软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006502]

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

面向大数据处理框架的 JVM 优化技术综述*

汪钇丞[1,2], 曾鸿斌[1,2], 许利杰[1,2,3], 王 伟[1,2,3], 魏 峻[1,2,3], 黄 涛[1,2]

[1](计算机科学国家重点实验室(中国科学院 软件研究所),北京 100190)

[2](中国科学院大学,北京 100190)

[3](中科南京软件技术研究院,江苏 南京 210000)

通讯作者: 许利杰,王伟, E-mail: {xulijie, wangwei}@otcaix.iscas.ac.cn

摘 要: 当前,以 Hadoop、Spark 为代表的大数据处理框架,已经在学术界和工业界被广泛应用于大规模数据的处理和分析.这些大数据处理框架采用分布式架构,使用 Java、Scala 等面向对象语言编写,在集群节点上以 Java 虚拟机(JVM)为运行时环境执行计算任务,因此依赖 JVM 的自动内存管理机制来分配和回收数据对象.然而,当前的 JVM 并不是针对大数据处理框架的计算特征设计的,在实际运行大数据应用时经常出现垃圾回收(GC)时间长、数据对象序列化和反序列化开销大等问题.在一些大数据场景下,JVM 的垃圾回收耗时甚至超过应用整体运行时间的 50%,已经成为大数据处理框架的性能瓶颈和优化热点.本文对近年来相关领域的研究成果进行了系统性综述: (1)总结了大数据应用在 JVM 中运行时性能下降的原因; (2)总结了现有面向大数据处理框架的 JVM 优化技术,对相关优化技术进行了层次划分,并分析比较了各种方法的优化效果、适用范围、使用负担等优缺点; (3)探讨了 JVM 未来的优化方向,有助于进一步提升大数据处理框架的性能.

关键词: 大数据系统;Java 虚拟机;分布式系统;自动内存管理

中图法分类号: TP311

Survey on JVM Optimization for Big Data Processing Frameworks

WANG Yi-Cheng[1,2], ZENG Hong-Bin[1,2], XU Li-Jie[1,2,3], WANG Wei[1,2,3], WEI Jun[1,2,3], HUANG Tao[1,2]

[1](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100190, China)

[3](Nanjing Institute of Software Technology, Nanjing 210000, China)

Abstract: Nowadays, the big data processing frameworks such as Hadoop and Spark, have been widely used for data processing and analysis in industry and academia. These big data processing frameworks adopt the distributed architecture, generally developed in object-oriented languages like Java, Scala, etc. These frameworks take Java Virtual Machine (JVM) as the runtime environment on cluster nodes to execute computing tasks, i.e., relying on JVM's automatic memory management mechanism to allocate and reclaim data objects. However, current JVMs are not designed for the big data processing frameworks, leading to many problems such as long garbage collection (GC) time and high cost of data serialization and deserialization. As reported by users and researchers, GC time can take even more than 50% of the overall application execution time in some cases. Therefore, JVM memory management problem has become the performance bottleneck of the big data processing frameworks. This paper makes a systematic review of the recent JVM optimization research work for big data processing frameworks. Our contributions include (1) We summarize the root causes of the performance degradation of big data applications when executed in JVM; (2) We summarize the existing JVM optimization techniques for big data processing frameworks. We also classify these methods into categories, compare and analyze the advantages and disadvantages of each, including the method's optimization effects, application scopes and burdens on users; (3) We finally propose some future JVM optimization directions, which will help the performance improvement of big data processing frameworks.

Key words: big data system; Java virtual machine; distributed system; automatic memory management

* 基金项目: 国家重点研发计划(2017YFB1001804); 国家自然科学基金(61802377); 中国科学院青年创新促进会
收稿时间: 2021-01-18; 修改时间: 2021-04-29, 2021-07-30; 采用时间: 2021-09-29; jos 在线出版时间: 2021-11-24

随着互联网技术的快速发展和普及,互联网产生的数据量也呈现出爆炸式增长的趋势.为了满足大规模数据处理的需求,工业界和学术界开发出了多种分布式的编程模型及处理框架,如 MapReduce[1], Dryad[2]等.这类编程模型采用分治的思想,将大的数据处理作业(job)拆分为多个小的计算任务(task),分配调度到分布式集群的不同节点上并行执行.当前主流的大数据分布式处理框架例如 Hadoop[3]、Spark[4]、Flink[5]等,基本继承了这类编程模型,并选择了 JVM 作为执行计算任务的具体运行时环境,使计算任务以线程或是进程的方式在 JVM 中执行,依赖 JVM 实现内存对象的分配和回收.

这些大数据处理框架采用 JVM 运行的原因在于:(1)开发者可以利用 JVM“一次编写,到处运行”,摆脱分布式集群中不同操作系统和硬件处理框架带来的束缚,更加专注于代码逻辑;(2)JVM 提供了便捷的自动内存管理机制,可以自动为新创建的对象进行内存分配,以及对不再使用的对象进行回收,减轻了开发者的编程负担;(3)Java 作为一个成熟的面向对象编程语言,拥有丰富的社区资源,能够实现快速开发.

然而,在实际使用大数据处理框架的过程中,学术界和工业界发现了一系列 JVM 相关的性能瓶颈.其中最主要的性能瓶颈来自 JVM 长时间,高频次的 GC.比如,在一些大数据应用中,GC 暂停时间在大数据应用的总执行时间中占比可达 50%[6].另外,分布式节点间的网络传输需要 JVM 序列化和反序列化数据对象,用时占比可达 30%[7];JVM 冷启动的预热用时可达 10-30s[8];JVM 内存溢出错误(Out of Memory,OOM)会导致计算任务执行失败[9].这些问题严重影响到了大数据应用的执行效率,使大数据处理框架难以实现低延迟、高吞吐率的目标.回退到更基础的非托管运行时环境可以部分避免上述问题,但目前主流的大数据处理框架都是基于 JVM,而在大数据框架的生态圈中,一个框架的功能往往建立在之前框架功能的基础之上,这使得选择 JVM 已经成为了一个长期的趋势.在这样的趋势之下,对 JVM 大数据环境适应性的优化至关重要,已经成为近年来的研究热点方向.

当前,一些综述研究[10,11]介绍了大数据处理框架的内存使用技术,但并没有关注运行时环境层面的内存管理问题.也有综述研究[12]介绍了大数据处理框架通过 GC 算法管理内存的过程,从可拓展性的角度分析了 GC 算法在管理大规模内存时存在的问题,并对已有优化技术根据宏观优化目标进行了分类.还有综述研究[13]从对象管理的角度,分析了大数据处理框架下的内存管理问题,并对各类解决方案的特点进行了总结和比较.鉴于发表年份稍早,这些综述研究[12,13]都未能包含近几年新出现的优化工作.本文结合大数据处理框架的结构层次和计算特征,深度分析了 JVM 在分布式大数据环境下的性能瓶颈,全面地整理和归纳了各个方向上的优化方法,主要关注三个核心研究问题:(1)JVM 在大数据处理框架中性能低下的原因是什么? (2)可以从哪些层次和方面优化大数据处理框架中 JVM 的性能,现有的研究工作具体采用了怎样的优化方法?这些优化方法有哪些优缺点? (3)未来可以从哪些方向继续优化 JVM,提高其对大数据处理框架的适应性?

针对这三个核心研究问题,本文对相关领域近年来的工作进行了广泛调研,利用文献数据库和搜索引擎,在国内外主流会议和期刊收录的论文当中进行了细致检索,收集并阅读了上百篇论文,最终筛选出了 40 余篇高度相关的高质量文献进行综述研究.本文的主要贡献在于:(1)结合大数据应用的特点和模式,总结了 JVM 相关性能问题产生的原因,如大数据应用内存使用量大、对象生命周期复杂、JVM 与上层框架存在隔阂;(2)将面向大数据处理框架的 JVM 优化技术系统地归纳为五个层面,包括大数据框架的内存管理、JVM 中数据对象的存储方法、JVM 的 GC 算法、JVM 集群的协同、JVM 在新型硬件架构中的应用等,分析了各个层面优化技术的目标问题、解决方法和局限性,并对同一层面中的优化方法从实现过程、适用范围、开发者负担等方面进行了比较;(3)提出进一步提升 JVM 在大数据处理框架下性能表现的优化方向,可以作为后续相关研究工作的参考.

本文第 1 节介绍了大数据处理框架、JVM、GC 算法的工作流程及关系.第 2 节总结了 JVM 在大数据环境中存在的性能问题并分析了问题原因.第 3 节概述了面向大数据处理框架的 JVM 优化技术方向.第 4 节介绍了大数据框架内存管理的优化技术.第 5 节介绍了执行器集群协同的优化技术.第 6 节介绍了 JVM 中数据对象存储方法的优化技术.第 7 节介绍了 JVM 中 GC 算法的优化技术.第 8 节介绍了新型硬件架构下 JVM 的优化技术.第 9 节对现有优化工作进行了总结,并探讨了未来可能的研究方向.

1 相关背景概述

本节简述大数据处理框架以及 JVM 相关概念,包括大数据处理框架如何将大数据应用转化为可并行执行的计算任务,JVM 执行任务代码的流程,以及 JVM 的垃圾回收机制和相关的 GC 算法.

1.1 大数据处理框架的工作流程

大数据处理框架为用户提供了简单且具有扩展性的编程模型,能够将大数据应用转化为可并行运行在分布式集群上的计算任务,实现大数据的高效处理[14].当前流行的大数据处理框架,如 Hadoop, Spark 等,都是以 MapReduce 编程模型为基础,具体的流程如图 1 所示,可以简单表示为:

Map 阶段: map<K1,V1> => list<K2,V2>

混洗(Shuffle)阶段: list<K2,V2> => <K2,list(V2)>

Reduce 阶段: reduce<K2,list(V2)> => list<K3,V3>

Hadoop MapReduce 基本实现了标准的 MapReduce 编程模型,而 Dryad 以有向无环图(DAG)形式的数据流取代了 MapReduce 固定的两阶段数据流,Spark 针对 MapReduce 和 Dryad 框架的一些问题,提出了基于内存,适用迭代计算的处理框架:首先允许用户将可重用的数据缓存到内存中,同时利用内存进行中间数据的聚合,缩短数据处理和 I/O 的时间;另外将输入输出数据,中间数据抽象为统一的数据结构,命名为弹性分布式数据集(RDD),并在此数据结构上构建了一系列通用的数据操作,实现复杂的数据处理流程.

