doi:10.3772/j.issn.1002-0470.2015.02.003

NUMA 结构的高效实时稳定的垃圾回收算法①

廖 彬②[*,**,***,****] 傅 杰[*,**,***,****] 靳国杰[**,***,****] 王一光[*,**,***,****]
王 磊[****,*****] 章隆兵[**,***] 王 剑[**,***]

([*]中国科学院大学 北京 100049)
([**]计算机体系结构国家重点实验室 北京 100190)
([***]中国科学院计算技术研究所 北京 100190)
([****]龙芯中科技术有限公司 北京 100190)
([*****]中国科学技术大学 合肥 230026)

摘 要 针对非一致性内存访问架构(NUMA)在垃圾回收(GC)过程中存在大量的远程内存读写导致GC性能降低的问题,对GC过程的各个阶段进行分析与研究,提出了一种基于NUMA结构的高效实时稳定的GC算法。该算法首先基于NUMA结构改进传统分代GC机制的堆空间布局,然后通过控制GC过程中扫描活跃对象阶段的初始根对象选取、动态负载均衡阶段截取任务队列的选取以及复制活跃对象阶段对象复制位置的选取,大大减少GC过程中的远程访问次数。这种改进的GC机制对所有NUMA结构具有通用性。以Godson-3处理器的NUMA平台为例进行的实验结果显示,优化的GC机制极大地缩短了GC的时间,而且提高了应用程序的性能以及稳定性。在SPECjvm2008测试中,GC时间平均缩短了14.6%(GC总时间缩短4.1% ~41.58%),应用程序的性能平均提升了4.68%(最高提升17.8%),应用程序的性能稳定性提升了76.2%。

关键词 非一致性内存访问架构(NUMA),垃圾回收(GC),分代GC,活跃对象,根对象,动态负载均衡

0 引 言

Java等高级语言程序设计的新对象分配与非活跃对象回收由底层虚拟机的垃圾回收(garbage collection, GC)机制自动完成。在非一致性内存访问架构(non-uniform memory access architecture, NUMA)[1]中,新对象的分配与非活跃对象的回收直接影响应用程序与GC的性能。NUMA结构包含多个CPU,每个CPU可以访问两种内存:本地内存和远程内存。与CPU位于同一节点的物理内存称为本地内存,访存延迟非常低;与CPU在不同节点的物理内存称为远程内存,CPU通过节点互联方式访问,访存延迟要比访问本地内存长。在Godson-3[2]处理器的NUMA结构中,访问远程内存比访问本地内存慢47%。

在Java等高级语言程序设计中,GC的效率直接影响应用程序的性能[3]。在NUMA结构中,如果GC没有考虑NUMA结构访存非一致性的特点,那么实际应用程序的性能和稳定性将会受到很大影响。首先,应用程序在分配新对象时,如果GC机制将新生对象分配在远程内存,应用程序在后续执行

① 国家"核高基"科技重大专项课题(2009ZX01028-002-003, 2009ZX01029-001-003, 2010ZX01036-001-002, 2012ZX01029-001-002-002),国家自然科学基金(61221062,61100163,61133004,61173001,61232009, 6122204, 61432016)和863计划(2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301)资助项目。
② 男,1990年生,博士生;研究方向:计算机系统结构,高级语言虚拟机;联系人,E-mail: liaobin@ict.ac.cn
(收稿日期:2014-10-11)

过程中可能会产生大量的访存延迟;其次,当虚拟机触发 GC 时,在 GC 过程中 GC 线程每次读写的活跃对象可能位于远程内存;最后,对象的随机分配与回收将导致应用程序在同一 NUMA 结构机器多次运行的时间与性能存在差异。因此,需要针对 NUMA 结构设计特定的 GC 算法。

针对新对象分配,Oracle java 虚拟机 Hotspot 的并行 GC 算法 Parallel Scavenge[4] 已根据 NUMA 结构特点做出改进。官方测试结果表明,Parallel Scavenge 算法针对 NUMA 结构的优化能使应用程序性能提升 30%[5]。但是在非活跃对象回收过程中,NUMA 结构因大量的远程内存访问依然制约着 GC 的性能。在当前互联网及大数据盛行的时代,应用程序对 Web 服务器响应实时性的要求越来越高。虽然一些并发 GC 算法例如 Concurrent Mark-Sweep (CMS)[6]、Garbage First(G1)[7] 等也可以满足实时性要求,但是并发 GC 算法都是以牺牲应用程序吞吐量为代价来满足实时性的要求。如何在保证应用程序吞吐量的前提下缩短 GC 时间成为当前学术界以及工业界普遍关注的话题。本文对 GC 过程的活跃对象扫描、动态负载均衡以及活跃对象复制三个阶段进行了研究,并根据 NUMA 结构的特点,提出了一种基于 NUMA 结构的高效、实时、稳定的 GC 算法。

1 基于 NUMA 结构的对象分配机制

1.1 分代 GC 策略

分代 GC 策略[8,9] 自从被提出后就广泛应用于各大商业 Java 虚拟机中。由面向对象语言实现的应用程序在执行过程中会大量创建对象,然而在许多应用程序中,大部分新创建的对象很快就会变成非活跃对象[9,10]。

