之上,而且 DRAM 是对用户程序透明的,只被用来存储 NVRAM 内存管理的易失元数据.其代码尚未开源.

2015 年 SAP 公司提出了 nvm_malloc[42],一个通用的 NVRAM 内存分配器,可作为持久化应用的基本构件.其被实现成了一个用户库,也建立在一个 DRAM 与 NVRAM 异构混合的架构下,NVRAM 地址通过 PMFS[43]文件系统暴露出来.nvm_malloc 建立了一套新的 API 与传统的 glibc malloc 并行,在无需修改硬件和操作系统的情况下实现对 NVRAM 的分配.nvm_malloc 将确保为 NVRAM 保留连续的虚拟地址.然而,它的起始地址是无法确定的.如果其中的内容在重新启动后恢复并且该起始地址改变,则存储在 NVRAM 上的任何指针变量都将失效.为了解决这个问题,nvm_malloc 要求 NVRAM 中存储的任何指针变量都必须使用相对寻址方案.

2016年,来自上海交通大学的学者提出了Wamalloc[44],一个高效的磨损可感知的新型非易失内存分配器.文中提出:非易失内存分配器在考虑寿命及磨损均衡问题的同时还需要考虑分配器的性能.具体采用了3个技术:(1)一个新颖的更为精确的混合(细粒度和粗粒度)磨损均衡策略;(2)使用线程缓存(thread-cache,与TCmalloc类似)结构和锁优化以获得更好性能;(3)数据与元数据解耦,将元数据放入DRAM减少对NVRAM的写入.值得一提的是,其测试部分仅对比了NVMalloc,而且没有使用任何已知的标准测试集和工作负载.此外,代码并未开源.

最新的工作还有2016年Bhandari等学者提出的Makalu[45]——一个更多考虑NVRAM崩溃及恢复一致性的持久化库.其主要目标是开发一个能与其它NVM持久化库集成在一起并且在分配/释放时无内存泄漏的分配器,并探究最小化故障一致性开销的设计方案.Makalu采用了两种方法来管理持久化内存.在线(online)阶段中,Makalu支持传统malloc()/free()结构的插入式替换;在发生崩溃后(offline),其采用一个元数据恢复阶段和紧随其后的并行mark-and-sweep垃圾回收操作来维护堆结构的一致性并回收泄露的持久化内存.此外,Makalu还将持久化元数据区分为核心(core)元数据和辅助(auxiliary)元数据.对于核心元数据,保证每次更新的一致性;而对于辅助元数据,只在程序执行完之后保证.这种方法可以减少维护持久化元数据一致性的开销.最后,由于垃圾回收算法的引入,Makalu有效地避免了持久化内存的泄露问题.值得一提的是,为了提供多核的可扩展性要求,其也维护了易失的线程局部空闲列表以支持线程内部的分配/释放.然而,Makalu选择性忽略了分配器的性能和磨损方面的考虑,我们也未找到开源实现.

其它的相关工作还有无锁的内存分配器nalloc①;Lockless Inc.推出的产品级无锁分配器llocall②.

3.5 总结与回顾

对相关研究进行的总结如表1所示.在设计表格时,我们首先根据内存分配器优化策略的出发点,将其归为三大类:堆管理、底层感知以及其它.堆管理,指内存分配器对堆空间内数据结构的管理;底层感知,指内存分配器对"底层"(操作系统及硬件,对分配器而言)部分特征的感知、利用或应对;其它,指基于这些出发点之外的设计,我们把它们笼统地归为一类.接着,在各大类之下,根据对所调研的文献的总结,各内存分配器所采用的具体策略(机制或算法)之间也具有一些规律.此外,为了展现内存分配器发展的趋势而特意区分了各文献出现的年代.下面将对各大类及各策略进行逐一地阐述,然后对内存分配器领域的发展情况进行总结与分析.

表1 文献中内存分配器优化方向总结

出发点	具体策略	1990～1999年	2000～2009年	2010年～至今
堆管理	区语义(Region Semantics)		reaps[10]	
	多堆(Multi-heaps)	LKmalloc[8],[9] ptmalloc	Hoard[6],Michael's allocator[18], jemalloc[19],Streamflow[21], tbbmalloc[24],TCmalloc,NBmalloc[26]	SFMalloc[32],SSMalloc[36], scalloc[39],SuperMalloc[25], nvm_malloc[42],Wamalloc[44]
	远程堆(Remote Heap)	[9]	Hoard[6],Streamflow[21],TCmalloc, NBmalloc[26]	SFMalloc[32],SSMalloc[36], scalloc[39],SuperMalloc[25], Wamalloc[44]
	远程对象删除 (remote object deallocations)		Streamflow[21],tbbmalloc[24]	SFMalloc[32],SSMalloc[36],scalloc[39]
	无锁算法/数据结构 (Lock-free Algorithm/DataStructure)		Michael's allocator[18], Streamflow[21],NBmalloc[26]	SFMalloc[32],SSMalloc[36],scalloc[39]
底层感知	感知bank			PALLOC[38]
	感知cache			CAMA[33]
	感知mmap/munmap			SSMalloc[36]
	感知请求调页(demand paging)			scalloc[39],SuperMalloc[25]
	感知NVRAM耐久性			NVMalloc[37],WAlloc[41],Wamalloc[44]
	感知NVRAM非易失性			nvm_malloc[42],Makalu[45]
其它	软件事务内存		McRT-Malloc[22]	
	硬件事务内存			SuperMalloc[25]
	软/硬件混合		[20]	