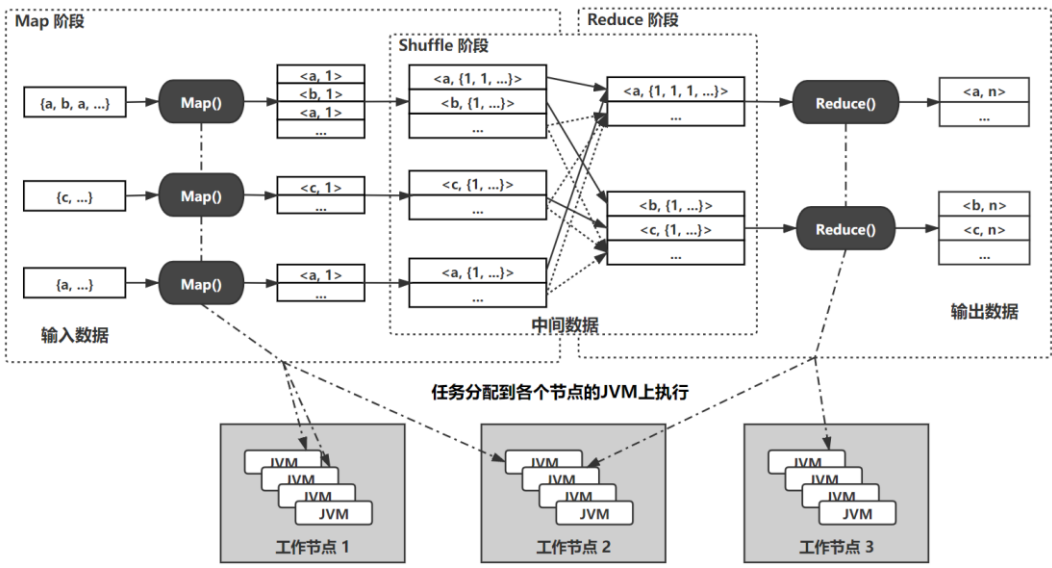


图 1 MapReduce 编程模型处理流程示例

大数据应用通常以<输入数据,用户代码,配置参数>的形式提交给大数据处理框架.大数据处理框架得到用户输入,生成一个驱动器(Driver)程序,将大数据应用分解为一个或多个作业(jobs).如图 1 所示,一个作业通常会被划分为多个数据处理阶段(stage),每个执行阶段包含有多个计算任务(task).计算任务经由任务调度系统,分配到集群的各个工作节点上,以 JVM 进程或者线程的方式运行.如 Hadoop 为每个计算任务启动一个执行器(Executor) JVM 进行运行,Spark 为每个任务启动一个 JVM 线程,该线程运行在执行器 JVM 中.来源于分布式存储系统的输入数据经过转换,以数据对象的形式存储在执行器 JVM 的内存当中,依赖 JVM 的自动内存管理机制实现内存的申请和回收.

1.2 JVM的运行过程

JVM 是在各类计算机环境和各种操作系统上构建的一种统一的运行环境.JVM 为应用程序隐藏了对底

层机器和操作系统的操作,使得 Java、Scala 等编程语言代码在编译成 Java 字节码后,能够“一次编写,到处运行”.Java 字节码在加载进 JVM 之后,经过类加载机制的校验,解析,初始化等步骤,成为可以使用的 Java 类型,通过 Java 解释器解释执行.当 JVM 发现部分代码运行频繁,就会将这部分热点代码即时编译(JIT)为本地机器码,提高代码后续的执行效率.类加载和热点代码的即时编译过程通常被认为是 JVM 初始化的冷启动消耗.

堆内存通常是 JVM 的运行时数据内存中最大的一块,用于存储所有的对象实例以及数组.JVM 的自动内存管理机制负责对堆内存进行维护,创建新的对象,保证存活的对象(还在被使用的对象)保留在内存当中,以及通过 GC 算法清理掉不再使用的对象.而在运行时数据内存之外,JVM 也可以在直接内存(也被称为堆外内存)进行对象分配和引用,而这部分内存则不由自动内存管理机制负责.

1.3 JVM垃圾回收机制

OracleJDK/OpenJDK 所使用的 HotSpot VM 是目前应用最广泛的 JVM,它所提供的 GC 算法都是通过**可达性分析**来判断对象的存活情况[15].**GC Roots 根对象集**是每一次可达性分析的起始节点,包括了一系列必须存活的对象.如图 2 所示,GC 线程从根对象出发,根据对象之间的引用关系,通过深度优先算法向下扫描.在 JVM 堆内存中能够扫描到的对象被认为是存活的对象,将在随后的阶段复制移动到新的内存位置;而没有被扫描到的对象被认为是不再被使用的死亡对象,最终被清理出内存.GC 流程的扫描和移动任务通常都被添加到 GC 任务队列中,由 GC 线程从任务队列中获取和执行.

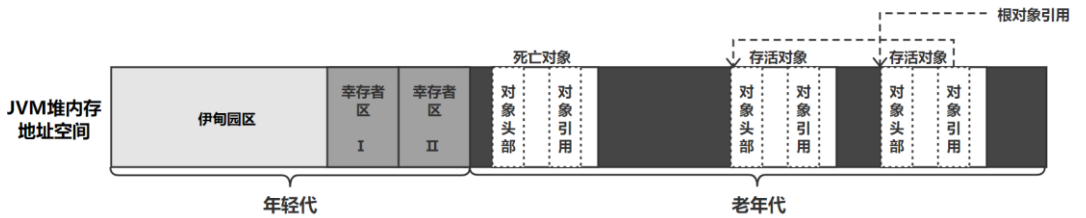


图 2 Parallel GC 算法下的 JVM 堆内存划分及可达性分析示例

HotSpot 中的 GC 算法都是基于**弱世代假说(**weak generational hypothesis)[16]:即绝大部分对象在创建后不久就不再被使用.因而现有的 GC 算法采用**分代收集**的思想,将堆内存划分为年轻代(Young Generation)和老年代(Old Generation),分别用于存储短寿命对象和长寿命对象.年轻代分为两部分:伊甸园区(Eden)和幸存者区(Survivor).对象最初被分配到年轻代的伊甸园区,伊甸园区在空间耗尽时会触发 Minor GC,将存活的对象复制到幸存者区中,并清空伊甸园区.当一个对象经历的 Minor GC 达到一定次数后依旧存活时,将被晋升复制到老年代中存储,老年代在空间使用量到达一定比例之后,触发 Major GC 对整个堆内存空间进行标记和整理.为了保证 GC 结果的一致性和安全性,所有的 GC 算法都存在部分处理阶段需要中断所有的工作线程,称为全局暂停(Stop-The-World,STW).

Java 7 和 Java 8 目前依旧是使用最广泛的 Hotspot 版本,因而它们的默认垃圾回收器 Parallel GC 也是最常被使用的传统 GC 算法.相比于更早的 Serial GC 算法,Parallel GC 的优势就在于允许多个 GC 线程同时从 GC 任务队列中获取任务并行执行.如图 2 所示,**Parallel GC 在堆内存中采取整块划分**,每一个年代的空间都是整体连续的.按照默认参数,JVM 堆的前 1/4 被划分给年轻代,后 3/4 划分给老年代,用户也可以通过参数自行指定.Parallel GC 默认开启动态自适应策略,根据运行时的情况对各年代的空间大小进行动态调整,以尽可能满足吞吐量和最大暂停时间的目标.

从 Java 9 开始到目前最新的 Java 17 版本,Garbage First(G1)[17]成为 Hotspot 的默认垃圾回收器.如图 3 所示,**G1 采用基于区域的堆内存划分方法**,JVM 堆内存被划分成等大小的区域,而每次 GC 只处理其中一部分的区域,避免一次处理过多的对象,解决了 Full GC 处理量过大,全局暂停时间过长的问题.G1 依旧采用年代划分法,部分区域会属于伊甸园区,部分区域属于幸存者区,部分区域属于老年代,但区域的归属并不固定,提供了更高

的灵活性.当一个老年代区域的存活对象比例低于一定阈值时,该区域会被加入待 GC 区域集,待 GC 区域集在数量达到一定比例时会触发混合 GC(Mixed GC),收集待 GC 区域集中的老年代区域和所有年轻代区域.G1 通过记忆集(Remember Set)快速处理区域之间的引用关系,通过堆快照(Snapshot)和写屏障(Write Barrier),使得 G1 可以在部分处理阶段实现工作线程和 GC 线程并发执行,达到降低全局暂停时间的目的.

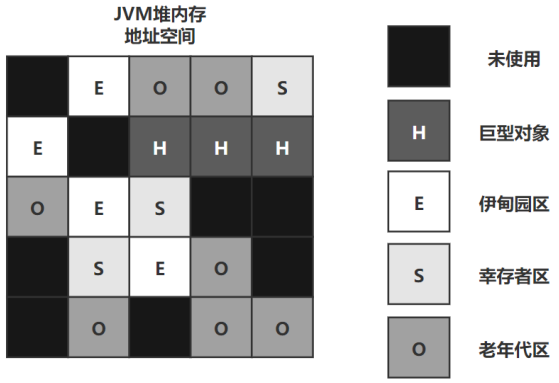


图 3 G1 GC 算法下的 JVM 堆内存划分

Parallel 和 G1 两种 GC 算法尽管在实现方法上存在差异,但都是针对一般 Java 程序的对象使用特点设计的.大数据应用的对象使用有自身的特点,使用这传统 GC 算法进行内存管理时会产生严重的性能损失.下一章节将分析当前的 JVM 在应用于大数据处理框架时遇到的性能问题及原因.

2 JVM 在大数据环境中存在的问题及原因分析

大数据处理框架的性能分析和优化是一直以来的研究热点.JVM 作为大数据处理框架的运行时环境,它的执行效率直接影响着大数据处理框架的性能表现.本章介绍了 JVM 在大数据环境中存在的问题,并分析了问题的成因.

2.1 JVM在大数据环境中的性能问题

工业界的实践和学术界的评测工作发现,**GC机制是对JVM执行效率影响最大的因素.具体表现包括:**

(1) GC 的占用时间长,在一些大数据应用中,GC 时间可占应用总执行时间的 50%[6,18];

(2) GC 频率高,造成任务执行频繁暂停,大数据应用的吞吐率降低,响应延迟升高[19];

(3) GC 算法挤占了应用线程 CPU 资源,存在 GC 线程竞争时,大数据应用执行时间增加了 60%[9,20];

传统并行 GC 算法下的全局暂停时间较长,增加了大数据应用的延迟,容易引发任务掉队,与作业的尾延迟有着直接关系[20,21];传统并发 GC 算法下虽然全局暂停时间可以得到一定控制,但是以高 CPU 使用率和高暂停频率为代价,GC 线程与并发执行的应用线程竞争降低了任务吞吐率,也不能有效降低作业尾延迟[20].

除了 GC,JVM 的其他机制也会影响到大数据应用的整体表现:

(1) JVM 中的数据对象在分布式节点间传输需要序列化和反序列化,在大数据应用执行的用时占比可达 30%[7,22];

(2) JVM 冷启动时需要大量的类加载和代码即时编译工作,在大数据应用执行的用时占比可达 33%[8];

(3) JVM 运行和维护需要内存消耗,在内存紧张的环境下,可能因内存耗尽或内存碎片触发 OOM 错误[9,23];

这些 JVM 运行时代价严重干扰了大数据应用的执行.我们将上述问题产生的原因总结为三个方面:大数据处理框架下的 JVM 内存使用压力增大、JVM 内存管理模式与大数据应用的内存使用模式不匹配、JVM 与上层框架存在隔阂(gap).

2.2 原因1: 内存使用压力增大

在与普通的 Java 应用不同,大数据应用是“**内存密集**”的,JVM 的内存使用量更大.在大数据处理框架下,执行器 JVM 的内存使用压力具体来源于:

(1) 大数据应用数据计算和存储产生的大量内存消耗.大量数据在计算过程中需要同时被读取到内存当中,当前流行的大数据处理框架为了更进一步加快处理速度,将中间数据的聚合和可重用数据也缓存在内存当中,这决定了执行器 JVM 在执行大数据应用时将面对更大的内存使用量.