基于新生对象很快变为非活跃对象的特性,分代 GC 策略将堆空间分为不同的区域。如图 1 所示,堆空间被分为年轻代和老年代,应用线程在新生代分配对象,如果发生多次 GC 后,某对象仍然存活,则将该存活对象移动至老年代。新生代自身又分为 Eden 区、From 区以及 To 区,其中 Eden 区为对象分配区,而 From 区和 To 区用于保存暂时存活的对象。新对象在 Eden 区中被创建,当 Eden 区大小不足时,虚拟机触发 GC。GC 过程中,From 区以及 Eden 区中的存活对象被复制到 To 区,接着 Eden 区和 From 区会被清空,并且 From 区会与 To 区进行交换。将新生代分区的主要目的是为了在提高堆空间利用率的同时采用高效的 Copying GC[4] 算法。

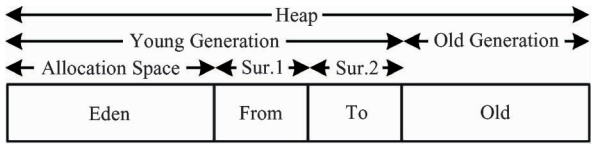

图 1 传统分代式 GC 的堆空间布局

分代 GC 策略根据对象存活的时间不同对新生代与老年代采用不同的 GC 算法,使堆的管理变得高效。但是,在 NUMA 结构中,传统的分代式堆空间结构并没有考虑 NUMA 结构访存非一致性的特点。如果新生对象没有被创建在合适的 NUMA 节点上,应用程序在后续执行过程中可能会发生大量的远程访存,在访存延时差距比较大的系统上,应用程序的性能会受到很大影响。为了解决这个问题,Oracle Java 虚拟机 Hotspot 将传统的分代式堆空间结构针对 NUMA 结构进行了改进。

1.2 堆空间布局的改进

如图 2 所示,针对 NUMA 结构,Hotspot 虚拟机将 Eden 区划分为与 NUMA 节点个数相同并且大小相等的内存空间。在堆空间初始化阶段,这些大小相同的内存区域被映射到不同节点的物理内存中。实验分析表明,新生对象在创建之后一般只会被创建线程访问,而 GC 之后仍然存活的对象,一般都是全局性数据,会被包括其他节点内的线程访问[5]。

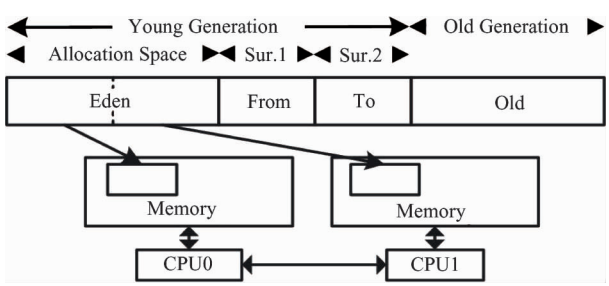

图 2 基于 NUMA 结构的分代式堆空间布局

由图 2 可知,Hotspot 在对传统分代式堆空间针对 NUMA 结构进行改进时,只改进了存储新分配对象的 Eden 区,而用于保存 GC 后存活对象的 From 区、To 区以及老年代都没有针对 NUMA 结构进行修改。

1.3　基于 NUMA 结构的对象分配机制

基于大部分新生对象只会被创建线程访问的特点,Hotspot 虚拟机在应用线程分配新对象时,仅在该应用线程所在节点的本地内存进行分配,如果本节点的内存不够,则直接触发 GC。

在 Godson-3 NUMA 结构服务器上,以 SPECjvm2008 为例,基于 NUMA 结构的对象分配机制带来的性能提升如图 3 所示。

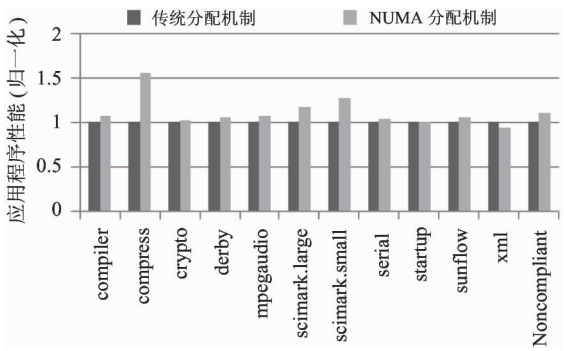

图 3　基于 NUMA 结构的对象分配机制对 SPECjvm2008 性能的影响

根据图 3 所示的结果可知,基于 NUMA 结构的对象分配机制能够使应用程序平均性能提升 10%(最高 54%),其中计算密集型应用程序的性能提升最为明显。

虽然基于 NUMA 结构的对象分配机制能够使应用程序的性能提高,但这种对象分配机制的改进并不能降低 GC 的停顿时间。在 NUMA 结构上,GC 过程中存在扫描活跃对象以及复制活跃对象等一系列访存操作。如果 GC 过程不针对 NUMA 结构的特性进行改进,在 GC 过程中可能会发生大量的访问远程内存操作。通过对 SPECjvm2008 的分析可知,在 GC 过程中,远程内存访问次数占所有内存访问的比例约为 33.4% ~56.1%。由此可见,在 NUMA 结构上,几乎所有的应用程序在 GC 过程中都会发生大量的远程访存,直接导致 GC 的停顿时间变长,这不仅影响应用程序的性能,也使一些对实时性要求较高的事务性应用程序的实时性得不到保障。为了优化 GC 过程中远程访存带来的开销,本文根据 NUMA 结构的特性,提出了能够在 GC 过程中减少远程访存次数的 GC 算法。

