METIS: 一个使用历史回收信息的智能内存分配器

Shijie Xu\*, Qi Guo+, Gerhard Dueck\*, David Bremner\*, Yang Wang\*

\*IBM Center for Advanced Studies (CAS Atlantic)

University of New Brunswick, Fredericton, E3B 5A3, Canada

+ECE, Carnegie Mellon University, Pitsburgh, PA 15213, USA

E-mail: {shijiexu, gdueck, bremner, yangw}@unb.ca, [qguo1@andrew.cmu.edu](mailto:qguo1@andrew.cmu.edu)

 Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS 2015, July 4, 2015

ISBN 978-1-4503-3657-4/15/07

DOI: http://dx.doi.org/10.1145/2843915.2843920

**摘要**：动态内存管理在过去十年中受到广泛关注。减少内存碎片是实现高效内存管理的主要考虑因素。然而，对于一些循环密集的应用（例如：Apache HTTP和Ngnix），由于反复的内存分配和内存释放，最优秀的动态内存分配器也不能有效减少碎片。为了解决这个问题，我们提出了一个名为METIS的智能内存分配器，专为循环密集型应用而设计。在METIS中，程序的运行时分为两个阶段：分析阶段和激活阶段。在分析阶段，METIS会建立一个模型，用于分组历史内存分配指令，创建的对象在相同的垃圾收集循环期间相互连接并可能一起回收。在激活阶段，METIS会创建一个区域组（一个连续的存储空间且以一个整体回收），以便为模型中的一个指令组提供分配指令。我们使用扩展的SPECjvm2008跟踪实验显示，可以减少全局堆中的79%的真实碎片，并且在区域组中虚假碎片减少的比例更大。

1. **介绍**

动态内存管理是编程语言的常用功能，它允许从堆空间按需分配内存，以实现更有效的用户程序。然而，动态内存管理存在两个设计挑战。第一个是如何有效地为运行的程序动态分配内存，第二个是如何有效地减少碎片，从而最大限度地减少对程序的影响。在过去50年中，通过利用多线程编程和堆分组，第一个挑战已经在各种广泛使用的内存分配器（如DL Malloc[1]，Google TCMalloc [3]和Free/Open BSD Malloc[24,2,6]。

为了最小化内存碎片，一个众所周知的解决方案是压缩，例如部分压缩[12,14,27,32]和并发压缩[27,33,22,32]。内存压缩将分配的对象移动到连续的空间，为后续分配留下一个大的连续可用空间。虽然内存压缩可以通过减少浪费的内存来有效利用内存空间，但由于对象移动和引用更新，可能需要停止正在运行的程序。减少内存碎片的另一种方法是在函数启动和退出时自动分配和释放堆栈上的对象[26]。这种方法可以在理想情况下消除几乎所有的碎片，然而，函数参数传递不方便，并且需要对编译器进行额外的工作（例如，部分转义分析来确定哪些对象适合于堆栈分配[31]）。此外，一些基于区域的分配器以及一些基于区域的垃圾回收（GC）算法通过改变区域大小进行碎片整理。例如，Hoard存储器管理器[8]通过将每个处理器本地区域的最大块返回到全局堆来减少分段，如果该区域的使用低于某个分数。

与压缩相比，只有在对象分配或释放后才能解决分片，我们提出了一种称为METIS的智能内存分配器，可以在分配时间内解决分片。METIS适用于循环密集型应用，如Web服务器和数据库管理系统。对于这些应用程序，我们观察到它们具有相对较长的循环执行时间，每个循环都有类似的对象分配和释放行为。例如，Apache HTTP Server不断重复处理验证，处理和响应生成的处理顺序。在此过程中，一旦服务器接收到请求，就会创建一个请求上下文，一旦响应被传递就被释放。因此，很有可能不同的循环迭代在对象分配和释放模式上具有相似性，这被现有的分配器忽略。

METIS的基本思想是在相邻的位置放置相互连接并可能由一个GC回收的对象。因此，METIS减少了潜在的小的空间分离的内存“空洞”的数量，因为这些小空洞合并成一个大的自由块。如图1所示，对象O4，O5和O6总是由一个GC回收。因此，B方案优于A方案，因为B方案很可能导致较少的碎裂。更具体地说，METIS将程序生命周期分为两个阶段：分析阶段和激活阶段。在分析阶段，构建了具有多个指令组的模型，用于跟踪分配指令，其中由一个组指令创建的对象总是相互连接并一起回收。然后，在激活阶段，如果需要，将会创建连续区域组，并为单个分配指令组留存。

A O1 O4 O2 O3 O5 O7 O6

B O1 O2 O3 O4 O5 O6 O7

图1内存分配和碎片示例

本文介绍了METIS，一种利用对象分配和释放历史记录的循环密集型应用程序的内存分配器。在本文中，我们提供了METIS的总体思路，其详细设计和实验结果。据我们所知，METIS是在对象分配期间利用历史回收信息的第一个响应式分配器。METIS不是用来取代现有的内存分配器。相反，它可以作为现有分配器的有效补充。

本文剩余部分这样安排：第2节简要总结了动态内存管理的背景。第3节介绍了METIS的概述，它由分析阶段和激活阶段组成。第4节详细介绍了分析阶段，第5节详细介绍了激活阶段。最后，第6节提供实验结果，第7节为本文做了一个总结。

1. **背景**

**2.1 动态内存分配**

目前比较先进的内存分配器主要关注如何减少并发和多核环境中的争用和虚假共享来达到高效的内存分配，而不是最小化内存碎片。例如，有些分配器使用并发数据结构来组织对内存，而有些基于区域的分配器，比如BSD Malloc，Hoard Allocator以及Google TCmalloc，会根据线程或者处理器的数量和全局堆的大小，将堆分成多个区域。基于区域的解决方案目前仍然是有效的，而且已经部署在诸如BSD和Solaris的产品中。