表 1 典型大数据应用的内存使用量和内存使用模式

应用名称	空间复杂度	内存使用模式	
GroupBy	reduceByKey(sum) : O(1)	Shuffle 阶段累积大量中间数据 (具有相同 key 的数据行)	
Join	join() : O(m + n)	Shuffle 阶段累积大量中间数据 (两张表具有相同 key 的数据行)	计算产生大量即刻输出的临时对象 (两张表的笛卡尔积结果)
SVM	reduce() : O(|x|)	部分巨型的数据对象 (高维度的参数向量)	大量长时间缓存数据 (输入的训练数据)
PageRank	join() : O(m + n)	每一次迭代中累积大量数据 (具有相同 key 的数据行)	大量长时间缓存数据 (输入的图数据)

注:m,n表示两个集合的元素数量,|x|表示数据的维度

(2) 数据在 JVM 堆内存当中以对象的形式存储需要的额外内存占用.对象在 JVM 当中的的数据结构包含了对象头以及对其他对象的引用,而数据本身在对象中的空间占比往往不超过一半[6,22].这些对象的“外壳”伴随着数据缓存在内存当中,也需要占用相当数量的空间.

JVM 的自动内存管理机制以对象为单位,数据和对象数量的增加意味着更大的内存管理负担,相应的 GC 机制会更频繁地触发更长时间的全局暂停.这个问题并不能够通过简单地增减内存大小解决.如果降低内存大小,GC 触发的频率则会增加,对象被扫描和移动的次数增加,应用程序的吞吐量相应降低.可用内存不足还会影响到大数据应用的正常的缓存和处理机制,甚至引发内存溢出;如果提升内存大小,单次 GC 则需要处理更多的数据对象,平均的暂停时间加长,应用程序的最大延迟相应增加.对于周期性标记扫描的 GC 算法而言,还会在最终触发 GC 之前消耗更多 CPU 时序进行不必要的标记.

2.3 原因2: 内存使用模式变化

大数据应用中数据在内存当中保留的时间周期与传统应用不尽相同.在 JVM 传统的应用场景下,堆内存中创建的绝大部分对象在产生之后不久就不再被使用,经典的 GC 算法正是基于这种内存使用模式,将堆内存进行粗粒度的年代划分,绝大部分转瞬即逝的对象会在针对年轻代的 Minor GC 当中很快被清理掉.而大数据应用产生的对象类型有两种,一种是由控制大数据处理框架运行逻辑的代码产生的,称为**控制路径对象**,它们的内存使用模式在通常情况下依旧符合弱世代假设;另一种是输入数据和计算中间数据在大数据处理框架中封装产生的,统称为**数据路径对象**[22,24].如表 1 中对典型大数据应用的分析,数据路径对象的内存使用模式要更加复杂,它们可能在内存中长时间累积或缓存,也可能在一个迭代轮次后被清理和输出.通常来说,数据路径对象的存活时间比控制路径对象更长,但各自的生命周期也不尽相同.尽管在代码数量上控制路径比数据路径多,但数据路径所创建的对象数量远超控制路径.传统 GC 算法并不能适应大数据环境下内存使用模式的这种变化,原因在于:

(1) 当前 GC 算法下,**长时间存活的数据路径对象最终都会晋升到老年代中**,它们在数次 Minor GC 当中幸存并最终晋升的过程中,需要在内存中多次移动.对象移动被认为是 GC 循环当中最耗时的部分[25,26],每一次移动都意味着内存读写,而内存位置的改变也需要对相关引用的指针进行更新.考虑到数据路径对象的庞大数量,整个晋升过程会消耗大量 CPU 时序,触发多次 GC 暂停.

(2) 数据路径对象在晋升到老年代之后,在作业执行的时间尺度上,短时间内也不会被回收.传统的 GC 算法不会考虑这些对象的存活时间,在涉及到老年代空间的 Major GC 或者 Mixed GC 之前还是会对整个堆内存空

间进行标记扫描,**这些标记扫描过程对于长时间存活的数据对象来说是不必要的**[9].当长时间存活对象占用老年代的比例过高,每次付出较大代价的 Major GC 就只能回收有限大小的空间,可能造成 Major GC 频繁触发,部分缓存数据被迫转移到磁盘,甚至出现 OOM 错误,浪费大量的 CPU 时序和全局暂停时间,影响到应用执行效率.

2.4 原因3: JVM与上层框架存在隔阂

大数据处理框架将计算任务分配调度到各个执行器JVM节点之后,并不会干预JVM的具体执行过程.每个执行器JVM独立运行,并不感知分布式集群中其他执行器JVM的执行情况,作业的整体进度,以及集群和节点的内存资源使用情况,只是根据自身的运行状态作出触发 GC,调整堆内存,进行代码即时编译等决策,而这些决策从历史和全局的角度上观察可能并不是最优的.

(1) JVM 不清楚任务执行产生的数据对象特征,例如对象数量、内存占用大小、生命周期等,只能根据弱世代假说,对所有对象进行一致的分配管理.由于大数据应用产生的大量对象长时间存活,JVM 的内存管理效率会受到严重影响,而这些对象本可以通过大数据框架对用户代码和数据流的全局静态分析进行甄别.

(2) 大数据处理框架不考虑 JVM 具体的内存管理机制,将所有 JVM 节点的内存当做连续的全局地址空间.但实际上 JVM 在 GC 算法下对堆内存空间采取分代管理,存在非连续区域,对象在内存中离散分布.另外大数据处理框架在采用全局地址空间的物理架构下,可能产生大量跨节点对象引用,给 JVM 的 GC 任务带来了远程内存访问的负担.

(3) 大数据处理框架下的 JVM 之间不清楚彼此的运行情况.如果大数据操作需要在各个 JVM 之间的同步,由于 JVM 独立进行 GC 决策,大数据操作的执行就可能被不同 JVM 的 GC 连续打断;另外,由于互相不感知,处于同一物理节点的 JVM 之间可能内存资源分配不合理,而大数据框架在相关问题上缺少统筹协调.

3 面向大数据处理框架的 JVM 优化技术概览

前述问题给 JVM 带来的性能下降不可避免,一些研究工作提出将运行时环境回退到非托管编程语言或者开发新的编程语言和运行时系统[27-29].然而用非托管编程语言编写的代码量更大,调试难度更高,将给框架开发者带来压力.而目前成熟的大数据处理框架也都是基于JVM,这使得选择JVM已经成为了一个长期的趋势,因而更多研究工作的目标依旧在于提升 JVM 对大数据处理框架的适应性.本节将从大数据处理框架的 3 个层次出发,概览面向大数据处理框架的 JVM 优化技术.

如图 4 所示,大数据处理框架可以划分为 3 个层次:用户层,大数据框架层,运行时环境层.按照 1.1 节描述的流程,用户层将应用代码和应用执行参数提交给大数据框架层,将应用资源需求参数提交给运行时环境层.大数据框架层由应用代码和应用执行参数,构建出应用的逻辑处理流程和分布式的物理执行计划,并根据集群节点的资源使用情况,将任务调度到各个机器节点的 JVM 上执行.JVM 所在的运行时环境层,根据应用资源需求参数,获得相应的CPU和内存等硬件资源,具体执行计算任务.输入数据和中间数据按照大数据处理框架和用户代码的定义,封装为对象的形式存储在 JVM 堆内存当中,而 JVM 的 GC 算法负责在对象不再被使用时,将对象从JVM的堆内存中清理回收.上述 3 个层次可以分别作为提高 JVM 对大数据处理框架适应性的切入点,主要存在 6 种优化的方向.

从用户层出发的主要优化手段是对应用执行参数和应用资源需求参数进行调优(Tuning),确定一个适应当前应用计算特征的配置,使大数据应用的执行时间更短[30].由于大数据框架和 JVM 的参数类型纷繁复杂,有效的人工调优需要开发者丰富的工程经验[31,32].为了降低调优工作的门槛,相关研究工作基于白盒(White-box)和黑盒(Black-box)两种调优模型,开发了一系列大数据处理框架和 JVM 的自动调优工具[33-39].基于白盒的调优工具主要通过建立假设分析(What-if)模型进行离线性能估计,根据性能估计结果进行参数调优[40,41].基于黑盒的调优工具主要根据真实在线测试的结果,对整个参数空间进行搜索和调优[42-44].尽管参数调优的方法可以取得不错的应用性能提升,但基于白盒方法的调优工具在假设分析模型的创建方面存在很大挑战,而基于黑盒方法的调优工具要获得足够好的结果需要耗费大量的时间进行测试,即使采用一些优化加速方法依旧需要小时级别的时间[45].另外,经过调优的参数通常只针对固定硬件环境下的一种大数据应用,并不能有效移植到其他应

用和硬件环境中,而不同的大数据应用在不同的处理阶段的计算特征和内存使用模式都存在差异,静态的参数无法在运行时同时满足各个阶段的需求.综上所述,在用户层的参数调优并不能从本质上改善 JVM 在大数据处理框架下的适应性问题,因而本文主要关注工作在大数据框架层和运行时环境层的优化技术,这些优化技术可以在一定程度上解决第 2 节描述的三个问题,如表 2 所示.

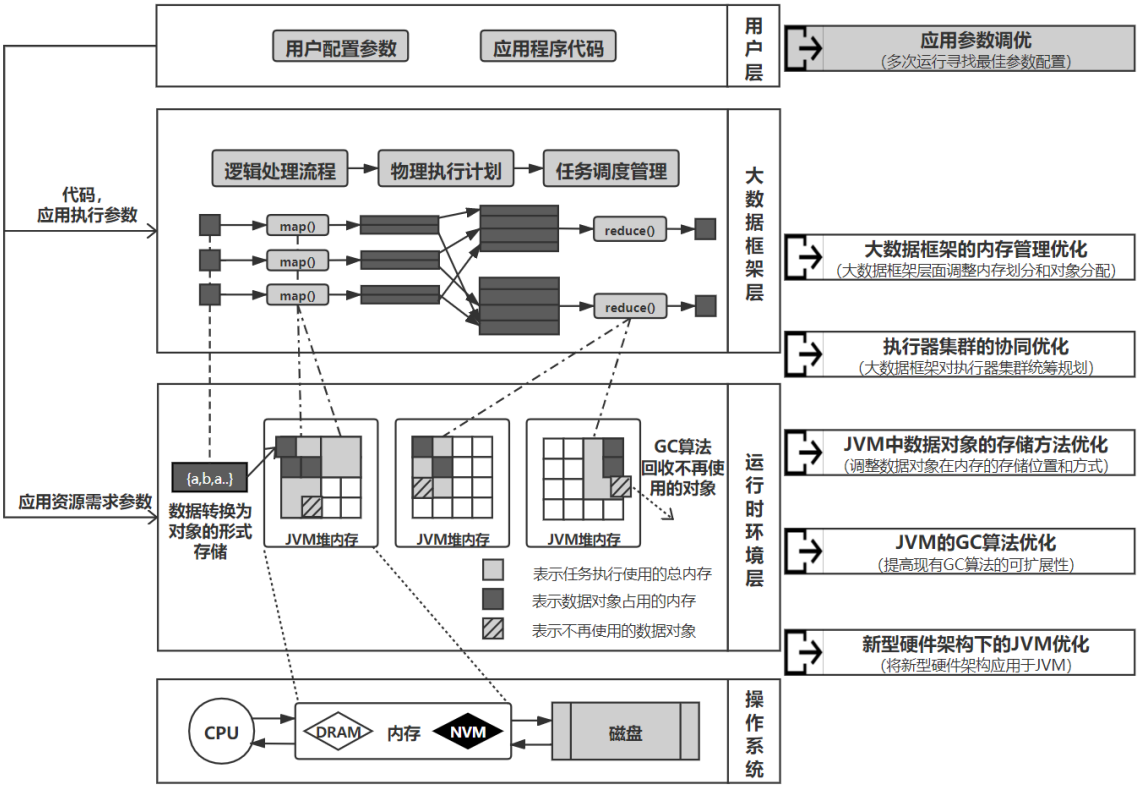


图 4 大数据处理框架的层次结构

从大数据框架层出发: (1)可以对大数据处理框架粗粒度的内存管理模式进行优化,以对执行器 JVM 更友好的策略进行对象申请,应对变化的内存使用模式.具体包括根据运行时状态信息,对框架的内存分配参数进行动态调整;根据对象在大数据框架中的用途和使用周期信息,将对象基于生命周期进行申请和管理. (2)可以对框架下的 JVM 集群进行协调,通过执行器 JVM 和大数据框架在运行时的交互,使得每一个 JVM 的任务执行过程最有利于大数据框架的整体性能,消除 JVM 与上层框架的隔阂.具体包括对所有执行器 JVM 的 GC 决策进行统筹;对历史的执行器 JVM 进行重用;对统一物理节点上的各个执行器 JVM 的内存分配进行动态规划.