2　基于 NUMA 结构的 GC 机制

2.1　堆空间布局的改进

为了实现本文提出的基于 NUMA 结构的 GC 机制,堆空间布局需要进一步修改,改进后的堆空间布局如图 4 所示。GC 过程在宏观上可以分为两个大的流程:扫描活跃对象和拷贝活跃对象。在扫描活跃对象阶段,GC 线程在 Eden 区以及 From 区扫描存活的对象;在拷贝活跃对象阶段,GC 线程将活跃对象拷贝到 To 区或者 Old 区。通过上述分析,GC 过程中对整个堆空间都存在访存操作,因此 NUMA 结构如果需要在整个 GC 过程中尽可能减少远程访存次数,需要将包括 Eden 区、From 区、To 区以及 Old 区都在内的整个堆空间布局进行修改。参考 Hotspot 中 Eden 区针对 NUMA 结构的修改,本文将传统分代堆空间中的其他区域做了类似改进。在虚拟机初始化阶段,分代 GC 策略中的不同区域都会被分成大小相同的块并分别映射到不同的 NUMA 节点的物理内存中。

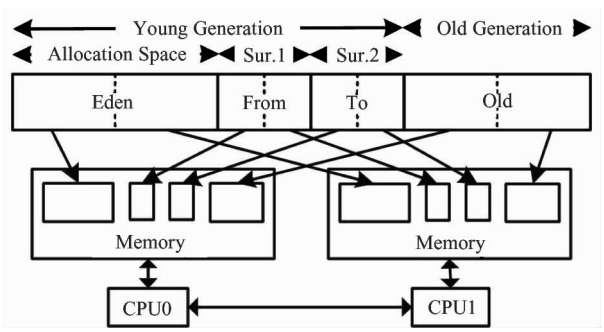

图 4　高效实时稳定 GC 算法的堆空间布局

2.2　优化的 GC 机制

当 Eden 区大小不足以继续分配新对象时,虚拟机会触发 GC,用于回收非活跃对象占用的内存。以 Parallel Scavenge 算法为例,GC 线程采用根对象搜索的方法扫描活跃对象。GC 线程从根对象集合出发,扫描对象引用关系图,能被 GC 线程扫描到的对

象称为活跃对象,活跃对象会被复制到堆中新的位置。新的 GC 机制就是从扫描活跃对象以及复制活跃对象两方面对 NUMA 结构的 GC 算法进行改进。

2.2.1 扫描活跃对象的优化

在扫描对象过程中,一个对象被扫描并确立为活跃对象有以下几种可能:

(1) GC 线程以应用线程栈(ThreadStack)中引用的对象作为根对象扫描到该对象;

(2) GC 线程以 CardTable[11] 中标记的对象作为根对象扫描到该对象;

(3) 并行 GC 中,当前 GC 线程从其他 GC 线程扫描任务队列中截取对象(StealTask)并继续进行扫描到该对象;

(4) GC 线程以方法区常量引用的对象、方法区静态属性引用的对象以及方法栈中 JNI 引用的对象等作为根对象扫描到该对象。

GC 线程在 GC 初始时刻随机选取根对象开始扫描过程,如果当前 GC 线程扫描任务完成,则随机截取其他仍未完成扫描任务的 GC 线程的对象继续进行扫描工作。无论是选取根对象还是截取对象,对象的随机选择性导致在 NUMA 结构上 GC 线程扫描到的活跃对象有可能与当前 GC 线程不在同一节点,由此将会有远程访存发生,进而影响 GC 的性能,因此在 NUMA 结构上减少扫描阶段的远程访存次数成为优化 NUMA 结构 GC 性能的关键。以 SPECjvm2008 为例,图 5 展示了前文所述四种扫描活跃对象途径的比例分布。由图可知,前三种途径扫描到的活跃对象占总活跃对象的 95% 以上,因此本文只针对前三种途径中的远程访存进行优化,具体优化方案如下:

(1) 应用线程栈引用对象的选取机制

在对象扫描初始时刻,如果随机选择应用线程栈中引用的对象作为初始根对象,则在后续扫描过程中将会发生远程访存的开销。在 NUMA 结构中,应用线程在分配新对象时,只在本地内存进行分配,因此 Eden 区中的对象与父对象一般都位于同一节点。对象的扫描过程是从根集合开始,因此根

对象的选取直接决定远程访存占所有对象访问的比例。如果选取的初始根对象与 GC 线程位于同一节点,后续扫描过程中的访存开销也相应得到减少。基于 NUMA 结构上应用线程仅在本地内存分配新对象的特点,因此 GC 线程在选取根对象时,只需要选择与当前 GC 线程位于同一节点的应用线程栈中的引用对象作为根对象即可确保后续扫描过程中的访存开销比随机选取根对象的访存开销要小。