大多数现有的碎片整理解决方案通过将这些片段移动到连续的内存空间来减少分段，这称为压缩。由于这些解决方案仅在分片生成后才会发生作用，因此在内存分配过程中无法防止分片。例如，大多数基于区域的解决方案，比如TCMalloc和Hoard，提供了一种在每个进程或者线程的本地区域（高速缓存）和全局堆之间交换空闲块的方式，从而可以减少碎片的累积大小。同时，其他一些研究用低成本下的压缩效率评估空间，他们认为如果成本不高，碎片整理是可以接受的。

Sangho等人提出了一种针对TCMalloc的反馈定向优化方案。在这种方法中，使用批量大小来确定区域和全局堆之间的块交换的时序，并且通过基于应用的分析数据的迭代算法来计算最佳大小。

**2.2 垃圾回收**

垃圾回收（GC）会在运行时自动回收无用对象占用的内存。它解决了指针悬挂的问题，防止内存泄漏，提高了软件工程的生产力。 基本GC算法之一是标记-清除（Mark-Sweep）算法[21]。 在标记-清除算法的初始收集阶段，如果从线程的根集合开始仍然可以访问到，就将对象标记为“alive”。在扫描阶段，未标记的对象被标识为“dead”，并且回收对应的内存。 此外，压缩是GC算法的最后一步，这会为之后的内存分配创造很大的空间。然而，GC经常会导致性能下降，因为它通常会暂停程序执行以进行对象的移动操作。

解决GC带来的性能影响的方法有很多。其中不乏一些采用新的架构，例如多核心处理器（multi-core processor）和闪存，用以加快GC的速度，减少GC时停顿的时间。一些其他解决方案将堆空间分成多个区域并行执行多个部分GC，而不是一个全局GC，例如Oracle HotSpot[4]和Immix[9]中的Generation Collection(GC)。这种分而治之的思想可以高效运行，而且也有的应用到了现有的一些工业产品中。最近有几种方法尝试通过利用开发人员的启发式知识来提高GC的效率，例如用于模糊引用的保守收集器[29]，识别无用对象的暗示收集器[28]，在离线分析阶段标识垃圾收集点，能根据成本效应模型选择一个托或者全堆GC的识别垃圾回收提示（Garbage Collection Hints）。

关于GC的相关工作中，有一些和我们的工作类似。例如，为了减少GC期间找到废弃对象的开销，Barry Heyes针对具有相似生命周期的对象进行聚类，并在决定检查相应集群中的对象之前检查每个集群代表性的对象废弃。在他的方案中，聚类是基于离线分析的对象的内存分配和释放模式。类似的，Hirzel等人介绍了一种基于对象连通属性的基于连通性的垃圾收集器（Connectivity-Based Garbage Collectors，CBGC）。在CBGC中，通过程序在离线编译期间获得的连通性分析结果来确定对象分配的选择。由于静态离线分析相对较为保守，Hayes和Hirzel的解决方案可能无法适应不同程序运行期间的对象分配和释放。

1. **概述**

METIS解决了有很多重复执行路线的循环密集型应用程序面临的问题。这类应用的其中之一就是web服务器，web服务器不断的接收、解析处理请求以及产生响应。在这些循环迭代中，会有很多重复的内存分配和回收模式。

METIS利用历史对象回收信息来指导之后的对象内存分配。它把一个程序的运行时间分为两个阶段：阶段剖析和激活阶段。METIS中的核心是指令计数器模型（IC Model），包括指令计数器组（IC Groups）、元组和IC列表。在分析阶段，该模型在GC期间使用可回收对象图的IC进行训练，以便使同一个IC组的指令创建的对象总是互连的，并且可能一起回收。在激活阶段，作为连续存储区域并由元组索引的区域组用于给来自一个IC组的分配指令提供服务。

总而言之，METIS的目的是通过将可能被回收的对象放置在相邻区域中来减少内存碎片。METIS并不意图替代现有的分配器和GC算法，相反，它是建立在现有的分配器和GC之上。虽然我们原型中的模型强烈依赖于GC，但可以通过监视释放指令的调用来轻松的扩展到其他不受管理的语言上，例如C语言。

0.7

0.2

0.6

0.3

0.8

0.1

O4.ic

O0.ic

O5.ic

O6.ic

O8.ic

group1

group3

group2

group4

group5

图2 IC模型示例

**4. 分析阶段**

**4.1 指令计数器**

与程序计数器类似，我们将指令计数器（IC）定义为内存分配指令的高级抽象。IC由三个字段组成：指令地址、操作码和操作数。指令地址由方法名称和偏移量构成，偏移量是距相应方法开头的距离。操作码是指示指令操作的一个字节符号，它可以是对象分配，对象赋值，对线程根集的对象添加，线程根集合中的对象删除或跳转。操作数是操作码操作的不同值。在我们的跟踪中，每个指令（或IC）由指令的地址决定，称为指令键。

**4.2 IC组和IC模型**

IC组是一些IC的集合，通过同一个IC组中的IC创建的对象是互联的，他们很有可能被一次GC回收。因此，在GC清除期间，在GC收集阶段之后，其对象全部标记为“dead”的可回收对象图将被映射到IC组中。

设定一个字段parentICs用于实现映射，追踪IC，和持有该对象引用的其他对象。这个字段在每次出现赋值指令的时候更新。例如，如果B对象被赋值给了A对象的某个字段，那么对象A的IC添加到对象B的parentIC里。此字段有助于保持可回收的对象图结构，而且GC期间IC组构建是有利的。

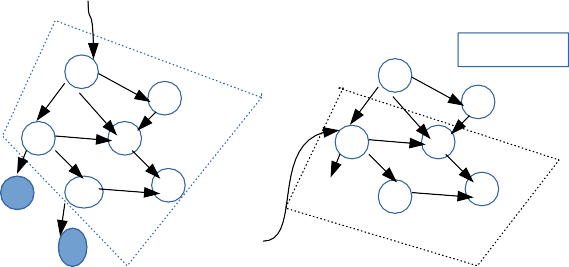
由于应用程序中的循环，具有相同或不同操作数的指令可能会多次执行。因此，由该IC创建的对象可以在不同的可回收对象树中。换句话说，IC可以被多于一个IC组共享。图2中示出了一个说明性的例子，其中由'O5.ic'创建的对象分成两组，即group3和group4。IC模型由IC列表、元组和多个IC组组成，如图2所示。在模型中，每个元组由IC索引，数组中的元组由两个字段组成 ：概率和IC组引用。