(1)堆管理.堆管理中采用的策略集中于五点:区语义、多堆、远程堆、远程对象删除以及无锁算法/数据结构.①区语义,是一种提供区(regions)的语

① nalloc: A Lock-Free Memory Allocator. http://www.andrew.cmu.edu/user/apodolsk/418/finalreport.html

② Lockless Inc. llalloc: Lockless memory allocator. http://locklessinc.com/

义[10]的堆管理策略，需要提供额外的删除整个区的接口以支持区的语义；② 多堆，与 dlmalloc 中的串行单堆[6]的做法不同，很多内存分配器会维护多个堆，从而减少了线程同步的开销，这是一种很普遍的做法，出现在 ptmalloc 及其之后的所有多线程内存分配器之中；③ 远程(remote)堆，其共同的特征是除了拥有线程私有的本地堆，还有一个所有线程都可以访问的全局远程堆，远程堆作为本地堆的补充，为内存分配器提供了更高的内存复用能力；④ 远程对象删除，首见于 Streamflow，为线程提供了释放不是由该线程分配的内存的能力，从而有效地避免了堆爆炸的问题；⑤ 无锁算法/数据结构，通过设计无锁算法，如将 malloc 和 free 拆解为若干原子操作[18]，或引入无锁数据结构，如 flat-set[26]、Treiber 栈[39]，减少甚至消除内存分配中锁的使用，从而减少线程之间的竞争.

(2) 底层感知. ① 感知 bank，指对 DRAM bank 的感知. DRAM 内包含多个可以被并行访问的 banks[38]. 通过尽可能将内存分布在各 bank 中来提供可扩展的性能；② 感知 cache，指对 cache 的感知. 传统分配器无法保证被分配的内存块映射到哪一个 cache set[33]，从而导致响应时间的不可预测性. 通过添加对 cache 的可感知提高了实时应用的可预测性；③ 感知 mmap/munmap，指对同时执行虚拟内存管理操作(比如 mmap 和 munmap)的多个线程会在内核里产生竞争的感知[35]. 通过细致地规划这些操作，减少因此而产生的性能开销；④ 感知请求调页(demand paging)，指对虚拟内存请求调页特性的感知. 由于 64 位地址空间是很大的(256 TB)，因此可以通过大量使用虚拟内存降低内存分配器的复杂度；⑤ 感知 NVRAM 耐久性，指由于 NVRAM 存储介质寿命有限，因此在设计时尽量考虑保证磨损均衡(Wear-Leveling)；⑥ 感知 NVRAM 非易失性，指由于 NVRAM 存储介质具有非易失的特点，为了利用该特性而提出了持久化方案.

(3) 其它. 软件事务内存、硬件事务内存，它们属于事务内存在不同层次的实现；软/硬件混合，利用了硬件的高性能和软件的低复杂度(灵活性)而设计的混合内存分配器.

不同技术在不同年代的发展趋势图如图 3 所示. 在 1995 年之后，出现了多堆策略和远程堆策略；在 2000 年后，出现了远程释放策略. 这三个策略在出现之后被广泛地使用在了后续的各内存分配器之中，几乎成为了多线程内存分配器的“标配”. 在我们所调研的 10 之后出现的 7 个多线程内存分配器(包括适用于 NVRAM 的新型内存分配器)中，有 6 个应用了多堆策略，有 5 个应用了远程堆策略，有 3 个应用了远程释放的策略. 此外，我们还发现所有的底层感知的内存分配器都出现在 2010 年之后，共计九个. 其中两个利用了操作系统请求调页的特性，通过大量使用虚拟内存加速性能；四个针对 NVRAM，提出了进行磨损均衡或提供持久化接口的 NVRAM 内存分配器. 值得注意的是，针对 NVRAM 的内存分配器，要么不具有可扩展性[37]，要么有，但是性能不佳[42]或没有和常见多线程内存分配器比较[44]. 此外，在非易失内存分配器领域尚无成熟的解决方案.

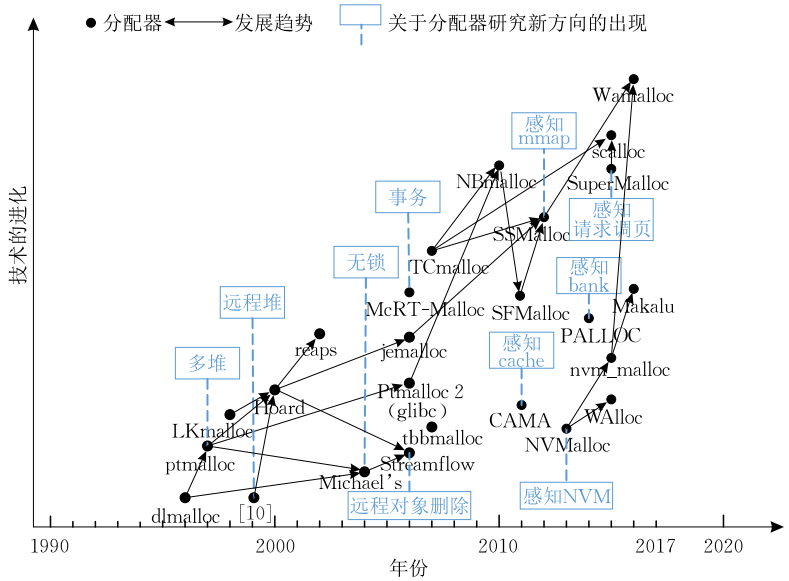

图 3　分配器所采用的技术发展趋势图

我们相信在未来,多堆策略、远程堆策略以及远程释放策略会继续应用在之后的多线程内存分配器中,也会有更多的感知别的底层细节的内存分配器出现.