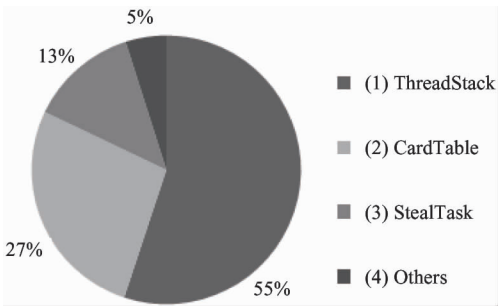

图 5 扫描活跃对象途径比例分布

(2) CardTable 中标记对象的选取机制

在对象扫描过程中,如果通过对象引用关系图扫描到某对象位于老年代,则扫描过程终止,并利用 CardTable 记录该老年代对象的位置。老年代对象的存活周期长,因而对象的引用关系也会比年轻代复杂很多,对象引用关系图也会比年轻代对象的引用关系图大很多。因此如果从老年代中对象继续扫描,则可能会扫描到老年代中大部分对象。虚拟机利用额外的数据结构记录扫描到老年代中对象的位置是为了避免扫描整个堆而使 GC 性能受到影响。在 NUMA 结构上,如果随机选取 CardTable 中记录的老年代的位置作为初始根对象,则在后续 GC 扫描活跃对象的过程中,远程访存次数可能会比较多。

在优化的 GC 机制中,GC 线程在选取 CardTable 中标记的对象作为初始根对象时,仅选择与当前 GC 线程位于同一节点的对象。基于 NUMA 结构年轻代中父子对象位于同一节点的特点,则要想满足从老年代对象引用的新生代对象与 GC 线程位于同一节点,就必须使年轻代对象与该老年代对象也位于同一节点。因此只要被选择的老年代中初始根对象在之前 GC 过程被复制到老年代时可以确保是从节点内的年轻代复制到老年代,则可以确保该对象与其年轻代中的子对象位于同一节点。由于初始根对象与其子对象位于同一节点,且选取的初始根对象

与 GC 线程位于同一节点,因此在扫描过程中远程访存次数要远少于随机选取初始根对象的远程访存次数。

(3) 动态负载均衡中截取对象所在任务队列的选取机制

在并行 GC 中,因为 GC 线程选取的初始根对象不同,因此对象引用关系图的大小也不同,进而导致不同 GC 线程完成扫描的时间也不相同。如果 GC 线程只扫描初始根集合的引用对象,则会导致 GC 过程中有空闲的 CPU 存在。为了使 GC 过程中 CPU 得到充分利用,虚拟机会在扫描对象过程中动态地对 GC 线程进行负载均衡。当某 GC 线程扫描完初始根集合引用的对象后,该 GC 线程会从其他仍在扫描对象的 GC 线程任务队列中随机截取对象继续进行扫描工作。在 NUMA 结构上,因为截取是随机发生的,因此在选择截取对象时不能确保 GC 线程访问截取对象及其子对象是本地访问。

在优化的 GC 机制中,当某 GC 线程初始根集合扫描结束后,该 GC 线程优先截取与其位于同一节点的其他 GC 线程的任务队列,如果本节点内其他 GC 线程的任务队列不为空,则从其任务队列截取对象,当且仅当本节点内所有 GC 线程任务队列均为空时,才选择从其他节点的 GC 线程任务队列截取对象。如前文所述,年轻代中父对象与子对象一般都位于同一节点,因此在截取对象时,只要保证截取的对象与当前 GC 线程位于同一节点,则截取对象的子对象也位于该节点内。改进的机制使截取后的大部分扫描操作都是本地内存访问,这将比随机截取对象节约了很多远程访存带来的开销。

经过以上几点改进,针对 NUMA 结构对 GC 机制进行改进之后,扫描活跃对象的执行流程如下:

(1)当 Eden 区的空闲内存无法继续分配新对象,虚拟机触发 GC。

(2)选取与当前 GC 线程位于同一节点的应用线程栈中的引用对象作为根对象开始扫描。

(3)随机选取方法区中静态变量以及 JNI 栈引用的对象作为根对象开始扫描。

(4)在 CardTable 标记的对象中选取与当前 GC 线程位于同一节点的对象作为根对象开始扫描。

(5)如果当前 GC 线程完成 GC 工作,则从本节点的其他 GC 线程任务队列截取对象继续开始扫描工作。

(6)本节点内所有 GC 线程任务队列为空,则从其他节点 GC 线程任务队列截取对象开始扫描工作。

(7)所有 GC 线程任务队列为空,GC 完成。

图 6 详细描述了优化 GC 算法的活跃对象扫描流程。在 Copying GC 算法中,活跃对象被扫描之后会直接被复制到新的位置,因此活跃对象拷贝位置的选择也直接影响 GC 的性能。