元组的概率被计算为该IC在相应的IC组中的次数与在分析阶段期间执行该IC的总次数的比率。此外，数组的所有元组按其概率按降序排序。换句话说，IC主要可能在的IC组始终处于元组的开头，这个有序的元组有利于减少元组遍历的代价。

在模型中，元组的长度是可变的。理想长度仅为1，表示由该指令创建的对象始终由与同一IC组IC集成的其他对象一起回收。为了减少数组的长度，开始时配置了两个规则：a）元组的长度应该小于一个预定义的阈值（例如3），b）如果一个尾元组的概率小于另一个 阈值（例如0.1），它将被移除。

**4.3 IC 模型训练**

在分析阶段，通过回收对象来创建和训练IC模型。训练的主要任务是将可回收对象的IC赋值到IC组中，使来自同一IC组的IC创建的对象互连，并可能由一个GC回收，并由元组在IC列表和IC组之间建立连接。检测到物体在GC中可回收时，训练开始，包括两个步骤：组的识别和IC的添加。



start

Parent ICs

O0 O0

O3 O3

O1 O2 R O1 O2

O6 O4

O5

r0

O4

O5

O7

start

图3 GC树回收示例

**组的识别**

可回收对象图是有向对象图，图中的有向边指示源对象中的一个字段引用目标对象。 可回收图中的所有对象都不能从线程的根集中访问，并被标记为“dead”。

IC组识别负责确定给定可回收对象图的所有IC的IC组。主要的困难是在GC中无法确定扫描“dead”对象的顺序，换句话说，一个可回收对象图中的非根对象将会被首先清除，而且很容易确定该对象子对象的以及该对象自身的IC组。但由于原始可回收图的部分已经被销毁，所以不可能为其余的对象找到正确的IC组。

我们以图3种左侧的可回收对象图为例。清除从对象O1开始，并且O1及其（直接和间接）子节点的IC可以在这些对象最终被重新回送到系统内存之前被标记到同一个IC组。然而，由于O1及其子元素已经从图中移除，所以之后不可能找到O0和O3的IC的正确IC组。

为了保持可回收对乡土的IC结构，即使一些对象已经被回收，一个新的字段parentIC也将会被添加到IC组。IC组的parentIC的IC创建的对象可能会引用组的IC创建的对象。以图3为例，将对象O1，O2，O3和O4添加到IC组ICG中。在将这四个对象从图形中删除并回收到系统内存之前，将它们的IC添加到ICG中，并将O1和O2的parentIC中的IC添加到ICG的parentIC（对象和IC组都有一个parentIC字段）。

于是，识别可回收对象的IC组有两个步骤。在第一步，如果访问的IC在模型的IC列表中，METIS会递归地访问对象的子节点并构建IC集合IC0。在第二步中，所识别的IC组是IC0中大部分IC的IC组。如果IC0为空，则创建并返回新的IC组。

**IC添加**

一旦确定了IC组，被访问对象以及被访问对象的父IC的IC分别被添加到IC组及其parentIC。在模型上的另外两个更新是1）如果这些IC没有在模型的IC列表中，插入将被访问对象的IC插入，然后通过元组建立这些IC和所识别的IC组之间的连接，以及2）计算元组中的相应概率并对数组中的元组进行排序，以便具有较高概率的元组位于元组的开头。

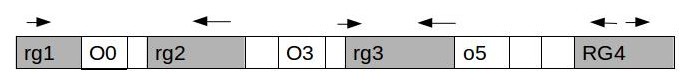
 从parentIC插入IC与被访问对象的IC的插入略有不同。对于前一种情况，模型中识别的IC组的IC列表条目和对应元组的相应条目与错误值相关联，表示该IC创建的对象尚未被先前的GC清除操作访问过。稍后访问该条目后，此值将被删除。在分析阶段结束时，也会删除具有错误值的条目，以避免激活阶段模型中的潜在的错误命中。将parentIC的IC添加到模型中的解释是，即使部分图表已被销毁，它仍保留可回收对象图的IC关系，这已在**组识别**中讨论过。

图4 全局堆和区域组

如图3中右图所示，扫描从O1开始，从O1可到达的对象的IC标有一个IC组，并插入到IC列表中。 同时，图中未被访问的对象O0和O3的IC也作为parentIC插入到IC列表中，并且它们都与错误值相关联。

**5. 激活阶段**

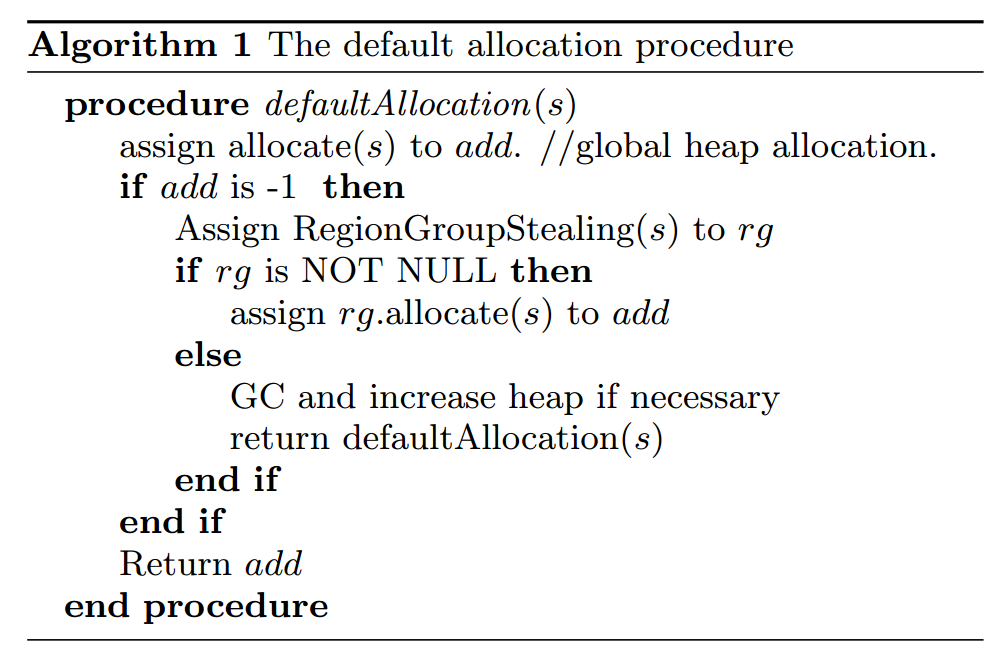
一旦满足一些预定义的标准，执行就进入激活阶段。这些标准可以是一些GC或指令的覆盖范围超过一定阈值。激活阶段，内存堆分为两部分：全局堆（图4中白色区域）和区域组堆（图4灰色区域）区域组是从全局对分配的连续存储器空间，并且保留用于制定IC组的分配指令。它也是动态管理的，可以在回收后发布到全局堆。