从运行时环境层出发: (1)可以对 JVM 存储数据路径对象的方法进行优化,提高数据在内存中的存储和管理效率,应对增大的内存使用压力.具体包括在专用的内存区域集中存储和管理数据对象;以二进制的形式在内存中存储数据对象的数据值. (2)可以直接对 JVM 中主流的 GC 算法进行优化,提高 GC 算法的执行效率和对大数据环境的适应性.具体包括提高 Parallel GC 的并行度;拓展 G1 GC 的年龄代划分. (3)可以将新型的硬件架构应用于 JVM,提高 JVM 的资源利用率和数据缓存能力,同时提高大数据框架的可拓展性和容错能力.

表 2 大数据处理框架下的 JVM 优化技术分类

优化层面	优化方向	优化技术	代表工作	解决问题
大数据框架层	大数据框架的内存管理优化	大数据框架内存分配参数的动态调整	MEMTUNE[31]	内存使用模式变化
		基于生命周期的大数据框架对象管理	Deca[46,47]	
	执行器集群的协同优化	基于全局协调的执行器 GC 时机决策	Taurus[19]	JVM 与上层框架隔阂
		基于历史的执行器 JVM 重用	HotTub[8]	
		基于动态规划的执行器内存弹性分配	ElasticMem[48]	
运行时环境层	JVM 中数据对象的存储方法优化	基于区域的数据对象存储和管理	Yak[49]	内存使用压力增大
		基于二进制的数据对象序列化存储	Gerenuk[22]	
	JVM 的 GC 算法优化	基于 Parallel GC 的算法并行度优化	Platinum[20]	
		基于 G1 GC 的年代划分数量拓展	ROLP[50]	内存使用模式变化
	新型硬件架构下的 JVM 优化	基于内存分解架构的 JVM 数据本地化	Semeru[51]	
		基于非易失性存储器的 JVM 内存拓展	Panthera[52]	

后续的章节将根据优化方向,具体介绍各种优化技术的具体实现和效果特点.第 4 节介绍了大数据框架内存管理的优化技术,第 5 节介绍了执行器集群的协同优化技术,第 6 节介绍了 JVM 中数据对象存储方法的优化技术,第 7 节介绍了 JVM 中 GC 算法的优化技术,第 8 节介绍了新型硬件架构下的 JVM 优化技术.

4 大数据框架的内存管理优化

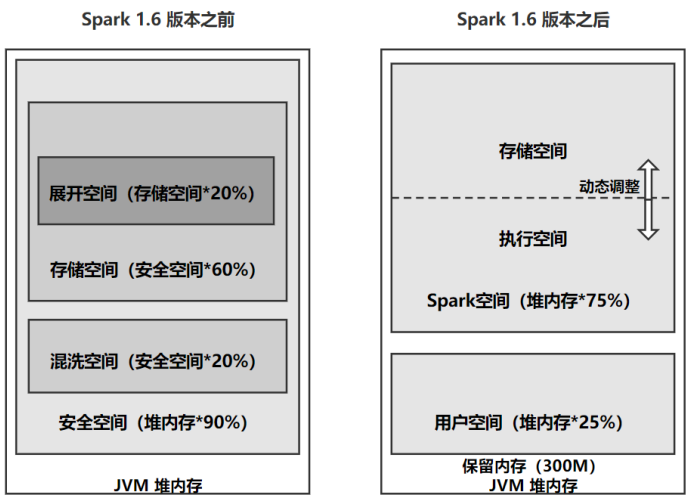
由于大数据框架静态的内存用途分配,并不能适应不同的大数据应用在不同处理阶段变化的内存使用模式,一些研究工作了提出在运行时动态调整大数据框架内存分配参数的策略;由于执行器 JVM 无法了解对象的存活时间,一些研究工作提出在大数据框架中解析应用产生对象的生命周期信息,利用相关信息协助执行器 JVM 的内存管理.表 3 列出了本章介绍的大数据框架的内存管理优化技术.

表 3 大数据框架的内存管理优化技术分类

优化技术	针对问题	实现方法	代表工作	其他工作
大数据框架内存分配参数的动态调整	大数据框架默认的内存用途分配固定而非最优	根据应用在运行时的执行情况动态调节大数据框架的内存分配比例	MEMTUNE[31] (IPDPS'16)	[53,54,55,56]
基于生命周期的大数据框架对象管理	执行器 JVM 在处理大量不同生命周期对象时的效率低下	根据对象在大数据框架中的用途和使用阶段确定对象的生命周期	Deca[47] (TOCS'19)	[57,58,59]

4.1 优化技术1: 大数据框架内存分配参数的动态调整

大数据处理框架并不能直接申请和管理执行节点内存,为了避免内存过量申请触发的 OOM 错误,大数据处理框架普遍采用粗粒度的内存管理策略,在计算任务执行过程中统计和估算存活对象在 JVM 堆内存中占据的大小,并将每种用途对象占用的内存总大小控制在一定参数阈值之内.以大数据处理框架 Spark 为例,如图 5 所示,Spark 在 1.6 版本之前,将执行器 JVM 堆内存划分为不同固定大小的空间,分别用于 RDD 缓存、混洗操作,和 RDD 的序列化/反序列化展开等任务.如果用于 RDD 缓存的存储空间剩余大小不足,Spark 就会通过一定的策



任务执行内存短缺或是混洗内存短缺.如果混洗内存短缺,控制器则降低 JVM 堆内存大小和存储空间比例,给予混洗空间和堆外的 I/O 缓冲区更大空间;如果执行内存短缺,控制器则增大 JVM 堆内存或是降低存储空间比例,保证有足够内存用于任务执行.否则,控制器就提供尽可能大的内存空间用于缓存 RDD,提高应用执行的数据处理效率.验证测试显示,MEMTUNE 可以将 Spark 的总体性能提高 46%.

图 5 Spark 对 JVM 堆内存的粗粒度划分

类似的,ATuMm[53]基于过往任务的运行时日志,在每一次执行新任务之前动态调整空间和执行空间的大小.TeraCache[54]对 Spark 的内存用途做了一定修改,更直接地分为用于任务执行和用于内存映射 I/O,通过比较 GC 频率和缺页频率,对两种区域的占比划分进行动态的自适应调整.Mammoth[55]将大部分内存划分为固定大小的缓存单元组成缓存池,用于 Hadoop 中 Map、Reduce 任务的内存分配,通过设置获取和释放缓存单元的优先级,动态调整用于不同类型任务的内存大小.DMATS[56]则根据 Spark 的任务内存需求量和运行时的 GC 信息,动态调整 Spark 的任务并行度参数.另外,Spark 从 1.6 版本开始提出了统一内存管理,如图 5 所示,统一内存管理允许用于 RDD 缓存和用于混洗的内存空间相互借用,实现大数据处理框架内存分配参数的动态调整.

4.2 优化技术2: 基于生命周期的大数据框架对象管理

直接管理内存的执行器 JVM 并不能感知每一个对象的生命周期,但大数据框架可以根据对象的用途和应用的处理流程,了解各种对象的使用阶段.通过解析对象的使用阶段,将生命周期相近的对象聚合管理,协助执行器 JVM 进行内存回收.

Deca 根据生命周期将大数据框架 Spark 中用户定义类型的对象分为 3 种类别:用户定义函数变量、缓存 RDD、混洗缓冲对象.其中,缓存 RDD 的生命周期可以显示地由 cache()函数和 unpersist()函数确定,通常是长时间存活的;混洗缓冲对象根据混洗操作的类型,可能在短时间内被新的对象替代,也可能长时间存活到混洗操作结束;而用户定义函数变量产生的大部分对象被认为是短寿命对象,可以当做普通对象由执行器 JVM 直接处理.Deca 通过代码分析,找出在运行时不会发生大小变化,或是通过增量对象申请实现大小变化的用户定义类型,分解出其各个域的数据值,并以二进制的形式存储在字节数组当中.这些对象的引用则根据上述 3 种类别,分别存放在对应的容器当中,基于生命周期管理和释放.Deca 极大降低了 GC 机制需要处理的对象数量,减轻了执行器 JVM 内存管理负担.在特定的测试条件下,Deca 可以减少 99.9%的 GC 暂停时间.

类似的,大数据处理框架 Flink 将执行器 JVM 堆内存的大部分空间以字节数组的形式划分为内存段,用于聚合存储对象的数据值,在一个任务管理器关闭时集中释放.而 TeraCache 将缓存 RDD 通过内存映射 I/O 保留在 JVM 堆中,也同样依靠 cache()函数和 unpersist()函数确定缓存 RDD 的生命周期,绕开常规的 GC 算法,在生命周期结束之后直接回收.

大数据处理框架根据代码构建出的数据流操作符图同样可以指示部分对象的生命周期.以大数据处理框架 Naiad[60]为例,Naiad 将计算依赖抽象为图,用节点来表示计算,用边来表示数据流.对于一个节点创建的对象,如果没有通过消息传递分享到其他节点,那么它的存活时间不会超过所在节点操作符.Broom[57]提出将 Naiad 创建的对象分为 3 种类型:跨计算节点控制的数据、单个计算节点控制的数据、短寿命的临时数据,3 种类型的对象的生命周期依次递减.Broom 创建了不同区域以存储不同类型的对象,使得用户在编写 Naiad 应用时,可以显式地将数据申请存放在对应的区域.在用户正确分类的情况下,Broom 可以在一个操作符完成后直接清理一个区域而无需系统 GC.验证测试显示,Broom 可以减少 34%的应用执行时间.

类似的,大数据处理框架 Spark 中表示作业处理流程的有向无环图也被用于推测缓存 RDD 的生命周期.MEMTUNE 根据有向无环图确定一个阶段内会被引用到的 RDD 块,并在 RDD 缓存空间不足时换出不会被引用的缓存 RDD 块.同样的,LCS[58]也根据有向无环图确定一个缓存 RDD 的最后使用阶段,以便于在缓存 RDD 生命周期结束后将其换出内存.而 MRD[59]根据有向无环图确定当前缓存 RDD 的引用距离,帮助决策缓存 RDD 的预取和换出.

4.3 小结

本章介绍了针对大数据框架内存管理的优化技术,包括根据运行时的状态信息,动态调整大数据处理框架的内存分配参数,提高执行器内存利用率;包括根据大数据处理框架的逻辑处理流程,判断大数据应用产生对象的生命周期,协助执行器 JVM 内存管理.这些方法能够有效提高 JVM 运行效率,许多方法都建立在某一个或者某一类大数据框架之上,通用性仍有待提高.