2.2.2　复制活跃对象的优化

通过前文所述对扫描活跃对象期间的优化,在 NUMA 结构上,扫描活跃对象的开销大大降低,但是扫描只是读取对象的相关信息。在扫描到活跃对象之后,需要将活跃对象复制到新的位置,即需要对对象进行写操作,因此拷贝活跃对象的时间比扫描活跃对象的时间要长。既然是写内存操作,在 NUMA 结构上就存在远程访存的问题。如果对象拷贝的位置选取不恰当,就会造成大量的远程访存,进而对 GC 时间以及应用性能产生影响。

在 GC 过程中,活跃对象只可能被复制到两个区域,即 To 区和老年代。在本文中,To 区以及老年代都针对 NUMA 结构进行改进。在拷贝活跃对象时,对象只拷贝到与当前 GC 线程位于同一节点的 To 区或者老年代中。因为 To 区以及老年代中的对象都是 GC 之后仍然存活的对象,因此该对象不像 Eden 区中对象一样大部分只会被本节点的应用线程访问,GC 之后仍然存活的对象不仅会被本节点应用线程访问,也会被其他节点应用线程访问。因此,GC 后存活对象的位置不会影响 GC 完成后应用线程对该对象的访存性能,但是在 GC 过程中,对象的复制过程却影响着 GC 的性能。在 NUMA 结构中,GC 线程在扫描到活跃对象之后,将该对象复制到本地内存中的 To 区和老年代中。

根据前面所述的优化机制,GC 线程可以使扫描到的活跃对象尽可能位于本地内存中,而基于 NUMA 结构复制活跃对象的优化又可以确保活跃对象的复制位置也位于本地内存中,因此新的GC机

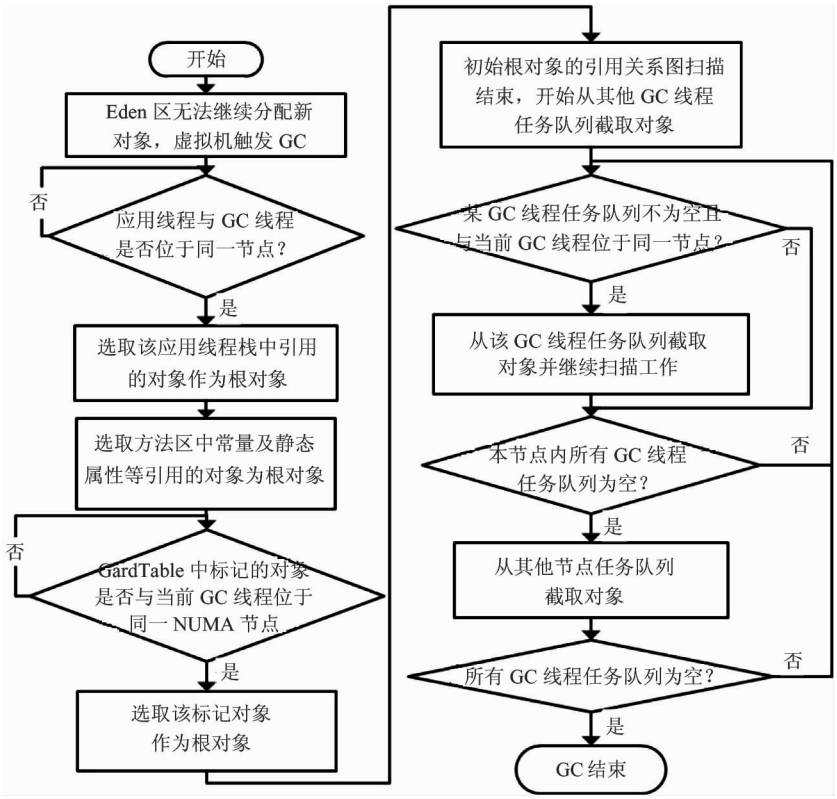

图 6 优化 GC 算法活跃对象扫描机制

制可以使 NUMA 结构上的远程访存次数大大减少，NUMA 结构上 GC 的停顿时间明显减少，应用程序的性能也可以得到提升。

图 7 描述了优化 GC 算法中复制活跃对象的机制。

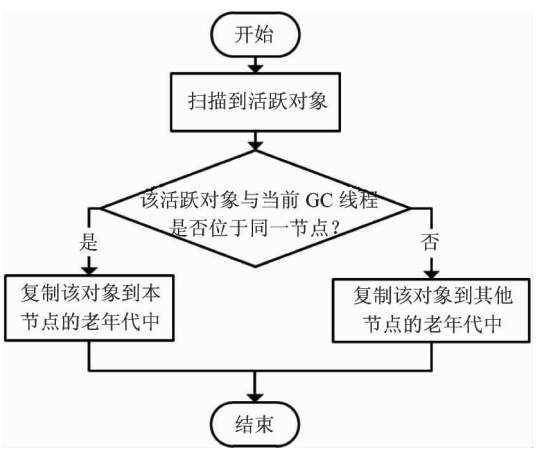

图 7 优化 GC 算法活跃对象拷贝机制

优化的 GC 算法针对 NUMA 结构修改了活跃对象的扫描机制和拷贝机制，新的 GC 算法可以避免 GC 过程中大量的远程内存访问。

3 实验评估

3.1 实验平台

本文提出的算法对于所有 NUMA 平台都具有通用性，算法的最终效果取决于 NUMA 平台中本地内存与远程内存的访存差异。在 Godson-3 NUMA 结构中，访问远程内存比访问本地内存慢 47%。