**5.1 整体过程**

共有三个程序：全局堆分配，区域组分配和窃取分配。 这三个过程一起工作以提供内存分配请求，整个工作流程如算法2所示。

**全局堆分配**

这是默认分配过程，其中分配器以与现有分配策略相同的方式工作，例如，来自堆空闲列表的首次拟合内存。当服务分配指令不在模型的IC列表中时，或者任何区域组中的块都不可用时，会发生此分配。如果全局空闲列表中没有足够的可用空间，则会触发GC。此外，如果GC之后的可用空间与总堆内存的比率仍低于预定阈值，METIS会增加整体内存大小。 该过程部分实现在算法1中的体部分中。



**区域组分配**

一旦在模型的IC列表中找到分配指令，分配器就成为区域组分配。 换句话说，在分析阶段，我们的模型已经记住了IC，并且可能有一个最佳的分配方式。

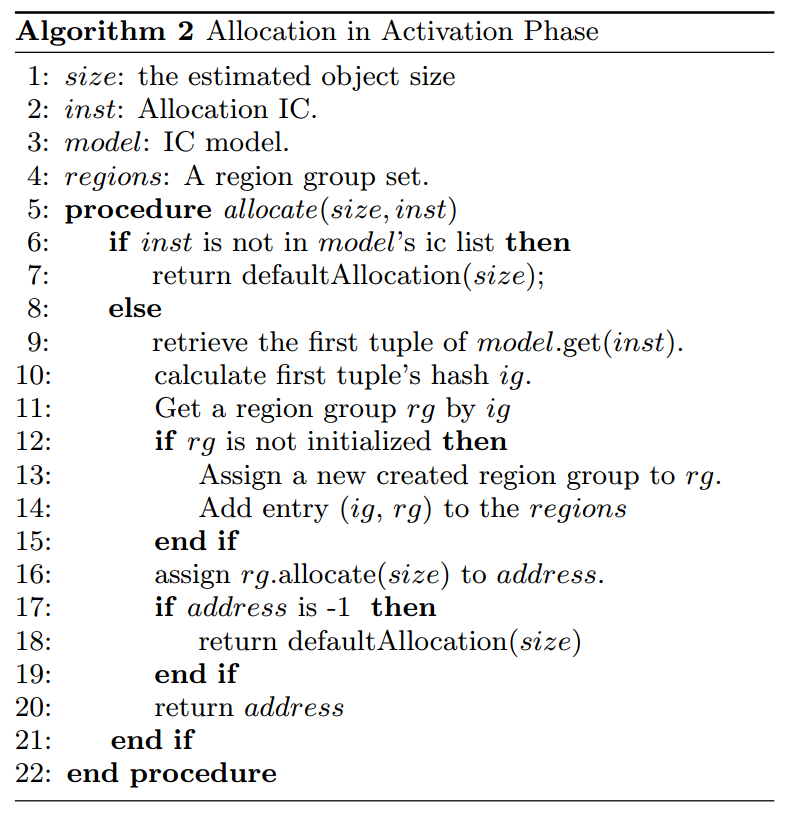
**区域组窃取分配**

窃取是指区域组从另一个区域组（目标区域组）借用空间来服务当前请求，这是在垃圾回收之前分配的最后一步。当窃取失败时，即使它仍处于循环中，完整的GC将被触发。

**5.2 区域组分配**

区域组分配由区域组创建和内部区域组分配两个主要步骤组成。

在创建期间，区域组的大小被估计为在分析阶段期间相应可回收图中的对象的累积大小，加上大约是平均对象大小的三倍的缓冲区大小。如果为区域组分配缓冲区但很少使用缓冲区，则缓冲区仅适用于前几个区域组，例如我们的原型中的十个。

一旦在模型的IC列表中找到IC，如果没有已经连接到IC的区域组，则会从全局堆中创建一个区域组。 然后，分配继续进行，找到区域组内的最佳地址。 不同区域组的分配策略可能不同。 例如，区域组的空闲块可以通过双向列表链接，并且分配方向可以是向前或向后，如图4所示。为了有效地回收区域组，区域组在创建时由其对应的IC组进行索引。 因此，有两个散列计算来确定模型IC列表中IC的区域组。第一个是从IC到IC元组，第二个是从元组中的第一个元素引用的IC组到目标区域组。

**5.3 区域组窃取**

在窃取期间，区域组（目标区域组）提供来自多于一个IC组的IC。 虽然这打破了区域组一个区域组只是来自同一个IC组的IC的设计规则，但它是GC和压缩之前的最后一步。仅当源区域组不足以进行分配时，才分配窃取，而目标区域组仍然具有足够的可用内存。

窃取从目标区域组选择开始。 与区域组中的分配类似，窃取策略可以是首次匹配或基于缓冲的。 对于前者，选择具有足够空闲空间的第一区域组，而在后者中仅选择具有足够空闲缓冲空间的第一区域组（即区域组末端的缓冲区域中的自由空间）。