此外,我们在这里提出一套新的全面、多维度的动态内存分配器评价指标体系,希望能尽可能涵盖到现有和未来动态内存分配器设计中需考虑的各个方面.并且在最后给出了使用该套标准对主流内存分配器(ptmalloc、TCmalloc 等)的评价结果以作参考(数据来自各自的文献、主页、代码及 github),如表 2 所示.

表 2 对主流分配器的评价结果(相对)

评价指标	glibc malloc	TCmalloc	jemalloc	Makalu
兼容性(Compatibility)	高	高	高	低
可移植性(Portability)	中	高	高	高
内存占用(Footprint)	低	不归还内存	低	中
分配延迟(Allocation delay)	中	低	低	低
可调性(Tunability)	有	有	有	无
局部性(Locality)	高	低	低	高
可维护性(Maintenance)	中	低	低	中
可扩展性(Scalability)	低	高	高	中
线程安全(MT-safe)	有	有	有	有
可预测性(Predictable)	低	高	高	低
非易失内存感知(NVRAM-aware)	无	无	无	有

(1)兼容性(Compatibility)

由于内存分配器主要负责堆空间的分配/释放操作,为程序提供动态内存管理的功能并参与程序的编译和运行,设计动态内存分配器时必须考虑如何使得内存分配器与其它应用程序相兼容,也就是尽量保持跟已有的接口(比如 malloc、free)相同,尤其需要遵守 POSIX 标准.

(2)可移植性(Portability)

分配器的设计和实现需要充分考虑可移植性问题.分配器不应该仅只能运行在某一种特定的操作系统之上或者仅支持某一种特定的编译器,而是至少需要考虑支持目前主流的 Windows、Unix/Linux 操作系统,支持各种标准的编译器.这一点 glibc malloc 做得很好,对不同机器字长的硬件架构都有支持.很多工作也将分配器实现成用户库,正是出于分配器可移植性的考虑.

(3)内存占用(Footprint)

内存分配器会占用物理内存,但不应该浪费宝贵的物理内存.相对于应用所需要的内存,它应该从操作系统请求尽可能少的物理内存并保持在相对稳定的水平.理论上而言,为了能最大程度地节约内存空间,程序开发人员需要内存时可以直接通过系统调用而不经过内存分配器这一层面,但这样会严重影响应用性能,尤其对于内存分配/释放比较频繁的场景来说,因为直接使用系统调用开销很大.另外对于某些特定的应用场景(比如生产者-消费者模式),分配器的内存占用不应该出现无边界堆爆炸的情况,至少应该将分配器所占用的物理内存限制在某一个阈值之内,避免由于分配器堆爆炸而导致 OOM(Out Of Memory)的情况.

(4)分配延迟(Allocation delay)

内存的分配/释放操作要尽可能快,尤其在内存频繁分配和释放的环境下,一个内存分配器的分配延迟指标非常关键,否则可能会成为整个计算机系统的性能瓶颈.很多情况下内存占用和分配延迟是一对矛盾的关系,内存占用少的分配器通常设计得比较复杂,延迟相对较高;延迟低的分配器通常设计简单,但内存占用和碎片问题就会严重一些.如何在两者之间权衡是分配器设计中需仔细考虑的问题.

(5)可调性(Tunability)

对于一个通用的分配器而言,需要满足各种不同的应用场景,但是往往不同的应用场景之间对分配器的需求差别比较大.分配器可以让用户控制部分优化调节选项,通过实施多种策略(比如宏)做到人性化调节.用户可以通过不同的宏开关控制动态分配器的行为,以获得自己最满意的效果.比如单线程环境下,不需要复杂的多线程保证和锁设计,用户可以通过一个宏开关将多线程优化关闭,保证单线程的最优性能;或者用户的某个应用场景需要频繁地使用到超过分配器 mmap 阈值的内存块,可以让用户通过宏定义设置新的阈值,从而不会因频繁使用 mmap 调用而带来性能退化.可调性将成为未来智能化的通用内存分配器的必备特征.

(6)局部性(Locality)

当前的操作系统基本都支持多任务处理,尽最大可能地保证分配时的局部性可以在一定程度上减

少页面换入换出和缓存不命中，从而提升程序性能.理论上来说，内存分配器并不能保证分配给应用程序的物理内存是不是临近的，因为真正分配物理内存的是操作系统内核.但是，可以通过一些措施改善这种情况，比如可以考虑页表边界问题和进程空间对齐问题，尽可能让程序的内存访问局部化.

(7) 可维护性(Maintenance)

软件的可维护性是指理解、改正、改动、改进软件的难易程度.通常影响可维护性的因素有可理解性、可测试性和可修改性.内存分配器在层次上属于系统软件，跟库耦合在一起之后更是如此.分配器一般不会频繁升级和更新，但是尽可能地减少维护成本也是需要考虑的一个指标.很多应用软件的维护成本很高，一方面跟不断增长的用户需求和不断更新的功能需求有关，另一方面可能跟最初的设计也有关系.内存分配器的设计要充分考虑到未来可能扩充的部分功能，尽可能减少维护成本.