5 执行器集群的协同优化

在运行时,执行器 JVM 通常独立进行决策,大数据框架对执行器的干预相对有限.JVM 根据自身运行指标的决策可能符合自身当前的需求,但在集群角度上并非最优.一些研究工作通过全局协调执行器 JVM 的 GC 时机,避免单个执行 JVM 的 GC 暂停影响大数据应用的整体进度;一些研究工作通过重用历史的执行器 JVM,缓解 JVM 冷启动对大数据应用执行总时间的影响;一些研究工作通过动态规划相同节点上执行器 JVM 的内存分配,使得执行器集群所有应用的整体故障数量最少,执行总时间最短.表 4 列出了本章介绍的执行器 JVM 之间的统筹优化技术.

表 4 JVM 集群的协同优化技术分类

优化技术	针对问题	实现方法	代表工作	其他工作
基于全局协调的执行器 GC 时机决策	不同执行器 JVM 交错的 GC 暂停影响全局进度	根据全局监控信息协调所有执行器 JVM 在同一时间统一 GC	Taurus[19] (ASPLOS'16)	[61,62]
基于历史的执行器 JVM 重用	JVM 冷启动用时在大数据应用执行总时间中的占比较大	在作业完成后保留执行器 JVM,执行新作业时匹配 JVM 重用	HotTub[8] (OSDI'16)	[63,64,65]
基于动态规划的执行器内存弹性分配	大数据框架无法在执行器启动时准确预估任务所需内存大小	根据执行器 JVM 增加堆内存的价值和 GC 的代价决定内存分配	ElasticMem[48] (ATC'17)	[66,67]

5.1 优化技术1: 基于全局协调的执行器GC时机决策

单个执行器 JVM 的 GC 可能影响到处理作业的整体进程.例如,对基于 MapReduce 的大数据处理框架如 Hadoop 和 Spark,各个执行器 JVM 在混洗阶段或者一个迭代的超步结束之后,需要完成相互的数据交换后才能整体向下执行.如果有 JVM 节点在这一个过程中触发 Major GC 长时间全局暂停,其它节点只能空等这一节点完成 GC.更严重的是当集群继续向下执行时,其它 JVM 节点可能在较短的时间间隔之后也触发 Major GC,造成整个作业频繁中止.研究工作尝试通过对所有执行器 JVM 的 GC 时机进行统筹规划,降低单个执行器 JVM 的 GC 对全局进度的影响.

Taurus 受分布式操作系统的启发,将集群中的运行时系统更紧密的集成,实现一个跨节点运行的整体运行时系统以协调全局的 GC 决策.对于前述场景,Taurus 相应的解决办法就是协调所有执行器 JVM 节点进行同步的 GC.Taurus 对所有执行器 JVM 堆内存的使用情况进行监测,一旦存在某个执行器 JVM 的内存使用达到阈值即将触发 GC,就协调所有执行器 JVM 同步 GC.由于绝大部分 JVM 节点执行的任务类型相近,在物理资源配置相同的情况下,堆内存使用情况也大致类似,这个 GC 决策对大部分节点而言也是合理的,可以避免交错的 GC 带来的连续全局暂停.验证测试显示,Taurus 可以加快 21%的应用执行速度.

然而,单一的全局 GC 策略并不能满足所有的应用场景,例如对于延迟敏感的数据库请求,一次 Minor GC 引发的全局暂停,就可能带来相对严重的额外延迟,产生掉队任务.此时的全局最优决策不再是全局同步 GC,而应当是将请求调度到短时间内暂时不会触发 GC 的 JVM 节点执行,避免在任务执行时触发 GC.因而 Taurus 的设计需要应用程序开发人员提供具体策略,用领域特定语言(Domain Specific Language, DSL)编写,指导 Taurus 执行进行 GC 决策.类似的,GCI[61]避免了将任务调度到正在进行 GC 的 JVM 节点上,而 MURS[62]挂起可能带来重 GC 负担的任务,让轻量级任务优先快速完成,缓解整体任务的内存压力.

5.2 优化技术2: 基于历史的执行器JVM重用

由于大数据处理框架复杂的软件栈,大数据应用类加载的工作负担远超普通应用,这使得执行器 JVM 冷启动的初始化耗时可能占据了大数据应用总执行时间的 30%以上,这对延迟敏感的大数据应用是难以接受的.但初始化工作的总量不会随着任务的数据量明显变化,因而相对来说,长时间执行的任务可以把初始化消耗平摊到总执行时间当中,更好地利用即时编译带来的性能提升.研究工作尝试通过延长执行器 JVM 的运行时间,减少冷启动 JVM 的次数.

HotTub 在大数据框架的层面上,保留执行过作业的执行器 JVM.当新的作业提交后,HotTub 参考执行过的作业信息,对历史执行器 JVM 进行选择和重用.HotTub 建立了一个 JVM 池,对于新提交的事务,首先计算它要加载的特征类的校验和,如果校验和与某个历史作业匹配,就认为当前作业和之前的作业具有一致性,可以调度 JVM 池中相应的执行器 JVM 执行.如果匹配到的 JVM 都处于工作状态,就通过 Fork 相应的 Java 进程再创建一个所需的 JVM.等待作业执行完毕,JVM 调用 GC 清理掉 JVM 堆栈中的任务数据,让 JVM 继续保持,等待再次被匹配.这种方法适用于对延迟敏感的查询事务,由于查询语句的类型有限,重复度较高,而相同的查询请求需要加载的类和编译的热点代码是高度一致的.如果重用已经预热的执行器 JVM,就可以省去冷启动 JVM 的开销,也可以有效利用 CPU 缓存和 TLB.在理想情况下,HotTub 可以将 Spark 查询的速度提升至原先的 1.8 倍.

类似的,一些大数据处理框架尽可能让每个执行器 JVM 保持更长时间,执行更多计算任务,或是重用执行器 JVM 执行不同的作业,以平摊 JVM 冷启动的开销.例如,在 Spark 的执行计划当中,一个执行器 JVM 将被保持到一整个作业结束,Tez[63]支持将执行器设计为可以连续执行多个作业,Nailgun[64]保持一个长时间运行的执行器 JVM 来动态运行各种作业.不过,这些重用方法并不能确保正确性和一致性,很多静态数据在运行前需要重新初始化.而 WM[65]在集群内提供一个半隔离的环境来对新添加的执行器进行预热.

5.3 优化技术3: 基于动态规划的执行器内存弹性分配

大数据框架对任务所需内存的估计不准确,将会影响任务所创建执行器 JVM 的运行效率.执行器 JVM 完成任务所需的内存受很多运行时因素影响难以准确估计.分配过多可能造成内存资源浪费,而分配过少可能引发频发 GC 和数据溢写,带来性能损失[68].对于基于服务的大数据处理框架,多个大数据应用可能在一个机器节点上同时运行,尤其需要在各个应用的内存分配上做好权衡.研究工作提出动态规划位于同一机器节点上执行器 JVM 的内存分配,最大化利用物理内存资源.

ElasticMem 通过动态规划同一机器节点上各个执行器 JVM 的内存分配,实现内存利用价值的最大化,即失败的执行器 JVM 尽可能少,所有任务的总执行时间尽可能短.ElasticMem 首先对 Hotspot 的 Parallel GC 进行了修改,实现运行时堆内存大小的动态调整,并提供了调整堆内存大小,外部触发 GC,获取堆状态信息的接口.在执行器 JVM 可用内存不足时,ElasticMem 考虑 5 种操作选项:调整堆大小、触发 Young GC、触发 Parallel 原生 Full GC(先进行 Young GC 晋升,再收集老年代)、触发改进版的 Full GC(直接对整个堆内存收集)、空置等待、终止 JVM 进程.ElasticMem 设计了启发式的操作选择策略,为每种操作都设计了对应的代价计算方法,代价越小则价值越高.其中终止 JVM 进程和空置等待的代价具有更高的考虑优先级,意味着这两种选项会尽量不被选择,而其他操作的代价是产生单位可用内存所需要的时间,具体的代价数值的计算依赖于机器学习模型对于 JVM 中存活和死亡对象数量的预测,以及对于进行 GC 用时的预测.ElasticMem 对每个执行器 JVM 的具体选择利用背包问题模型进行动态规划,为了降低算法复杂度,堆内存大小的调整尺度被设置为粗粒度的块,或是一定比例的剩余物理内存.由此 ElasticMem 在可接受的代价下,实现对同一个物理节点上不同执行器 JVM 内存的动态协调,提高了内存资源的使用效率.在内存紧张的环境下,ElasticMem 可以减少 30%的应用执行时间.

类似的,Forseti[66]对每个 JVM 的堆大小与应用程序吞吐量进行建模,基于经济效用,最大限度地提高 JVM 集群的综合吞吐量.而 SmartGC[67]根据应用程序的需要动态伸缩 JVM 内存,降低集群整体的内存资源消耗.

5.4 小结

本章介绍了 JVM 集群的协同优化技术,包括全局协调执行器 JVM 的 GC 时机,匹配大数据应用执行的整体

需求;包括重用历史的执行器 JVM,平摊 JVM 的初始化消耗;包括动态规划同一节点上执行器 JVM 的内存分配,最大化利用机器的内存资源加快集群的整体处理速度.这些方法使得在运行时,执行器 JVM 之间的联系更加紧密,协作更加高效.不过,不同应用的不同阶段对执行器 GC 决策方法有不同期望,依赖用户编写和调整全局的 GC 策略过于繁琐;维持一个 JVM 池用于重用的做法只适用于连续执行相同类型大数据应用的场景,否则维持大量 JVM 也需要内存消耗,对内存紧张的应用场景并不友好;动态协调执行器 JVM 内存分配的规划算法复杂度较高,预测模型需要提前训练且可移植性有限.

6 JVM 中数据对象的存储方法优化

大数据应用数据路径产生的数据对象在堆内存中数量庞大,分布分散,长时间存活,对 GC 算法的工作速率存在负面影响.一些研究工作对数据对象采取集中的存储和处理,提高执行器 JVM 对数据对象的管理效率;一些研究工作用二进制的形式来存储数据值,控制大数据应用产生数据对象的整体数量,降低 GC 算法的工作量,提高节点内存的利用效率,节省数据在节点间传输所需的序列化/反序列化消耗.表 5 列出了本章介绍的 JVM 中数据对象存储方法的优化技术.

表 5 JVM 中数据对象存储方法的优化技术分类

优化技术	针对问题	实现方法	代表工作	其他工作
基于区域的数据对象存储和处理	数据对象和传统对象生命周期的差异影响 GC 算法效率	在专门的内存区域存储和管理数据对象,减轻 GC 算法工作量	Yak[49] (OSDI'16)	[24,69,70,71]
基于二进制的数据对象序列化存储	以对象的形式存储数据内存利用率较低,GC 负担较重	将数据对象序列化为二进制的形式,算法直接对二进制数据值操作	Gerenuk[22] (SOSP'19)	[6,47,72]

6.1 优化技术1: 基于区域的数据对象存储和处理

执行器 JVM 的原生 GC 在管理大量长时间存活的数据对象时往往效率低下,而数据对象的集中存储给执行器 JVM 提供了绕过原生 GC,直接回收数据对象的机会.由于数据路径代码和控制路径代码有清晰界限,开发人员可以在应用代码中感知数据对象的创建使用周期.根据开发人员对相关信息的注释,执行器 JVM 可以在特定的内存区域中集中存储数据对象,并在恰当的时机直接清理相应内存区域.