为了最小化入侵IC对目标区域组的干扰，避免区域组内的潜在内存“空洞”，被窃取区域组中的分配方向由区域组的策略决定。对于首次配对分配策略，分配方向与原始内部区域组分配方向相反。对于基于缓冲区的窃取，入侵IC的分配限制在该区域组的缓冲区。

窃取可以平衡不同区域组之间的工作量。 由于对象局部性，一些区域组比其他组访问更频繁。通过为将到热区域组的IC从较少使用的区域组临时借用空间用于减轻区域组的这种不均衡的工作量。另外，目标区域组的这些借用空间预计将很快回收，因此对目标组本身的影响将会尽可能的减少。

**5.4 GC与区域组回收**

区域组随着死亡对象回收。

METIS中的默认GC算法是标记扫描。首先从区域组和全局堆收集并清除“死”对象。不同区域组的回收可以并发进行，因为区域组是独立的。此外，如果该区域组中没有活动对象，则还可以将区域组作为整体回收到全局堆。

表1 METIS的参数

|  |  |  |
| --- | --- | --- |
| Heap Size | 100000 |  |
| Threshold | 0.8 | Threshold to increase |
|  |  | heap size after GC |
| Heap incremental ratio | 0.2 | Increase by 20% |
| Instruction Block Size | 230000 | size of |
|  |  | instruction block |

**6. 结果**

METIS的实现是一个跟踪解释器，他也实现了标记清除，而不用带有循环检测压缩和引用计数的垃圾回收。它接受跟踪文件作为输入并重现内存操控操作。 通过在IBM J9 [5]上运行SPECjvm2008基准套件（包括Sunflow，Compress，XML-transformer等）来收集这些痕迹。 由于跟踪文件的较大，我们的实验采用第一个指令块，它的大小足以包含复杂的程序工作流程，例如if-else和while循环作为输入，并用带着随机整数n重复执行该块，表示是否执行第n个赋值指令。另外，在解释期间，一个新创建的对象的id会通过添加最后一个循环中的对象的最大的id，以避免对象id冲突。这样做的目的是在实际程序中模拟大范围的循环操作。

跟踪中支持五个内存操控操作，即对象分配，对象赋值，线程根集合中的对象添加，线程根集合中的对象引用删除以及跳转。这五个操作是今天内存操控的基本操作。

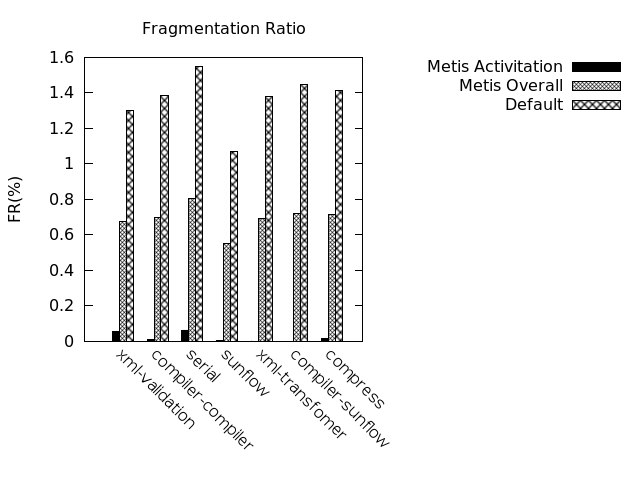
在我们的实验中，METIS与没有内存压缩的标记清除进行比较（以下简称“DEFAULT”）[21]。 我们的实验参数如表1所示。

图6 平均碎片比率对比

**6.1 GC效率**

GC效率（GE）定义为GC之后的回收对象大小与GC之前的总占用内存大小的比率。GE的规模越大，GC效率就越高。

图5显示了来自SPECjvm2008的METIS和Default的六个基准的GE。在这些图中，x轴是由执行的分配指令的数量测量的时间，y轴是GE值。我们可以看到METIS和DEFAULT的GC效率相对一致。一个小小的影响是GC在METIS中比在DEFAULT中稍早出现。这是因为METIS中的区域组的一些预先分配的缓冲区不再可用于从全局空间进行一般分配，从而导致更少的全局内存和较早的GC。

**6.2 碎片比率（FR）**

碎片比率定义为当GC发生时，全局区域中的可用空间大小与总全局堆大小的比率。对于DEFAULT，全局堆大小与整个堆大小相同，而它是METIS的非区域组的大小。全局区域中的空闲空间都是碎片，因为它们不足以满足当前的内存分配。

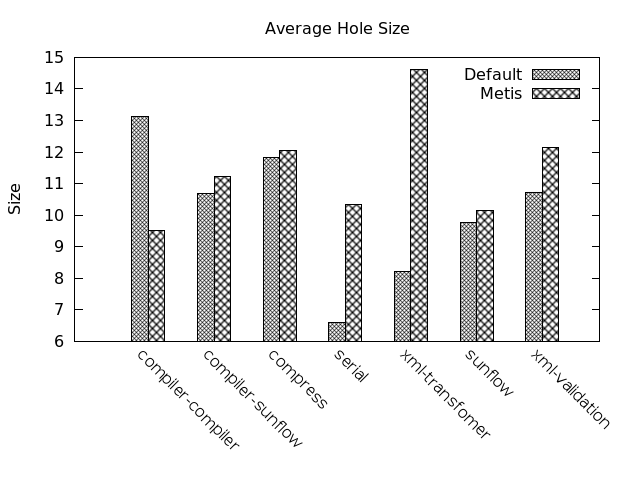
DEFAULT和METIS的平均碎片比率（FR）如图6所示。在本实验中，跟踪将会重复至系统报告内存不足或成功退出。在该图中，黑色显示METIS激活阶段的平均FR，而灰色显示METIS总体执行的平均FR（包括分析和激活阶段）。可以清楚地看到，METIS的平均FR可以显着降低48.9％。 在某些情况下，例如XML-Transfomer和Compiler-Sunflow，在METIS的激活阶段可以完全消除碎片。