(8) 可扩展性(Scalability)

分配器的性能和可扩展性往往成对出现，如果分配器对于所有的应用很慢，那就是性能差；但是如果对一个应用很快，多个应用就变得很慢，这就是可扩展性差.随着系统中处理器数量的增长以及多线程应用的普及，内存分配器应该尽力保证可扩展性，使得对多线程应用的支持尽可能好.

(9) 线程安全(MT-safe)

内存分配器必须保证能够在多个线程并发执行期间也能正确地运行.一般地，它需要满足任何时间任意多个线程同时访问同一份共享数据的需求.特别情况，比如线程中止、崩溃错误、线程调度等都不会影响内存分配器在接下来的时间里的正确运行.

(10) 可预测性(Predictable)

内存分配器的可预测性指内存分配器的分配/释放操作的执行时间要有严格的上下界.当内存分配器应用在要求每个操作都有确定上下界执行时间的实时应用中，如果内存分配器不具有可预测性，将会导致实时应用中使用的动态内存分配的操作无法确定上下界，从而导致实时应用性能的不可预测性.内存分配器可能从两方面引入不可预测性：首先内存分配器无法保证分配的内存块映射到的 cache set；其次内存分配器通过遍历空闲分配链表来选择要分配的内存块，但遍历的时间并不固定.内存分配器应从这两方面着手保证分配器的可预测性.

(11) 非易失内存感知(NVRAM-aware)

非易失内存与传统易失的内存介质(DRAM)有很大差别.具体体现在其非易失性和有限的寿命上.随着 NVRAM 进入操作系统，分配器需要能够感知到非易失内存，并从非易失内存介质中分配.此时分配器需要保证一致性，即以保险的方式维护持久化数据，使得重启或故障后继续正常分配.此外，分配器还要避免由故障导致的持久化内存泄露.比如假设非易失内存被分配了，但在应用句柄被指派给该内存之前发生故障，那么内存就被泄露了，因为其在程序重启时是无法访问的.最后，分配器应采取有效的方式缓解非易失内存的磨损问题.

上述的评价指标中，内存占用(以前也被称为“内存碎片”)、局部性、分配延迟来源于传统的内存分配器如 dlmalloc；可扩展性和线程安全来源于针对多核环境优化的多线程分配器如 jemalloc 和 ptmalloc；可维护性来源于代码行数最少的 SuperMalloc；可预测性来源于从实时应用的角度考虑的 CAMA；可调性、兼容性、可移植性来自软件工程对通用型用户库的软件要求；非易失内存感知是由我们新提出的，整合了来自于针对非易失内存的分配器如 nvm_malloc 和 Makalu 等中提到的多个细粒度指标，比如恢复时间、一致性开销、磨损考虑等.我们总结整理了一个最为全面的多维度动态内存分配器评价指标体系.相比较之前文献中使用的个别评价指标，能更全面、完善地评价一个分配器.

4　未来研究方向

随着大数据时代的到来以及新型非易失存储器件、机器学习与人工智能等技术的不断发展，应用场景对动态内存分配器的要求将会越来越高并出现新的特征.未来，动态内存分配器的研究将可能从以下几个方面展开：

(1) 工作负载和标准测试集

我们对本文中提及的分配器文献中涉及到的主要测试集和工具进行了整理，如表 3 所示.

除了表中列出来的这些，还有来自 reaps[10] 的 Web 服务器 Apache、来自 Hoard[6] 的 BEMengine、来自 Streamflow[21] 的 Knary 和 MPCDM、来自 Mcrt-Malloc[22] 的 Machias、来自 jemalloc[19] 的 cca 和 smlng、来自 CAMA[33] 的 MiBench、来自文献[20]的 treeadd，voronoi，bisort，perimeter 和 health、来自 SuperMalloc[25] 的 SuperServer 和 Vyukov 等.值得一提的是 SFMalloc[32] 的 LargeTest 是据我们所知的第一个针对大块(64 KB)对象而不是通常内存分配器