Yak 将执行器 JVM 的堆空间划分为控制空间和数据空间,分别用于存储控制路径对象和数据路径对象,并修改了原生 GC 算法的工作范围,使其只在控制空间工作.开发者根据经验标注年代纪元(epoch)的开始和结束,系统在执行过程中,每当处理到纪元开始的标记,则在数据空间创建一个区域,在对应的纪元结束出现之前,所有的数据路径对象都申请在这个区域当中,当纪元结束出现时,Yak 会清理相应区域.纪元的创建支持嵌套,而每个纪元区域的记忆集记录着其他纪元区域引用到本区域的对象,这些存活的对象将被晋升到嵌套最外层祖先纪元的区域当中去.由于数据对象的数量在大数据应用产生的对象数量中占了绝大部分,更多的内存在初始化时被分配给数据空间,控制空间在剩余不足时再申请更多.Yak 在绕开了常规 GC 的同时引入了数据空间对象的晋升机制,提升了可用性和适用范围.但为了实现晋升,为每个对象增加了 4 个字节的头部,这增加了平均 12.2% 的内存开销,在内存压力和 GC 效率之间做出了一定权衡.

类似的,Yak 之前的工作 FACADE[24]在 JVM 堆外的内存空间集中存储数据对象的数据值,而在堆内内存保留了操作数据值的简化对象,同一个类的数据对象值都映射到同一个堆内的简化对象上.FACADE 要求开发者标注出所有数据路径的类,创建相应数量的简化对象,并根据开发者标注的迭代开始和结束位置,在每次迭代结束之后直接清理掉堆外的数据存储空间,使绝大部分的数据对象绕开了正常 GC 算法.Gerenuk 同样将数据对象值集中存储在堆外内存,并根据开发者标注的序列化和反序列化节点确定数据对象可以清理的时机.

基于区域的数据对象存储也有助于提高 GC 线程和应用线程的缓存命中率.GC 算法的执行是一种典型的图遍历过程,而图遍历容易受到空间局部性的影响.大数据处理框架下的数据对象和控制对象在执行器 JVM 的堆内存中交错离散分布,算法处理顺序相近的数据对象空间不临近,造成 CPU 执行过程中触发 TLB 缺页和

L1/L2/L3 缓存未命中的概率较高.而集中存储和处理数据对象可以提高算法空间局部性.

DSA[69]为大数据框架的主要数据结构类在 JVM 堆内存中创建一块连续的区域,用于单独存储属于这个类的数据对象.大数据存储框架大都面向单一或者少数数据结构,如树结构或者哈希链表结构,其中数据对象以节点的形式插入在数据结构当中.DSA 根据开发者标注出主要数据结构类,创建对应数据对象节点的集中存储区域,并根据开发者标记出的节点从数据结构中被删除的位置,确定在每次 GC 触发时具体存活的数据对象.在 GC 扫描的过程中,DSA 将数据结构区域中存活的节点直接标记存活,加入到 GC 根节点集合当中,作为可达性分析的初始对象.由于这些数据对象节点的内存位置相近,遍历的过程近似于按照内存位置顺序,因而处在相同 TLB 上的数据对象可以一次性处理,减少 TLB 缺页出现次数.直接将存活节点对象加入根节点,相比由深度优先搜索到达再标记降低了复杂性,也减小了遍历扫描栈的大小.而节点对象之间没有依赖,使得对这些数据结构区域的处理可以高度并行,并在 GC 算法的任务窃取机制下能够灵活被调度,充分平衡 GC 线程负载.但如果开发者没能准确地标注出所有节点的增删,将会直接触发 Full GC,带来性能损失.

类似的,大数据处理框架 Flink 以及 Spark 的 Tungsten[72]内存管理器,都以字节数组的形式集中连续存储用于操作的数据对象值.这些大数据处理框架通过设计缓存友好的底层算法和数据结构,连续访问这些集中存储的数据值,可以充分利用的 CPU 的 L1/L2/L3 缓存,提高算法的执行速率.而 HCSGC[70]和 ThinGC[71]按照程序访问顺序和访问热点聚合热点对象,分离低访问频率对象的方法,在 GC 算法层面上改善内存布局.

6.2 优化技术2: 基于二进制的数据对象序列化存储

大数据处理框架下的数据都以对象的形式存在执行器 JVM 当中,然而在对象的形式下,对象头以及对象引用占用了大量内存空间,造成内存能够容纳的数据条目数量远小于理论数量.另外,对象作为 GC 算法操作的基本单元,大量的数据对象的长期存活将给内存管理带来巨大负担.因而,研究工作尝试将数据对象以二进制的形式进行压缩.

Hyracks[6]早先提出了一种开发者层面的编程范式,将大量相同类型的数据合并成一个大对象,将数据以二进制的形式填装到大对象中.Hyracks 注意到,数据对象的数量会随着数据处理量的增加而陡增,而较大的数据对象可以更多平摊对象外壳的空间开销,很多同类型的数据对象又有着相近的生命周期.所以,Hyracks 使用类似非托管编程语言的内存分配方法,显示地申请固定长度的字节数组对象作为内存页,将数据对象的值以二级制的形式逐个写入,显著提高了内存的利用率和 GC 的工作效率.尽管如此,Hyracks 作为一种编程范式,还需要开发人员自行实现不同类型对象的存储和获取方法.

类似的,FACADE 在堆外内存页中存储二进制的数据对象值,与 Hyracks 不同,FACADE 在编译器层面实现了二进制数据值的存取,通过重新编译应用程序的 Java 字节码,自动化完成数据值在堆外内存的二进制存储.Deca 同样以字节数组的形式存储用户定义类型对象的数据本身,并提出了按照用户定义类型的域来分别存储数据值,降低了存储的信息熵,进一步提高了内存使用率.Deca 观察到数据对象被压缩到容器之后,更多的缓存可以保留在内存当中而不被换出到磁盘,提升了大数据应用的工作效率.Spark 的内存管理器 Tungsten 直接以二进制形式存储和操作数据,极大提升了查询的效率和速度.Flink 则通过内存段实现对堆外内存以及堆内内存的直接管理,由定制化的序列化工具将绝大部分数据类型对象高效序列化,并实现了二进制数据的直接操作.

以二进制的形式存储数据值也为节省数据在网络中传输所需的序列化/反序列化开销提供了可能.由于大数据处理框架的分布式结构特点,大量数据对象需要通过网络在节点之间传输,而数据对象在网络中传输需要序列化为二进制形式.具体的对象传输过程如图 6 所示:首先发送节点提取待传输对象和它引用图中对象的数据,序列化成二进制数值,然后进入网络传输到接收 JVM 节点.接收节点读取二进制序列后,将数据反序列化,重新构建起对象.测试表明序列化和反序列化的数据转换过程需要大量的运行时操作,频繁调用反射函数,整个过程在 JVM 的执行时间中占比超过 30%.为了节省这个过程带来的开销,Skyway[7]统一采用对象的形式在网络传输和 JVM 堆内存中表示数据,将整个数据对象的信息全部传递.这种方法尽管节省了数据形式转换的开销,但增加了网络传输的开销.

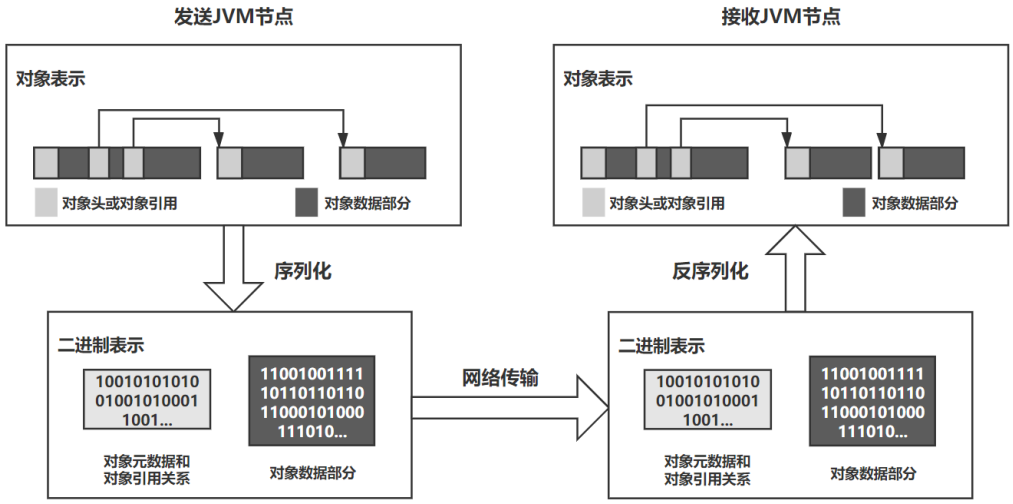


图 6 大数据分布式环境的跨节点数据传输流程

Gerenuk 采用相反的策略,统一以二进制的形式在网络传输和内存中表示数据.Gerenuk 观察到绝大部分的数据对象在执行过程中是数值不变且大小受限的,以对象的形式存储是没有必要的.Gerenuk 将开发者标注的序列化和反序列化位置之间的执行区域称为投机执行区域,通过这两个点代码静态分析确定数据流程,再根据开发者标注出的顶层用户定义数据类型和大数据处理框架的数据集合类型,找出所有被引用的数据类.Gerenuk 的运行时将这些类型的对象数据值以二进制的形式存储在本地内存的缓冲区中,Gerenuk 的编译器则自动转换用户代码的操作语句,使其直接操作内存缓冲区中的数据值,数据值以内联的形式在代码中展开,而不再需要反序列化为对象.然而 Gerenuk 对数据对象一致不变性的乐观假设并不一直成立,在违反假设的程序位置终止投机执行,Gerenuk 重新启动一个新的 JVM 执行器将数据反序列化为对象的形式,再执行原始的任务代码.测试表明不符合乐观假设的情况出现频率有限,并不会使投机执行的效果产生过多折扣,Gerenuk 在 Spark 和 Hadoop 上分别将端到端性能提高到了 2 倍和 1.4 倍.

6.3 小结

本章介绍了 JVM 中数据对象存储方法的优化技术,包括在独立区域中集中存储和处理数据对象,提高执行器 JVM 缓存命中率,并高效直接地清理相关内存区域;包括以二进制的形式存储数据对象的数据值,提高内存的使用效率,降低 GC 算法需要处理的数据对象数量,并节省一定的序列化/反序列化开销.这些方法能够有效提高执行器 JVM 的对数据对象的管理效率,但大多数优化方法都额外增加了开发者的工作负担,需要开发者标注数据对象的类型,数据对象的生命周期等等,要求更多的开发者经验,存在一定的执行异常风险.

7 JVM 的 GC 算法优化

在 JVM 的主流实现 HotSpot 包含的 GC 算法当中,绝大部分算法都是基于弱世代假说而采取分代收集,并不能适应大数据环境下巨大的对象数量,复杂的对象生命周期.当前最新版本的 Hotspot 包含了并行收集算法 Parallel GC,部分并发收集算法 CMS,G1,以及主体并发收集算法 Shenandoah[73],ZGC[74].其中 Parallel GC 在大数据处理框架的使用过程当中常常会触发长时间暂停,而 CMS,G1 算法尽管可以控制单次暂停时间,但 GC 频次和总暂停时间无法保证,高频次的对象移动和对象扫描无法避免,新一代的 Shenandoah 和 ZGC 能够和工作线程高度并发,但算法复杂度较高,伴随着物理资源消耗和吞吐量代价.研究工作尝试对最常用的两种 GC 算法 Parallel GC 和 G1 GC 进行优化.一些工作致力于提升 Parallel GC 的并行度,充分利用 CPU 资源,加速 GC 算法的处理过程,或是实现与任务线程的高度并发;另一些工作尝试拓展年代数量,提高 G1 GC 对数据对象复杂生命周期的适应性.表 6 列出了本章介绍的 JVM 的 GC 算法优化技术.