本章基于 Godson-3 处理器的 NUMA 结构，以 Hotspot 虚拟机中 Parallel Scavenge 算法为实验平台，对优化的 GC 机制进行了评估。Godson-3 是主频为 900MHz 的 4 核处理器，采用 64 位 MIPS 体系结构，每核配备 64K 一级指令 cache 和 64K 一级数据 cache，多核共享 4M 二级指令 cache；基于 Godson-3 的 NUMA 平台由两个 Godson-3 处理器组成，本地内存与远程内存的访存延迟差为 47%；Hotspot 虚拟机为开源 OpenJDK 中所使用的 Java 虚拟机，是目前使用最广泛的 Java 虚拟机；Parallel Scavenge 算法也是目前商用 Java 虚拟机中使用最广泛的 GC 算法之一。

3.2　实验结果与分析

基于 SPECjvm2008[12]测试程序,分别统计优化 GC 机制对 GC 过程中远程访存次数的影响、对应用程序总 GC 时间的影响、对应用程序性能以及稳定性的影响。

图 8 展示了优化前后 GC 机制对 GC 过程远程访存的影响。选择 SPECjvm2008 测试程序,分别展示了 SPECjvm2008 中各项在 GC 过程中访问远程内存次数占所有内存访问的比例。通过比较可知,改进后的 GC 机制在 GC 过程中的远程访存比例要比之前降低 3.14% ~46.03%,远程访存次数的减少直接导致 GC 停顿时间也相应减少。

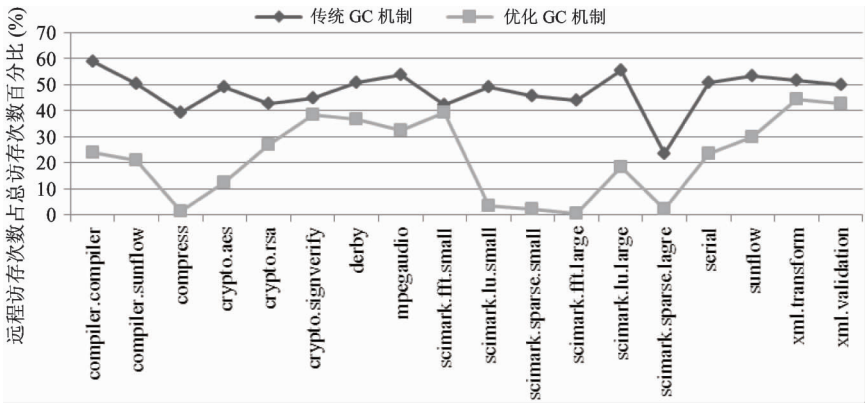

图 8　优化 GC 机制对应用程序远程访存比例的影响

图 9 选择 SPECjvm2008 作为测试程序,展现了优化前后各项总 GC 时间的变化。应用程序的对象引用关系图大小以及结构的不同导致各项的 GC 时间差异较大,因此对优化前 GC 时间归一化处理。实验结果表明,优化使 GC 总时间减少 4.1% ~41.58%。

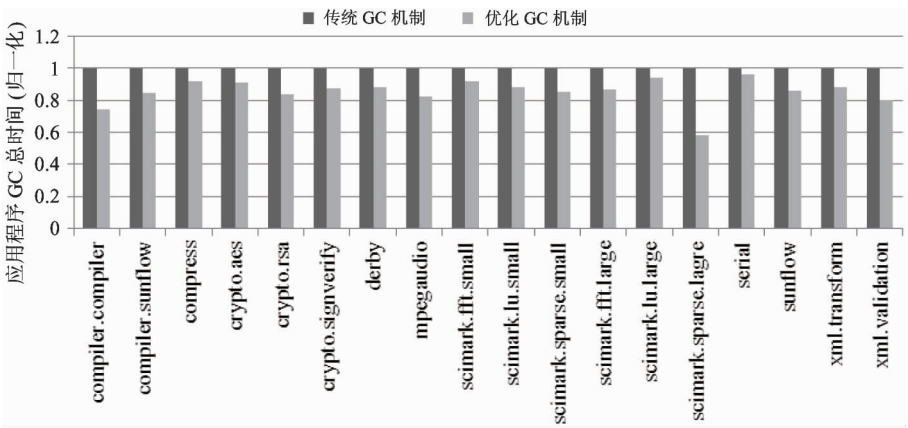

图 9　优化 GC 机制对应用程序总 GC 时间的影响

由于 GC 时间的减少,应用程序的总执行时间也会相应减少,因此应用的性能也会得到提升。优化 GC 机制对应用程序性能的影响如图 10 所示。数据表明,在没有发生 GC 的项目上,应用程序的性能没有下降,说明优化的 GC 机制对应用程序的性能没有负面影响;在有 GC 发生的项目上,优化 GC 机制可以使 SPECjvm2008 性能最高提升 17.8%,平均性能提升 4.68%。