表 3 动态内存分配器所采用的测试集及工具

名称	来源	描述	使用了的文献	备注
SPECint2000	①	197.parser 语法分析器等	reaps[10],Vam[27],[20],Streamflow[21],SSMalloc[36]	定制
lcc	[46]	可重定目标 C 编译器	reaps[10]	定制
mudle	[47]	MUD 编译器/解释器	reaps[10]	定制
boxed-sim	[48]	balls-in-box 模拟器	reaps[10],[20]	定制
C-Breeze	[49]	C 到 C 的优化编译器	reaps[10]	定制
expresso	②	PLA(Programmable Logic Array)优化器	[53],Hoard[6],reaps[10],[20],[21],SSMalloc[36]	通用 单线程
lindsay	[12]	超立方体模拟器	reaps[10]	通用
Ghostcript	[50],[51]	PostScript 解释器	[53],Hoard[6],jemalloc[19],SuperMalloc[25]	单线程
P2C	[50],[51]	Pascal 到 C 的翻译器	[53],Hoard[6]	单线程
LRUsim	[51]	局部性分析器	[53],Hoard[6]	单线程
Recycle	Stream-flow[21]	合成测试集,多个线程同时分配/释放对象	Streamflow[21],SSMalloc[36]	定制
Larson	LKmalloc[8]	模拟了服务器:每个线程分配和释放对象并转移一些对象到其它线程释放	LKmalloc[8],Hoard[6],Michael's allocator[18],nbmalloc[28],Streamflow[21],NBmalloc[26],SFMalloc[32],SSMalloc[36],SuperMalloc[25],scalloc[39],Makalu[45]	多线程
shbench	③	每个线程分配和随机释放随机大小的对象	Hoard[6],jemalloc[19],SFMalloc[32],SSMalloc[36],scalloc[39]	多线程
Barnes-Hut	[52]	n-body 粒子求解器	Hoard[6],Streamflow[21],Makalu[45]	多线程
Active-false&passive-false	Hoard[6]	测试积极及消极的伪共享避免	Hoard[6],Michael's allocator[18],nbmalloc[28],NBmalloc[26],SFMalloc[32],scalloc[39]	多线程
threadtest	Hoard[6]	p 个线程重复分配并释放 100 000/p 个 64 B 的对象	Hoar[6],Michael's allocator[18],SFMalloc[32],scalloc[39],Makalu[45]	多线程
Consume	Hoard[6]	人工合成的生产者-消费者应用	Michael's allocator[18],Streamflow[21],SFMalloc[32],Makalu[45]	多线程
malloc-test/Linux-Scalability	[53]	生产者-消费者线程各 K 个	Michael's allocator[18],jemalloc[19],tbbmalloc[24],SFMalloc[32],SuperMalloc[25]	多线程
cfrac	Hoard[6]	求大数的因子	jemalloc[19],[20]	多线程
Pin	[54]	二进制插桩工具,可以记录对内存的写	NVMalloc[37],Walloc[41]	工具
SPEC CPU2006	[55]	标准测试集,从 400.perlbench 到 483.xalancbmk	PALLOC[38]	商业
ACDC	[56]	负载生成工具,可以生成多种工作负载特征	scalloc[39]	生成工具
PMEP	PMFS[43]	来自 PMFS 的模拟 NVRAM 的平台	nvm_malloc[42]	平台

注:① Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org
② Standard Performance Evaluation Corporation. SPEC95. http://www.spec.org
③ MicroQuill, Inc. http://www.microquill.com

聚焦的小对象的测试集;使用 SPEC CPU2006 测试集的 PALLOC[38] 实际上并不属于本文讨论的动态内存分配器,而是内核态基于伙伴算法的 KMA.

可以看到,三十年间动态内存分配器的测试集和工具可谓花样繁多,文献作者自己写测试程序的也不在少数,被使用较多的包括 Larson、malloc-test、shbench 等,但也不具一般性;其次,针对新型非易失器件的动态内存分配器研究工作普遍采用 Pin 这样的二进制插桩工具以及自己编写的统一或随机测试程序,而没有使用之前已经存在的 benchmark,这是由于他们没有过多考虑多线程的问题.

伴随着分配器技术的不断发展,我们认为未来将会出现从各方面综合评价动态内存分配器的标准测试集,它可能是一套全新的评价动态内存分配器各项指标的工具和测试套件;也有可能是由各针对不同特性及问题的小项测试程序组合起来的测试集合.毋庸置疑的是,为了让分配器之间的比较更具参考性和一般性,开发这样的测试集将是非常有价值的工作.事实上,2017 年 4 月,威斯康辛大学和惠普实验室的学者就提出了针对持久化内存标准测试套件——WIHSPER[57].

(2)多种内存架构下分配器的设计

首先,随着 NVRAM 的不断发展,学术界内越来越多的学者们倾向于将传统的 DRAM 与新型 NVRAM 混合使用,这是由于当前 NVRAM 的读写性能仍与 DRAM 有较大差距,以及需要 DRAM

没有寿命问题的特性来辅助实现元数据管理，针对这样的混合内存架构，如何设计动态分配器将是需要考虑的重要问题. 当然我们也看到有文献提出保留两套分配接口，但是仍然需要不少操作系统的支持与假设，还不具备实用性.

其次，倘若未来 NVRAM 器件的性能已经全面大幅超过 DRAM，那么将会出现 NVRAM 完全取代 DRAM 成为新一代的主存介质的情况. 届时，仅仅适用于 DRAM 的传统分配器将会面临一场变革，需要越来越考虑到针对 NVRAM 的优化. 针对 NVRAM 主存系统的组织结构方法研究、访问方法研究、数据可靠性研究等将会不可避免影响基于 NVRAM 的动态内存分配器设计.

其外，SCM(Storage-Class Memory)是一种采用非易失存储介质实现的新型存储技术，支持字节访问，性能与 DRAM 相当，存储密度高，容量大，能耗低，还具有非易失性. 它在计算机中可能会统一内外存，给传统的“寄存器-缓存-内存-外存”存储系统层次带来挑战. 届时内存分配的概念可能会被隐藏起来，取而代之的是管理 SCM 的复杂文件系统；或者恰恰相反，文件系统的复杂特性可能会被隐藏起来，只剩下内存分配的概念.