图7平均空洞大小比较

表2 全局对中的空洞数量

|  |  |  |  |
| --- | --- | --- | --- |
| Average hole number | Default | Metis |  |
| compiler-compiler | 129.8 | 35 |  |
| compiler-sunflow | 119.6 | 38.5 |  |
| compress | 128.4 | 33.67 |  |
| serial | 135.2 | 44.8 |  |
| sunflow | 138.6 | 39.8 |  |
| xml-transfomer | 133.6 | 30.4 |  |
| xml-validation | 132.2 | 42 |  |

**6.3 内存空洞分布**

平均空洞大小（AHS）是全局范围内的可用内存大小与GC之前的内存空间数量的比值。METIS中的“空洞”（连续可用内存区域）可以位于全局区域（真实碎片）或区域组（伪碎片）。

不考虑“空洞”的位置，如图7显示了不同基准的DEFAULT和METIS实现的AHS。结果表明，METIS的AHS比DEFAULT的大18％，意味着METIS中的“空洞” 如果所请求的内存分配大小变小，则不太可能是碎片。

METIS的优越性在表2和图8中变得更加明显。表2显示了全局堆中的平均空洞数（真实碎片）。根据这个表，METIS可以将全局堆中的空洞数减少70％。 图8显示了Compiler-Compiler套件每个GC之前的空洞数的变化。我们可以看到，偶尔可以完全删除真正的碎片，即第1，第3和第4个GC。 我们还观察到其他情况（即sunflowh和compress）的类似结果。对于这个结果的解释是，在METIS的区域组中，绝大多数的空洞是虚假的，这些空间是保留的，但在GC发

生之前没有被占用。

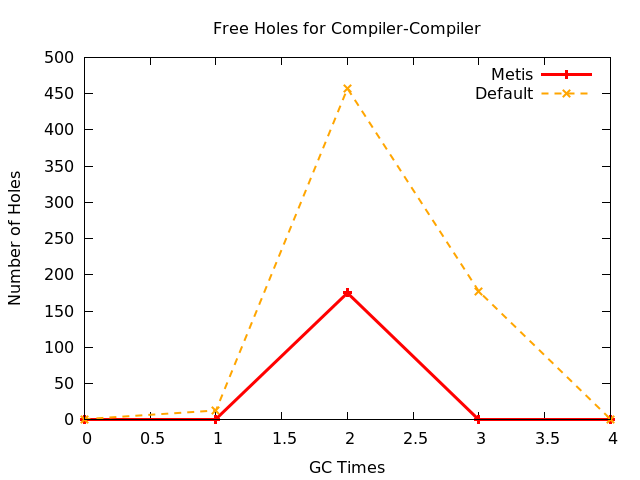
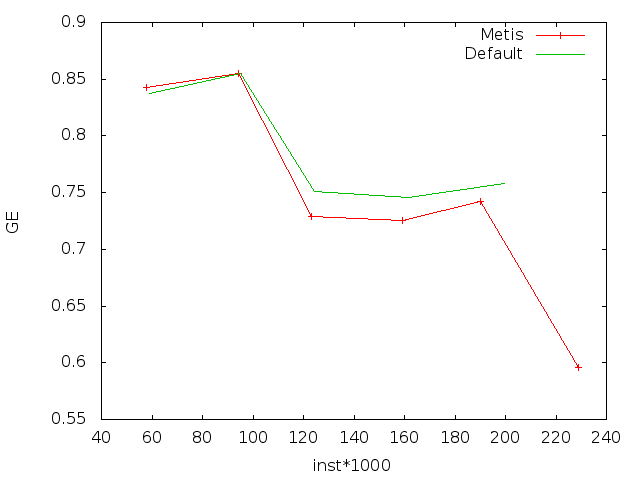
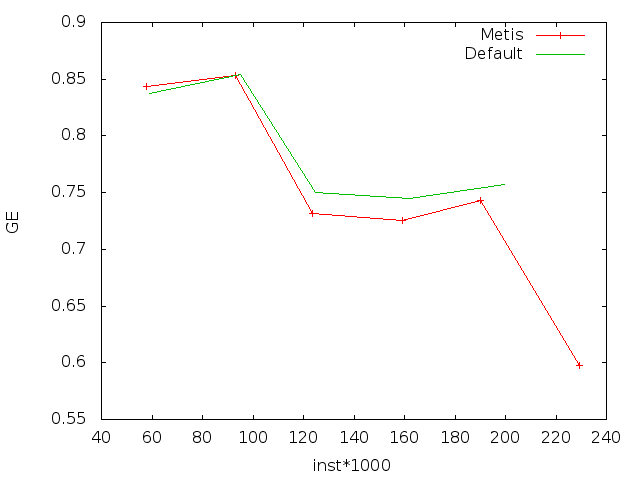
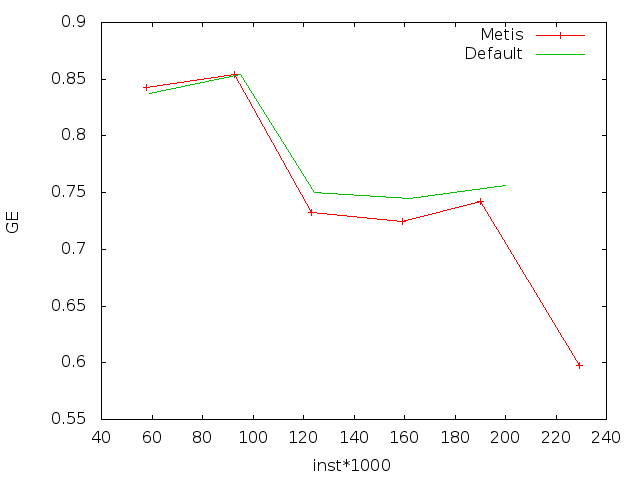
 总而言之，METIS平均将全局堆中的真实碎片数量减少了70％，而对于某些基准则达到了100％。具体来说，区域组中的虚假碎片的平均大小比至少比没有METIS的分配器要大1.8倍。换句话说，因为这些“空洞”足够大并被保留，所以虚假碎片不太可能成为真正的碎片。