因为 NUMA 结构访存不一致性的特点,导致应用程序多次运行发生 GC 时间不稳定的现象,进而对应用程序的性能稳定性也会产生一定影响。本文以SPECjvm2008中compiler. compiler项为测试目

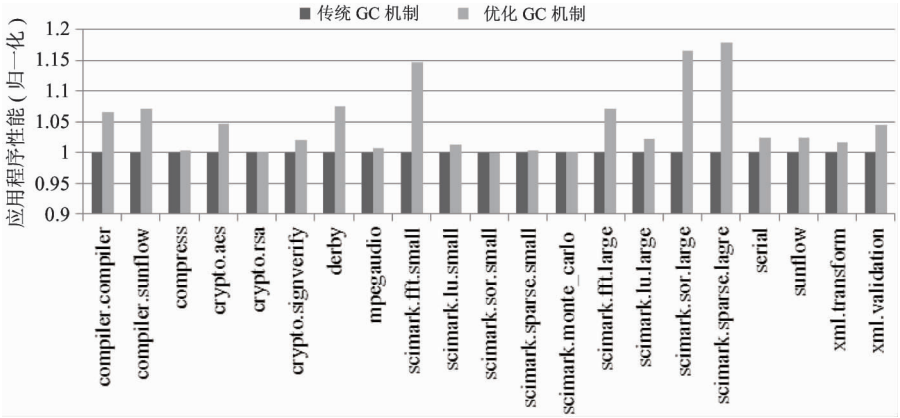

图 10 优化 GC 机制对应用程序性能的影响

标,分别利用优化前后的 GC 算法对该项目测试 10 次,测试结果如图 11 所示。结果表明,优化后的 GC 机制可以使应用程序的性能稳定性得到很大提升。如果以方差度量稳定性,优化的 GC 机制可以使应用程序的性能稳定性提升 76.2% 。

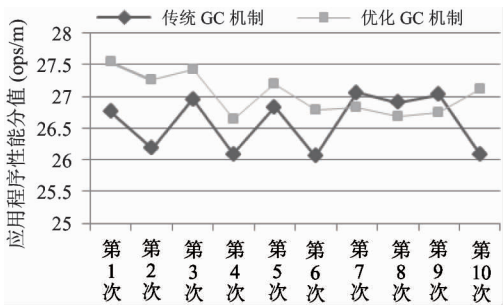

图 11 优化 GC 机制对应用程序性能稳定性的影响

4 相关工作

GC 是一个热门研究领域,GC 最早应用于 Lisp 语言中。自 20 世纪 70 年代 Steele[13] 等发现在大型 Lisp 程序中 GC 时间占程序运行时间 40% 后,学术界和工业界开始重视 GC 的性能问题,并且在后续的时间里提出了大量新的 GC 算法。以 Hotspot 虚拟机为例,虚拟机中自动内存管理机制有多种可供选择的 GC 算法。有简单但性能较低的串行 GC 算法[4],也有复杂但性能高的并行 GC 算法[14],还有为了满足应用程序实时性的并发 GC 算法[6,7]。

多核硬件的广泛使用加速了多线程 GC 算法的发展,Lokesh[15,16] 等提出的多线程无锁的 GC 机制提高了多线程 GC 的性能,但是这些性能的优化只在 16 线程以上的硬件中有好的效果。对于本文中使用的 8 到 16 线程的硬件没有任何效果,甚至会因为额外开销而导致 GC 性能降低。

NUMA 结构的出现也使 GC 算法的发展多了一个新的方向。Sun Microsystems[5] 将传统的堆空间布局进行改进,通过修改年轻代中 Eden 区的布局改进 NUMA 结构中新对象的分配机制,从而提高了 NUMA 结构上应用程序的性能; Mustafa[17] 和 Takeshi[3] 等将堆空间布局进一步改进,通过对新生代中其他区域以及老年代的修改来改进活跃对象的复制机制,使 GC 后应用线程的访存性能得到进一步提高,但是对于大部分应用程序而言,GC 之后仍然存活的对象会被大部分应用线程同时访问,因此文献[3,17]提出的改进需要基于特定的应用场景进行取舍。

文献[5]虽然提高了 NUMA 结构上应用程序的性能,但是 NUMA 结构上 GC 的暂停时间却仍然没有变化,本文提出的优化方法能够减少 GC 的暂停时间,并且进一步提高应用程序的性能和稳定性。

5 结 论

在 NUMA 结构中,自动内存管理机制的性能会受到 NUMA 结构非一致性访存因素的影响。自动内存管理机制包括新分配对象的位置管理以及非活跃对象的回收,当前内存管理机制已经针对 NUMA 结构在新分配对象的位置管理中做出改进,并且取