(3) 嵌入式内存分配器

随着物联网，智能家居，智能穿戴设备等技术的不断发展，传统嵌入式设备上内存资源宝贵，大多采用静态分配以满足实时性和可靠性的时代可能不复存在. 嵌入式系统的内存将会越来越大，需要使用内存的应用也将会越来越复杂，现有动态内存分配的思想和概念将被越来越多地引入嵌入式内存系统中. 当然，如何在利用动态内存分配器灵活性的同时保证嵌入式设备的快速性、可靠性、低能耗、低成本及低碎片将是该领域的研究方向之一.

(4) 内存分配器安全性考虑

一方面是 NVRAM 介质本身所具备的非易失性而导致的安全性问题：系统掉电不丢失，通过恶意修改数据所导致的错误可能是永久的，系统也可能面临被入侵、篡改或者数据被盗窃的危机. 这是在构建 NVRAM 内存子系统时需考虑的问题. 分配器层面应该尽力避免指针误操作或者越界访问带来的永久数据丢失. 另一方面，由于寿命有限而带来的恶意磨损安全问题也不得不考虑. 比如前面提到的恶意攻击和磨损欺骗. 分配器如果能够检测到恶意攻击不正常的分配模式并加以避免，将可以从软件层面最大化保证内存的安全性.

(5) 机器学习和人工智能的渗透

在 MWC 2017 世界移动通信大会上，华为和谷歌展示了他们深度优化的手机系统，该系统采用了基于机器学习算法的 Ulatra Memory 内存管理技术，可以提升应用启动速度，更高效地释放手机内存资源. 同样的，在与 Intel 的傲腾内存产品配套的软件中，其智能软件也可以自动学习用户的计算行为，从而加快常用任务的处理并自定义用户的计算机体验. 虽然其中的细节暂时不得而知，但我们有理由相信，随着机器学习和人工智能渗透入更多领域，未来的内存分配器将会越来越“智能”. 如何将机器学习、人工智能与内存管理技术相结合可能成为未来的研究热点之一.

(6) 通用型与专用型

未来内存分配器的发展可能会出现两个趋势：一种是集碎片问题、多线程、内存介质等各方面考虑于一身的全能通用型分配器，属于在横向上扩展分配器适用的范围；另一种就是针对不同领域、不同负载的专用型分配器，属于在纵向上深度优化的设计. 很多公司如 Google，Facebook 等都在自身应用的不断驱动下进行内存分配器的优化工作. 我们相信学术界和工业界会推动内存分配器往横向和纵向两个方向发展，并且它们也将在发展中相互借鉴.

综上所述，考虑到大数据应用对计算速度，存储容量、系统功耗和性能等方面的综合需求，内存分配器的发展将仍然与计算机系统各个领域的变革与发展紧密结合在一起. 在新型的计算机体系结构下，如何充分发挥动态内存分配器在软件层面的潜力，通过各种优化方法弥补其缺陷，尽可能权衡各项性能指标，将成为内存分配器未来的发展趋势和重点研究方向.

5　结　论

本文总结了最近三十年动态内存分配器相关研究，分析了当前内存分配器设计的优势和缺陷，探讨了推动分配器发展的历史原因. 最后提出了一套全面、多维度评价分配器的性能指标，并展望了该领域未来的研究和发展方向.

随着 NVRAM 技术的快速进步和 3D XPoint 产品容量、性能等各方面的不断提升，DRAM 终将被取代. 届时，包含内存分配器在内的针对 DRAM 优化的传统内存管理套件将面临一场大的变革. 另外，随着机器学习和人工智能的不断发展，有理由相

信未来的动态内存分配器会越来越智能,甚至会自动根据不同应用分配模式进行调整.当然也期待计算机领域,无论是 CPU、存储还是软件发生新的重大变革,引导内存分配器往更新的方向发展.

参 考 文 献

[1] Wilson P R, Johnstone M S, Neely M, Boles D. Dynamic storage allocation: A survey and critical review//Baler H G eds. Memory Management. Lecture Notes in Computer Science 986. Berlin, Germany: Springer, 1995: 1-116

[2] Lee B C, Ipek E, Mutlu O, et al. Architecting phase change memory as a scalable dram alternative. ACM SIGARCH Computer Architecture News, 2009, 37(3): 2-13

[3] Lee B C, Zhou P, Yang J, et al. Phase-change technology and the future of main memory. IEEE Micro, 2010, 30(1): 143

[4] Fackenthal R, Kitagawa M, Otsuka W, et al. 19.7 A 16Gb ReRAM with 200 MB/s write and 1 GB/s read in 27 nm technology//Proceedings of the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). San Francisco, USA, 2014: 338-339

[5] Kültürsay E, Kandemir M, Sivasubramaniam A, et al. Evaluating STT-RAM as an energy-efficient main memory alternative//Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). TX, USA, 2013: 256-267

[6] Berger E D, McKinley K S, Blumofe R D, et al. Hoard: A scalable memory allocator for multithreaded applications. ACM SIGPLAN Notices, 2000, 35(11): 117-128

[7] Krishnan M R. Heap: Pleasures and Pains. Washington, USA: Microsoft Developer Newsletter, 1999

[8] Larson P Å, Krishnan M. Memory allocation for long-running server applications. ACM SIGPLAN Notices, 1998, 34(3): 176-185

[9] Vee V Y, Hsu W J. A scalable and efficient storage allocator on shared-memory multiprocessors//Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks, 1999 (I-SPAN'99). Perth/Fremantle, Australia, 1999: 230-235