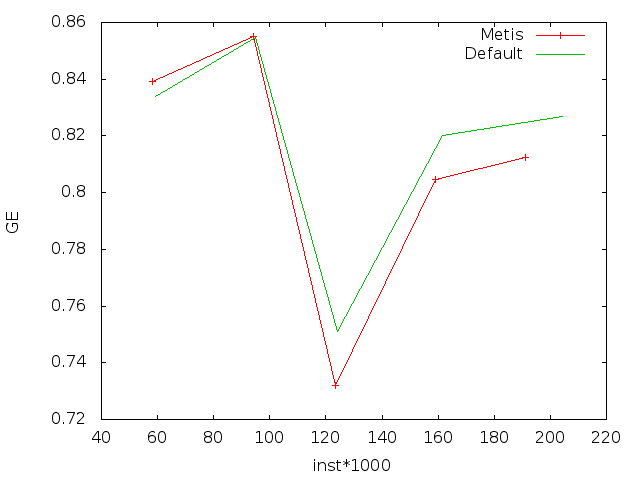
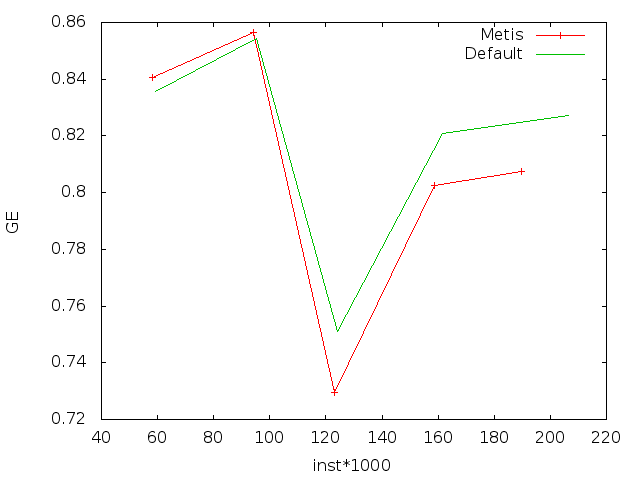
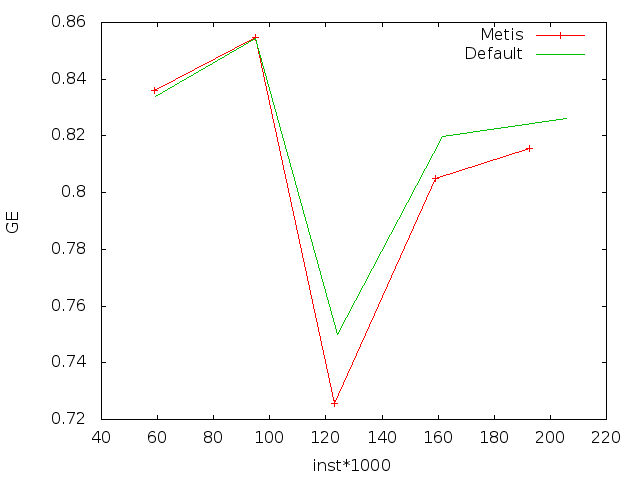
图8 Compiler-Compiler中的全局对空洞数量

**6.4 区域组效率**

 区域组效率（RGE）用于度量GC期间区域组的减少量。它被定义为：

图9显示了不同基准的METIS的RGE随着指令数的变化情况。结果表明，区域组在激活阶段工作高效，因为通过GC可以减少（以整体回收）多达83％的区域组。因为区域组只有在其所有对象都死亡时才可以回收，所以实验结果也表明，分析阶段中训练的模型可以成功地对IC进行分组，由此创建的对象可能被一起回收。

* + 1. compiler-sunflow (b) compiler-compiler (c) compress

(d) sunflow (e) serial (f) xml-validation

图 5 GC效率vs GC时间

**6.5 区域组重要性**

区域组重要性（RGS）用于估计区域组中分配的对象数量。该指标提供了另一种估计区域组效率的方法。RGS的定义是：



与RGE类似，图10显示了随着指令计数变化的RGS，其由基准测试执行的分配指令的数量作为度量。我们可以清楚地看到，多达75％（平均）被放置在区域组中。

|  |  |
| --- | --- |
| 图9 区域组效率 | 图10 区域组重要性 |

**7．结论和未来工作**

在本文中，我们提出了一种名为METIS的智能内存分配器，通过使用历史回收信息来减少潜在的内存碎片。更具体地说，METIS由两个阶段组成：分析阶段和激活阶段。我们将METIS实现为一个跟踪解释器，它可以在JVM上重现真正的内存操作。广泛使用的基准测试结果很好地证明了METIS的效果和效率。

今后，这项工作可以在各方面得到延伸。

**提高模型训练效率**

在当前的实现中，为了避免将一个可回收对象树的IC分配给不同的IC组，在模型训练期间使用父IC集。相关消耗随着回收对象树的复杂性增大而大幅增加。因此，我们有必要构建一个原型来了解程序运行时的训练过程的成本。在未来，通过减少父IC集合的大小并同时训练模型，可以有所改进。

**调整区域组大小**

在当前实现中，区域组的内存空间是连续的，并且一旦创建了组，大小就被固定。为了提高效率，应该启用组大小调整。组大小调整可以增加或减小组的大小。 对于前一种情况，拥挤区域组通过注册全局堆的相邻可用内存来扩展自身。 对于后一种情况，区域边界的空闲内存与全局堆中的可用内存合并。

**支持多线程程序**

为了支持多线程程序，METIS将利用线程ID信息，这是指令操作数之一。 它将基于踪迹执行转义对象分析，并将这些结果反馈给分配器以获得更好的分配策略，例如将转义对象放置在单独的堆区中。

**8. 引用**

[1] Doug Lea Allocator. <http://gee.cs.oswego.edu/dl/html/malloc.html>.