得良好效果,但是,在非活跃对象回收时,NUMA 结构的远程访问开销依然会影响内存管理的性能以及应用程序的性能稳定性。本文研究了这一问题,并针对性地提出了 NUMA 结构上非活跃对象回收的改进机制,再次提高了 NUMA 结构上自动内存管理的性能。本文将非活跃对象的回收分为扫描活跃对象以及复制活跃对象两个大的阶段,并采取新的机制使 GC 过程中远程访存次数大大降低。优化的 GC 机制对所有 NUMA 结构都具有通用性,且本地内存与远程内存访存差异越大的 NUMA 结构,该算法的效果越明显。新的 GC 机制能够应用于并行、并发等所有多线程 GC 算法,使 NUMA 结构上的应用程序的性能、稳定性以及 GC 的性能得到全面提升。

参考文献

[1] Manchanda N. Non-Uniform Memory Access. http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf: New York University, 2010

[2] Wang H D. 龙芯 3A 处理器用户手册. http://www.loongson.cn/uploadfile/cpumanual/Loongson3A1000_processor_user_manual_P1_V1.14.pdf: Loo-ngson Technology, 2014

[3] Ogasawara T. NUMA-aware memory manager with dominant-thread-based copying GC. In: Object Oriented Programming, Systems, Languages & Applications, Orlando, USA, 2009. 377-390

[4] Java SE Documentation. Memory Management in the Java HotSpot™ Virtual Machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf:Sun Microsystems, 2006

[5] Java SE Documentation. Java HotSpot? Virtual Machine Performance Enhancements. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#numa:Sun Microsystems, 20-09

[6] Printezis T, Detlefs D. A generational mostly concurrent garbage collector. In: Proceedings of the 2nd international Symposium on Memory Management, Minneapolis, Minnesota, USA, 2000. 143-154

[7] Detlefs D, Flood C, Heller S. Printezis T. Garbage first garbage collection. In: Proceedings of the 4th international Symposium on Memory Management, Vancouver, Canada, 2004. 37-48

[8] Lieberman H, Hewitt C. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 1983, 26(6): 419-429

[9] Ungar D. Generation scavenging: A nondisruptive high performance storage reclamation algorithm. *ACM Sigplan Notices*, 1984, 19(5): 157-167

[10] Shuf Y, Gupta M, Bordawekar R, et al. Exploiting prolific types for memory management and optimizations. *ACM SIGPLAN Notices*, 2002, 37(1): 295-306

[11] Alain A, Elliot K. K, Erez P, et al. Combining Card Marking with Remembered Sets: How to Save Scanning Time. In: Proceedings of the 1st international Symposium on Memory Management, USA, 1998, 10-19

[12] SPEC Documentation. SPECjvm2008. http://www.spec.org/jvm2008/index.html: Standard Performance Evaluation Corporation,2008

[13] Steele Jr G L. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 1975, 18(9): 495-508

[14] Flood, Christine H, et al. Parallel Garbage Colle- ction for Shared Memory Multiprocessors. In: Java Virtual Machine Research and Technology Symposium. Monterey, California, USA, 2001, 2-10

[15] Gidra L, Thomas G, Sopena J, et al. Assessing the scalability of garbage collectors on many cores. In: Proceedings of the 6th Workshop on Programming Languages and Operating Systems, Cascais, Portugal, 2011. 7-12

[16] Gidra L, Thomas G, Sopena J, et al. A study of the scalability of stop-the-world garbage collectors on multicores. *ACM SIGPLAN Notices*, 2013, 48(4): 229-240

[17] Tikir M M, Hollingsworth J K. NUMA-aware Java heaps for server applications. In: Parallel and Distributed Processing Symposium, Denver, USA, 2005. 108-117

A high-efficient, real-time and stable garbage collection algorithm for NUMA

Liao Bin[*,**,***,****], Fu Jie[*,**,***,****], Jin Guojie[**,***,****], Wang Yiguang[*,**,***,****],
Wang Lei[****,*****], Zhang Longbing[**,***], Wang Jian[**,***]

([*] Graduate University of Chinese Academy of Sciences, Beijing 100049)

([**] Key Laboratory of Computer System and Architecture, Chinese Academy of Sciences, Beijing 100190)

([***] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([****] Loongson Corporation, Beijing 100190)

([*****] University of Science and Technology of China, Hefei 230026)

Abstract

In order to solve a non-uniform memory access architecture (NUMA)'s performance degradation in garbage collection (GC) caused by a large amount of remote access during GC, each phase of the GC process was analyzed and studied, and a high-efficient, real-time and stable GC algorithm was proposed for NUMA. The algorithm improves the traditional generational GC mechanism's heap space based on the non-uniform memory access architecture first, and then, greatly decreases the number of remote access in the course of GC by controlling the selection of initial root objects during the live object scanning phase, the stealing task queue in the phase of dynamic load balance, and the object copying location during the procedure of copying live objects. The advanced GC algorithm can be applied to all NUMA platforms. The final results of the experiments on the Godson-3 NUMA platform show that the proposed algorithm can reduce the stop-the-world (STW) time during GC, and enhance the performance and stability of the application program. For the SpecJVM2008 benchmarks, the new algorithm averagely reduced the STW time by 14.6% (reduced the total time by 4.1% to 41.58%), averagely increased the performance of the application program by 4.68% (the ceiling value was 17.8%), and improved its stability by 76.2%.

Key words: non-uniform memory access architecture (NUMA), garbage collection (GC), generational GC, live object, root object, dynamic load balance