[10] Berger E D, Zorn B G, McKinley K S. OOPSLA 2002: Reconsidering custom memory allocation. ACM SIGPLAN Notices, 2013, 48(4S): 46-57

[11] Zorn B, Grunwald D. Empirical measurements of six allocation-intensive C programs. ACM SIGPLAN Notices, 1992, 27(12): 71-80

[12] Zorn B. The measured cost of conservative garbage collection. Software: Practice and Experience, 1993, 23(7): 733-756

[13] Zorn B, Grunwald D. Evaluating models of memory allocation. ACM Transactions on Modeling and Computer Simulation, 1994, 4(1): 107-131

[14] Vo K P. Vmalloc: A general and efficient memory allocator. Software: Practice and Experience, 1996, 26(3): 357-374

[15] Jones R, Lins R D. Garbage Collection: Algorithms for Automatic Dynamic Memory Management//Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Computing. New York, USA, 1996: 12-20

[16] Jeremiassen T E, Eggers S J. Reducing false sharing on shared memory multiprocessors through compile time data transformations. New York, USA: ACM, 1995

[17] Torrellas J, Lam H S, Hennessy J L. False sharing and spatial locality in multiprocessor caches. IEEE Transactions on Computers, 1994, 43(6): 651-663

[18] Michael M M. Scalable lock-free dynamic memory allocation. ACM SIGPLAN Notices, 2004, 39(6): 35-46

[19] Evans J. A scalable concurrent malloc (3) implementation for FreeBSD//Proceedings of the BSDCan Conference. Ottawa, Canada, 2006: 1-14

[20] Li W, Mohanty S, Kavi K. A page-based hybrid (software-hardware) dynamic memory allocator. IEEE Computer Architecture Letters, 2006, 5(2): 13-13

[21] Schneider S, Antonopoulos C D, Nikolopoulos D S. Scalable locality-conscious multithreaded memory allocation//Proceedings of the 5th International Symposium on Memory Management. Ottawa, Canada, 2006: 84-94

[22] Hudson R L, Saha B, Adl-Tabatabai A R, et al. McRT-Malloc: A scalable transactional memory allocator//Proceedings of the 5th International Symposium on Memory Management. Ottawa, Canada, 2006: 74-83

[23] Saha B, Adl-Tabatabai A R, Hudson R L, et al. McRT-STM: A high performance software transactional memory system for a multi-core runtime//Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, USA, 2006: 187-197

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

[25] Kuszmaul B C. Supermalloc: A super fast multithreaded malloc for 64-bit machines. ACM SIGPLAN Notices, 2015, 50(11): 41-55

[26] Gidenstam A, Papatriantafilou M, Tsigas P. NBmalloc: Allocating memory in a lock-free manner. Algorithmica, 2010, 58(2): 304-338

[27] Feng Y, Berger E D. A locality-improving dynamic memory allocator//Proceedings of the 2005 Workshop on Memory System Performance. Chicago, USA, 2005: 68-77

[28] Gidenstam A, Papatriantafilou M, Tsigas P. Allocating memory in a lock-free manner//Proceedings of the European Symposium on Algorithms. Palma de Mallorca, Spain, 2005: 329-342

[29] Mao Wei, Liu Jing-Ning, Tong Wei, et al. A review of storage technology research based on phase change memory. Chinese Journal of Computers, 2015, 38(5): 944-960 (in Chinese)

(冒伟，刘景宁，童薇等. 基于相变存储器的存储技术研究综述. 计算机学报，2015，38(5)：944-960)

[30] Zhao Wei-Sheng, Wang Zhao-Hao, Peng Shou-Zhong, et al. Re-cent progresses in spin transfer torque-based magnetoresistive random access memory (STT-MRAM). Scientia Sinica (Physica, Mechanica & Astronomica), 2016, 46(10): 107306(in Chinese)

(赵巍胜，王昭昊，彭守仲等. STT-MRAM 存储器的研究进展. 中国科学：物理学力学天文学，2016，46(10)：107306)

[31] Wang Xiao-Ping, Shen Yi, Wu Ji-Sheng, et al. Review on memristor and its applications. Acta Automatica Sinica, 2013, 39(8): 1170-1184(in Chinese)

(王小平，沈轶，吴计生等. 忆阻及其应用研究综述. 自动化学报，2013，39(8)：1170-1184)

[32] Seo S, Kim J, Lee J. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores// Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT). Galveston Island, USA, 2011: 253-263

[33] Herter J, Backes P, Haupenthal F, et al. CAMA: A predictable cache-aware memory allocator//Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems (ECRTS). Porto, Portugal, 2011: 23-32

[34] Masmano M, Ripoll I, Crespo A, et al. TLSF: A new dynamic memory allocator for real-time systems//Proceedings of the 2004 16th Euromicro Conference on Real-Time Systems (ECRTS). Catanic, Italy, 2004: 79-88

[35] Wang Zhen-Jiang, Wu Cheng-Gang, Zhang Zhao-Qing. Dynamic pool allocation on improving heap data locality. Chinese Journal of Computers, 2011, 34(4): 665-675(in Chinese)

(王振江，武成岗，张兆庆. 提高堆数据局部性的动态池分配技术. 计算机学报，2011，34(4)：665-675)