表 6 JVM 的 GC 算法优化技术分类

优化技术	针对问题	实现方法	代表工作	其他工作
基于 Parallel GC 的并行度优化	Parallel GC 在多核环境下不能充分利用 CPU 资源	在 CPU 核之间平衡 GC 任务负载,或将闲置 CPU 核用于应用线程	Platinum[20] (ATC'20)	[75,76,77]
基于 G1 GC 的年代划分数量拓展	不同生命周期对象没有区分,增加长寿对象的移动次数	拓展 G1 年代到多年代,将生命周期相近的对象申请到相同年代区域	ROLP[50] (EuroSys'19)	[78,79]

7.1 优化技术1: 基于Parallel GC的并行度优化

Parallel GC 算法的设计初衷是在全局暂停时,JVM 能够多线程并行地从 GC 任务队列当中获取和执行 GC 任务,达到加快处理速度,降低全局暂停时间的目的,然而测试结果显示,Parallel GC 在多核环境中的可拓展性并不能达到预期.研究工作尝试通过平衡 CPU 核之间的任务负载,减少各个 GC 线程之间的区域依赖,以及利用闲置的 CPU 核并发执行应用线程,提高 Parallel GC 在多核环境下的 CPU 资源利用率.

Suo 等人[75]注意到,Parallel GC 在 GC 线程和 CPU 核之间的负载不均是影响 Parallel GC 并行度的重要因素.他们使用 Parallel GC 在 Dacapo[80],Hibench[81]等几个经典的 JVM Benchmark 上进行测试分析,发现绝大部分 GC 任务都是由个别 GC 线程完成的,且这些 GC 线程都是由个别 CPU 核完成的.原因是 Linux 定期平衡多核负载的时间粒度过大,远超过了正常 GC 线程的执行时间,而 GC 线程频繁进入睡眠状态使得在 Linux 的平衡策略下更难有机会移动.GC 任务在 GC 线程之间的分布不均则是 Parallel GC 任务队列不合理的锁竞争策略造成.由于最初持有 GC 任务队列锁的 GC 线程和接替锁的 GC 线程都在同一个核上,持有锁的 GC 线程释放锁之后,可能利用剩余的 CPU 时间片完成了 GC 任务,再次申请并获得了锁,导致 GC 任务总是由个别 GC 线程取得.

Parallel GC 设计了任务窃取机制来平衡 GC 线程之间的负载,当 GC 线程拿到 CPU 时间片却没有可执行 GC 任务时,就会尝试从其他 GC 线程窃取任务.但任务窃取的方式是从任意两个其他 GC 线程的本地任务队列中窃取.由于大部分 GC 线程的本地任务队列都为空,任务窃取的失败率很高,导致没有任务的 GC 线程最终只能浪费 CPU 时间片空等.Suo 等人的优化工作协助操作系统,将 GC 线程平衡到各个 CPU 核上,并规定 GC 线程在被唤醒之后,如果感知到当前 CPU 核的负载较重,就重定向到低负载的 CPU 上执行.对任务窃取的不平衡,Suo 等人的优化工作记录还有 GC 任务的 GC 线程,作为任务窃取的目标.通过这种方法还可以简化判定 GC 任务结束的条件,无需在多次窃取失败之后再结束 GC,提高了 Parallel GC 的并行度.性能测试显示,这些优化方法可以将应用的总执行时间降低接近 50%.

Scissor[76]通过对 JVM 堆内存的分析,将 Parallel GC 在线程之间并行程度较低的问题归咎于区域依赖.Parallel GC 为了实现 GC 任务的并行执行,将堆内存按区域划分,GC 线程在整理阶段将源区域的存活对象拷贝到线程自身负责的区域.然而当一个 GC 线程负责的区域还有存活对象没有被其他 GC 线程复制时,该 GC 线程不能在这个区域开始工作,即此时源区域的处理依赖于当前区域的处理.由于大数据框架产生的对象分布稠密而长时间存活,很多区域在 GC 时都存在不少存活的对象,造成区域依赖和连锁的区域依赖现象普遍出现,大

部分 GC 线程只能阻塞等待依赖的区域先被少数 GC 线程清理干净.Scissor 在 GC 线程处理到非空区域时,先申请一个影子区域用于拷贝源区域的存活对象,而当负责区域的存活对象都被处理干净之后,再将影子区域的对象拷贝回来.由于 Parallel GC 当中两个幸存者区中总有一个为空,恰好为影子区域的申请提供了空间,另外堆外内存也可以作为备选项.影子区域尽管增加了一定的 GC 任务工作量,但可以有效提高 GC 工作的并行度.另外对于存活对象占比很大的稠密区域,Scissor 选择了不移动,测试表明可以降低的内存读写压力.Parallel GC 在对象移动和拷贝中还存在的另一个问题是引用更新时的重复计算.Yu 等人[77]发现 Parallel GC 在计算存活对象的新地址时,需要累加源区域在当前对象之前所有存活对象的大小,这个工作是重复且不必要的.Yu 等设计在每个 GC 线程中保留上一次处理的源区域信息,包括其存活对象累加大小,如果 GC 线程处理的下一个对象仍在同一源区域中,则可以根据上次保留的结果,直接计算得到当前对象的新地址.

在充分利用分配给 GC 算法的 CPU 核的同时,多余的 CPU 核还可以被利用于在 Parallel GC 的全局暂停阶段并发执行应用线程.由于创建过多的 GC 线程对 GC 算法的整体效率帮助有限,甚至可能产生副作用.因此 Parallel GC 在 CPU 核数量较多时,会创建少于 CPU 核数量的 GC 线程数,这意味着在 GC 的全局暂停阶段会有部分 CPU 核闲置.而 Platinum 利用这一点,将 Parallel GC 全局暂停阶段中剩余的 CPU 核资源分配给应用线程,缓解 Minor GC 全局暂停对应用延迟的影响.根据观察,在 Cassandra 等大数据框架中,每个任务内存修改的对象位置都在一定的内存区间之中.于是 Platinum 在伊甸园区中保留了一部分用于在 GC 收集时继续应用线程的对象申请,并在保留内存的外围设置一个隔离区域,在 GC 前期搁置该区域,等大部伊甸园区收集完毕后再快速处理.为了实现隔离而不对所有操作都引入读写屏障的逻辑,Platinum 使用了硬件特性内存保护键(Memory Protection Keys),设置 GC 线程和工作线程的读写范围.另外 Platinum 还使用了限制事务内存特性(Restricted Transactional Memory),实现对在移动过程,对复制的新对象和源区域老对象进行同步修改,保证对新老对象的引用能够保持一致.通过将 Parallel GC 改造为一个并发收集器,Platinum 提高了 JVM 的整体 CPU 利用率和并行程度,并可以将应用的尾延迟降低接近 80%.

7.2 优化技术2: 基于G1 GC的年代划分数量拓展

尽管 G1 GC 采用基于区域的堆划分策略,在一定程度上能够降低管理长时间存活对象的开销,但其依旧延续了传统的年代划分,不能有效降低长时间存活对象的拷贝次数.由于在大数据处理框架之下,数据对象的生命周期与弱世代假说相悖,传统年代划分下的伊甸园区,幸存者区,老年代区 3 种年龄段,已经不能满足区分不同寿命数据对象的需求,研究工作尝试通过拓展年代划分数量,根据用户经验或是运行历史得到对象的寿命,将生命周期相近的对象放在一个年代中,减少对象的经历的扫描和拷贝次数.

NG2C[78]将 G1 GC 拓展到 N 个代,并将寿命预期相近的对象申请在同一个年代的区域当中.NG2C 要求开发者根据经验对数据对象的生命周期做出注释.在申请内存时,如果对象没有开发者的注释,NG2C 将遵循 G1 原生的申请方式,将对象放置在年轻代的区域;如果对象有开发者注释的年龄代,那么负责申请空间的线程将创建或寻找相应年龄代并分配区域,放置对象到该年龄代的区域当中.在 GC 循环当中,这些新增年龄代的区域会和普通老年代区域一样,在存活对象比例低于一定阈值时被回收,存活对象都晋升到老年代中,而每一个年龄代所分配的区域数量可以在运行时根据需求可以动态调整.为了帮助开发者更准确地做出注释,NG2C 开发了一种分析工具,便于在注释之前先小规模的测试,确定各类对象的生命周期分布,更准确的注释也能帮助 NG2C 更好地展现出性能优势.

POLM2[79]在 NG2C 工作的基础之上,摆脱了年代注释工作对开发者和对 Java 源码的依赖.POLM2 根据运行时信息自动地预测对象的生命周期,并直接在应用程序的字节码上进行注释.POLM2 基于对象申请时的函数调用状态和代码位置信息来区分不同类型对象.工具分成了 4 个部分,其中记录器使用字节码改动工具 ASM,对每个对象申请的位置进行标注,并生成一个对象的 ID;转储器在每次 GC 结束之后使用快照工具 CRIU 给存活的对象记录一个快照;分析器在 JVM 运行一段时间之后通过对象 ID 匹配历史快照,确定每个申请位置对象存活的 GC 轮数,并以存活的 GC 轮数作为这个位置的对象的年龄预测;插装器通过 ASM 在相应的字节码位置上进行年龄注释,剩余的对象申请由 NG2C 完成.由于同一个代码位置上申请的对象不一定生存周期相

同,POLM2 利用函数调用树进行区分.不同调用栈下相同代码位置创建的对象,在函数调用树上有不同的父节点,最终得到的生命周期标注也就可以得到区分.

ROLP 继 POLM2 拓展了 NG2C 在 JVM 中的应用范围.POLM2 工作在 Java 字节码上,而 ROLP 对经过 JIT 即时编译的汇编代码进行类似操作,并将生命周期预测结果作为注释传递给 NG2C.经过 JIT 即时编译的热点代码,产生了绝大部分的数据对象,因而 ROLP 的优化对大数据应用性能更加关键.与 NG2C 不同的是,ROLP 为对象增加头部用于记录上下文:16 位用来表示对象申请的代码位置,16 位用来表示对象申请时线程函数调用状态.每种对象的生命周期被记录在一个全局的分布表上,分布表每 16 个 GC 周期更新一轮.一类对象在前一轮的统计结果中的最长存活时间,将在下一轮被认为是该类的寿命.应用线程维护 16 位的函数调用状态,用于在对象申请时传递给对象的头部,线程每次调用一个函数或从函数中返回时,函数调用状态值增加和减掉当前函数的特征数值.为了在解决代码位置冲突的同时降低状态维护的损耗,ROLP 只记录有区分作用的函数状态.测试结果显示,在对应用吞吐率影响不超过 6%的情况下,ROLP 可以降低超过 50%的尾延迟.

7.3 小结

本章介绍了 JVM 主流 GC 算法优化技术.其中针对 Parallel GC 的优化方法提高了 CPU 利用率和 GC 线程的并行度,加快了对象引用的更新速度,而针对 G1 GC 的优化方法在算法层面有效区分了不同生命周期对象,降低了对象在不同年代区域之间的移动次数.针对 GC 算法的优化技术能够普适于包括大数据处理框架等各种场景,但对于缓解内存使用量的帮助有限.另外,拓展年代划分数量的方法需要依赖开发者,或是通过一定时间的运行来确定对象具体的年代所属.