[2] FreeBSD Allocator. http://people.freebsd.org/˜jasone/jemalloc/bsdcan2006/jemalloc.pdf.

[3] Google TCMalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[4] HotSpot Garbage Collection. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#introduction.

[5] IBM Developer Kits. <https://www.ibm.com/developerworks/java/jdk/>.

[6] Open BSD Allocator. http://www.openbsd.org/ papers/eurobsdcon2009/otto-malloc.pdf. [7] Bendersky, A., and Petrank, E. Space overhead bounds for dynamic memory management with partial compaction. ACM Trans. Program. Lang. Syst. 34, 3 (Nov. 2012), 13:1–13:43.

[8] Berger, E., McKinley, K., Blumofe, R., and Wilson, P. Hoard: A scalable memory allocator for multithreaded applications. Tech. rep., Austin, TX, USA, 2000.

[9] Blackburn, S. M., and McKinley, K. S. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2008), PLDI ’08, ACM, pp. 22–32.

[10] Buytaert, D., Venstermans, K., Eeckhout, L., and Bosschere, K. Transactions on high-performance embedded architectures and compilers i. Springer-Verlag, Berlin, Heidelberg, 2007, ch. GCH: Hints for Triggering Garbage Collections, pp. 74–94.

[11] Clebsch, S., and Drossopoulou, S.Fully concurrent garbage collection of actors on many-core machines. SIGPLAN Not. 48, 10 (Oct. 2013), 553–570.

[12] Click, C., Tene, G., and Wolf, M. The pauseless gc algorithm. In Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (New York, NY, USA, 2005), VEE ’05, ACM, pp. 46–56.

[13] Cohen, N., and Petrank, E. Limitations of partial compaction: Towards practical bounds. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2013), PLDI ’13, ACM, pp. 309–320.

[14] Detlefs, D., Flood, C., Heller, S., and Printezis, T. Garbage-first garbage collection. In Proceedings of the 4th International Symposium on Memory Management (New York, NY, USA, 2004), ISMM ’04, ACM, pp. 37–48.

[15] Feng, Y., and Berger, E. D. A locality-improving dynamic memory allocator. In Proceedings of the 2005 Workshop on Memory System Performance (New York, NY, USA, 2005), MSP ’05, ACM, pp. 68–77.

[16] Hayes, B. Using key object opportunism to collect old objects. In Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (New York, NY, USA, 1991), OOPSLA ’91, ACM, pp. 33–46.

[17] Hirzel, M., Diwan, A., and Hertz, M. Connectivity-based garbage collection. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (New York, NY, USA, 2003), OOPSLA ’03, ACM, pp. 359–373.

[18] Huelsbergen, L., and Winterbottom, P. Very concurrent mark-&amp;-sweep garbage collection without fine-grain synchronization. In Proceedings of the 1st International Symposium on Memory Management (New York, NY, USA, 1998), ISMM ’98, ACM, pp. 166–175.

[19] Iyengar, A. Parallel dynamic storage allocation algorithms. In Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on (Dec 1993), pp. 82–91.

[20] Iyengar, A. K. Dynamic Storage Allocation on a Multiprocessor. PhD thesis, Cambridge, MA, USA, 1992. Not available from Univ. Microfilms Int.

[21] Jones, R., Hosking, A., and Moss, E. The Garbage Collection Handbook: The Art of Automatic Memory Management, 1st ed. Chapman & Hall/CRC, 2011.

[22] Kermany, H., and Petrank, E. The compressor: Concurrent, incremental, and parallel compaction. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2006), PLDI ’06, ACM, pp. 354–363.

[23] Lam, K.-Y., Wang, J., Chang, Y.-H., Hsieh, J.-W., Huang, P.-C., Poon, C. K., and Zhu, C. J. Garbage collection for multi-version index on flash memory. In Proceedings of the Conference on Design, Automation & Test in Europe (3001 Leuven, Belgium, Belgium, 2014), DATE ’14, European Design and Automation Association, pp. 57:1–57:4.

[24] Larson, P.-A., and Krishnan, M. Memory allocation for long-running server applications. In Proceedings of the 1st International Symposium on Memory Management (New York, NY, USA, 1998), ISMM ’98, ACM, pp. 176–185.

[25] Lee, S., Johnson, T., and Raman, E. Feedback directed optimization of tcmalloc. In Proceedings of the Workshop on Memory Systems Performance and Correctness (New York, NY, USA, 2014), MSPC ’14, ACM, pp. 3:1–3:8. [26] Molnar, P., Krall, A., and Brandner, F. Stack allocation of objects in the cacao virtual machine. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (New York, NY, USA, 2009), PPPJ ’09, ACM, pp. 153–161.

[27] Pizlo, F., Petrank, E., and Steensgaard, B. A study of concurrent real-time garbage collectors. In Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA, 2008), PLDI ’08, ACM, pp. 33–44.

[28] Reames, P., and Necula, G. Towards hinted collection: Annotations for decreasing garbage collector pause times. SIGPLAN Not. 48, 11 (June 2013), 3–14.

[29] Shahriyar, R., Blackburn, S. M., and McKinley, K. S. Fast conservative garbage collection. SIGPLAN Not. 49, 10 (Oct. 2014), 121–139. [30] Siebert, F. Concurrent, parallel, real-time garbage-collection. In Proceedings of the 2010 International Symposium on Memory Management (New York, NY, USA, 2010), ISMM ’10, ACM, pp. 11–20.

[31] Stadler, L., Wu¨rthinger, T., and Mo¨ssenbo¨ck, H. Partial escape analysis and scalar replacement for java. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (New York, NY, USA, 2014), CGO ’14, ACM, pp. 165:165–165:174. [32] Veldema, R., and Philippsen, M. Parallel memory defragmentation on a gpu. In Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (New York, NY, USA, 2012), MSPC ’12, ACM, pp. 38–47.

[33] Wegiel, M., and Krintz, C. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 91–102.