[36] Liu R, Chen H. SSMalloc: A low-latency, locality-conscious memory allocator with stable performance scalability// Proceedings of the Asia-Pacific Workshop on Systems. Seoul, Korea, 2012: 15

[37] Moraru I, Andersen D G, Kaminsky M, et al. Consistent, durable, and safe memory management for byte-addressable non volatile main memory//Proceedings of the 1st ACM SIGOPS Conference on Timely Results in Operating Systems. Framington, USA, 2013: 1

[38] Yun H, Mancuso R, Wu Z P, et al. PALLOC: DRAM bank- aware memory allocator for performance isolation on multicore platforms//Proceedings of the 2014 IEEE 20th Real-Time and Embedded Technology and Applications Symposium (RTAS). Berlin, Germany, 2014: 155-166

[39] Aigner M, Kirsch C M, Lippautz M, et al. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. ACM SIGPLAN Notices, 2015, 50(10): 451-469

[40] Treiber R K. Systems programming: Coping with parallelism. New York, USA: International Business Machines Incorporated, Thomas J. Watson Research Center, 1986

[41] Yu S, Xiao N, Deng M, et al. WAlloc: An efficient wear-aware allocator for non-volatile main memory//Proceedings of the 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC). Nanjing, China, 2015: 1-8

[42] Schwalb D, Berning T, Faust M, et al. nvm malloc: Memory allocation for NVRAM//Proceedings of the ADMS@VLDB. Kohala Coast, USA, 2015: 61-72

[43] Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory//Proceedings of the 9th European Conference on Computer Systems. Amsterdam, Netherlands, 2014: 15

[44] Zhu J, Li S, Huang L. Wamalloc: An efficient wear-aware allocator for non-volatile memory//Proceedings of the 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). Wuhan, China, 2016: 625-634

[45] Bhandari K, Chakrabarti D R, Boehm H J. Makalu: Fast recoverable allocation of non-volatile memory//Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Amsterdam, Netherlands, 2016: 677-694

[46] Fraser C W, Hanson D R. A Retargetable C Compiler: Design and Implementation. Boston, USA: Addison-Wesley Longman Publishing Co., Inc., 1995

[47] Gay D, Aiken A. Memory Management with Explicit Regions. New York, USA: ACM, 1998

[48] Chilimbi T M. Efficient representations and abstractions for quantifying and exploiting data reference locality. ACM SIGPLAN Notices, 2001, 36(5): 191-202

[49] Brown A, Guyer S Z, Jiménez D A, et al. The C-Breeze compiler infrastructure. Department of Computer Sciences, The University of Texas, Austin, TX: Technical Report: UTCS-TR01-43, 2004

[50] Grunwald D, Zorn B, Henderson R. Improving the cache locality of memory allocation. ACM SIGPLAN Notices, 1993, 28(6): 177-186

[51] Johnstone M S, Wilson P R. The memory fragmentation problem: Solved?. ACM SIGPLAN Notices, 1998, 34(3): 26-36

[52] Barnes J, Hut P. A hierarchical $O(N\log N)$ force-calculation algorithm. Nature, 1986, 324(6096): 446-449

[53] Lever C, Alliance S N, Boreham D. malloc() performance in a multithreaded Linux environment//Proceedings of the USENIX Annual Technical Conference. New York, USA, 2000: 56-56

[54] Luk C K, Cohn R, Muth R, et al. Pin: Building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Notices, 2005, 40(6): 190-200

[55] Henning J L. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4): 1-17

[56] Aigner M, Kirsch C M. ACDC: Towards a universal mutator for benchmarking heap management systems. ACM SIGPLAN Notices, 2013, 48(11): 75-84

[57] Nalli S, Haria S, Hill M D, et al. An analysis of persistent memory use with WHISPER//Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems. Xi'an, China, 2017: 135-148

LIU Xiang, born in 1993, M. S. candidate. His research interests include non-volatile memory, wear-leveling, memory management, memory allocator.

TONG Wei, born in 1977, Ph.D., lecturer. Her research interests include massive networked storage system, non-volatile memory and input/output virtualization.

LIU Jing-Ning, born in 1957, Ph. D., professor. Her research interests include computer system structure, computer storage system, high-speed interface and channel technology.

FENG Dan, born in 1970, Ph. D., professor. Her research interests include information storage system, network storage, solid state storage.

CHEN Jin-Long, born in 1995, M. S. candidate. His research interests include memory management, memory allocator.

Background

Research on dynamic memory allocator is a fundamental study in the field of memory management. With the development of multicore processor in recent years and the advent of novel non-volatile memory, researches on dynamic memory allocator also focused on different optimization directions. Numerous studies focused on above those have been carried out both at home and abroad.

In this paper, we summarize the research on dynamic memory allocator and analyze the historical reasons behind. In addition, a set of evaluation indicators on dynamic memory allocator is given. Besides, we pointed out advantages and disadvantages of current technologies and predicted the directions of further research.

In recent years, our research tram has focused on the related research with dynamic memory management, such as memory allocator based on NVRAM, file system for NVRAM memory, heterogeneous hybrid memory architecture and so on.

This work is supported by the National High Technology Research and Development Program (863 Program) of China (Nos. 2015AA015301, 2015AA016701), project "Key Technologies and Systems for Large Data In-Memory Computing", sub project "Research and Development of Heterogeneous Hybrid Memory Architecture".