8 新型硬件架构下的 JVM 优化

一些拥有新特性的 CPU,内存,磁盘等硬件架构在近些年不断普及,其中很多新型硬件技术能够大数据框架下的执行器 JVM 带来性能提升.研究工作探索了利用新型硬件架构提升大数据框架下执行器 JVM 性能表现的方法.一些研究工作将内存分解架构(Memory Disaggregated Architecture)应用于大数据处理框架,并通过提高数据的本地化加快内存分解架构的工作速率;一些工作将非易失性存储器(Non-Volatile Memory, NVM)引入大数据处理框架下的 JVM 中,利用 NVM 缓解传统动态随机存取存储器(Dynamic Random Access Memory, DRAM)内存的使用压力,并利用 NVM 持久化存储特性加快大数据处理框架的故障恢复.表 7 列出了本章介绍的执行器 JVM 之间的统筹优化技术.

表 7 新型硬件架构下的 JVM 优化技术分类

优化技术	针对问题	实现方法	代表工作	其他工作
基于内存分解架构的 JVM 数据本地化	对象的空间局部性差,影响 GC 算法在资源分解架构上的性能	将大部分对象的管理工作交给距离更近的处理器,提高空间局部性	Semeru[51] (OSDI'20)	[82,83]
基于 NVM 的 JVM 内存拓展	使用 DRAM 的 JVM 在大数据框架下内存空间紧张,且数据宕机易失.	使用 NVM 和 DRAM 混合的 JVM 堆内存空间,缓解使用压力,提供持久化	Panthera[52] (PLDI'19)	[54,84,85]

8.1 优化技术1: 基于内存分解架构的执行器JVM数据本地化

内存分解架构通常提供了有效的远程内存访问技术,使得所有节点的内存资源可以统一调配和使用,但相对于本地内存访问,远程内存访问依旧是比较耗时的.由于 JVM 中对象的空间位置通常是离散分布的,体现在内存分解架构上就是对象分散在各个节点的物理内存当中,对象间的引用跨越物理节点.而传统 GC 的处理过程无法改善对象分布的空间局部性,使得资源分解架构对大数据处理框架和执行器 JVM 的性能提升受限.研究工作尝试通过修改 GC 算法在内存分解架构中的工作方式,提高对象的本地化和空间局部性.

Semeru 提出了一种分布式 JVM,针对基于远程直接内存访问(Remote Direct Memory Access,RDMA)的内存分解架构,将 GC 算法的大部分工作交由距离数据更近的内存服务器完成.Semeru 将所有的内存资源组成一个全局的 JVM 堆空间,统一了内存服务器和 CPU 服务器中的内存地址空间,其中每个内存服务器负责管理一段

内存,基于区域的管理.CPU 服务器自身拥有一小段内存作为运行缓存,负责具体执行应用程序,并在缓存缺失时通过 RDMA 从内存服务器中获取.内存服务器在 CPU 服务器工作的同时,利用多余空闲的 CPU 资源对自身负责的内存段进行持续的并发扫描,并计算出下一次清理之后的新地址.为了提高对象清理之后的内存局部性,对象更新地址的位置排列顺序根据对象被扫描的顺序决定,而用户可以选择深度优先扫描或是广度优先扫描.当整个堆内存的使用量超过一定比例时,Semeru 触发全局暂停的 GC,由 CPU 服务器清理自身的缓存空间,内存服务器根据标记阶段的结果清理相关区域,并接收来自 CPU 服务器的记录的关于跨节点和跨区域的引用信息,为后续的并发标记服务.

类似的,NumaGiC[82]针对缓存相干的非一致性内存访问(cache-coherent Non-Uniform Memory Access, ccNUMA)架构,通过 GC 算法实现对象和对象引用的本地化.NumaGiC 继承了 NAPS[83]的工作,使得 GC 线程可以根据地址信息获知对象所在节点位置,并能够将对象放置在期望的节点之上.在此基础上,为了减少跨越节点的对象引用,避免大量的远端内存访问,NumaGiC 设计了增强对象本地引用的内存布置策略,要求应用线程在对象申请时就在应用线程所在节点申请,而 GC 线程只扫描所在节点的 GC 根对象,并在对象整理阶段将所有存活对象都移动到 GC 线程所在节点.对于这种策略下可能的负载不均,NumaGiC 设计了 GC 任务窃取模式,在 GC 线程出载不均时窃取其他 GC 线程的任务,在前述策略的之下,恰好也可以将存活的对象复制到 GC 线程工作的结点,达到平衡对象数量和数据引用本地化的目的.在一定测试条件下,NumaGiC 相比直接使用 Parallel GC 可以将整体性能提升超过 90%.

8.2 优化技术2: 基于NVM的执行器JVM内存拓展

DRAM 作为 JVM 堆内存通常使用的存储介质,尽管读写速度远高于磁盘,但造价高昂,功耗较大,并且断电之后数据不可持久化.在对吞吐量和延迟有较高要求的大数据环境下,DRAM 需要承受巨大的使用压力.而 NVM 作为一种介于 DRAM 和磁盘之间的存储介质,拥有相比 DRAM 更廉价的单位价格,相比磁盘更快的读写速度,带宽可以达到 DRAM 的 1/3[86],并且具有可持久化特性.NVM 与 DRAM 组成的混合内存环境,为大数据应用的数据缓存和故障恢复带来了优化方向[87].

Panthera 将 NVM 加入到大数据处理框架的内存环境之中,为 Spark 提供更多的 RDD 缓存空间,提升数据处理的效率.Panthera 将 JVM 的 Parallel GC 拓展到混合内存空间.其中年轻代全部由 DRAM 组成,满足新创建对象的快速存取,老年代的一部分也由 DRAM 组成,存储长时间存活而且频繁读取的 RDD 对象,另一部分老年代由 NVM 组成,存储使用频率不高而生命周期较长的 RDD.Panthera 通过静态分析决定一个缓存或者混洗的 RDD 的存储区域:定义在循环外而在循环内使用的缓存 RDD 会长时间存活且频繁被使用,存储在 DRAM 上;而定义在循环内的缓存 RDD,最终存储在 NVM 上.Panthera 在内存申请时调用本地方法调整线程的申请位置到相应存储介质.与 Parallel GC 不完全一致的是,标注到 DRAM 上的 RDD,其数组部分直接申请到老年代的相应区域,避免了通过 GC 算法进行移动的消耗,而其他部分和没标注的 RDD 一起申请到年轻代,最终通过 Minor GC 再晋升.Major GC 负责整理消除碎片,并根据 RDD 具体调用的次数,对标签进行修改调整.在静态分析的帮助之下,Panthera 可以在混合内存上的性能表现与同等大小的纯 DRAM 内存性能持平.类似的,Teracache 用 NVM 来缓解 DRAM 的缓存压力,通过 I/O 映射的模式,实现混合堆内存的存储和管理.而 Flat[84]同样采用 DRAM 和 NVM 的混合内存结构来缓存 RDD.

NVM 的持久化特性被运用在大数据处理框架的故障恢复当中.Espresso[88]提供了一个协助 JVM 将数据持久化到 NVM 上的方法,而 GCPersist[85]拓展了 Espresso 用于 Spark 的检查点(checkpoint)持久化,实现快速故障恢复.Spark 的 RDD 缓存通常存储在 DRAM 内存中,或是持久化在到磁盘中,如果执行器 JVM 宕机造成内存当中的所有缓存断电损失,可以从磁盘中重新反序列化得到,但磁盘读写的代价高昂,而持久化到 NVM 则可以有效提升速度.Espresso 和其他类似持久化备份方法采用积极的同步策略,缓存一有脏写就会更新到 NVM,然而 NVM 的读写性能和 DRAM 内存的性能还存在一定差距,积极的同步持久化策略将对应用执行效率存在影响.GCPersist 借用 GC 的时机,将持久化的工作和 Parallel GC 的工作同步进行,而在 GC 的全局暂停过程中,RDD 对象也恰好能够保持静态一致.GCPersist 具体的持久化过程和 GC 相似,从根对象开始标记扫描需要持久化的

部分,之后总结和拷贝相应的对象,不同的是拷贝的目标位置位于 NVM,作为年轻代和老年代之外的一个专属堆区域.另外,GCPersist 定期对 NVM 堆区域进行扫描,确定已经被有效备份的 RDD.用于读写的数据本身依旧保留在 DRAM 内存中,NVM 的持久化部分只作为一个快照备份用于故障恢复,因而 GCPersist 下的任务执行速度和在原生环境下基本没有差异,但在宕机发生后的故障恢复速度得到了有效提高.

8.3 小结

本章介绍了针对新型硬件架构的适应性优化技术,包括将内存分解架构应用于大数据处理框架,通过将 GC 任务本地化,减少了对象扫描过程中的远程内存访问;包括将 NVM 和 DRAM 的混合内存结构引入大数据处理框架的执行器 JVM,利用 NVM 高效廉价的可持久化存储,提高大数据处理框架的缓存能力和故障恢复速度.这些优化工作对硬件环境的针对性很强,对未来大数据处理框架与新兴硬件的结合具有启示作用.

9 总结及未来展望

本文总结了传统 JVM 在大数据处理框架下存在的性能问题,对问题产生的原因进行了细致分析,从 5 个方向综述了现有在大数据分布式场景下,具有代表性的 JVM 优化技术.随着云计算技术的发展和普及,作为主流运行时环境的 JVM 在软件栈中扮演着越来越重要的角色[89,90],针对 JVM 在各种环境下的性能优化研究不断涌现.而大数据处理框架也在被使用到更多更复杂的计算场景当中[91,92],使得大数据处理框架下的 JVM 优化成为了一个研究热点.本文综述的优化工作已经在自动内存管理,数据对象转换,虚拟机冷启动等一系列相关问题中取得一定进展,表 8 对这些优化方法的作用特点和优缺点进行了总结和对比,可以看出大部分优化技术依然存在一定局限性,包括:(1)适用的大数据框架类型有限,可迁移能力不足;(2)要求开发者对框架处理流程的深入理解,需要开发者大量的辅助工作;(3)可用性和可靠性缺少保障,容易出错;(4)辅助算法的时间复杂度或空间复杂度较高等.结合现有工作的局限性和本文对 JVM 性能问题的原因分析,本文认为未来研究工作面临的挑战和可能解决方案如下:

表 8 大数据处理框架下的 JVM 优化方法对比

优化方法	作用特点	优点	缺点
大数据框架的内存管理优化	利用了大数据框架特定的内存分配结构和计算流程	有效适配相应大数据框架,提升内存管理效率	只适用于特定的大数据框架,通用性有限
执行器集群的协同优化	对分布式环境下的所有执行器 JVM 进行统筹规划	协调全局所有 JVM 执行器,提升大数据框架整体速率	可能需要占用额外的计算资源
JVM 中数据对象的存储方法优化	以显式内存管理的方式减少对象外壳对数据存储的影响	降低垃圾回收机制负担,提升内存使用效率	可能增加额外开发者负担,存在执行异常的风险
JVM 的 GC 算法优化	提升 GC 算法的处理速度和对大数据环境的适应性	能够普遍适用于各种大数据框架和计算场景	无法降低内存使用量,可能需要额外辅助工作
新型硬件架构下的 JVM 优化	提升大数据框架下的执行器 JVM 对新兴硬件技术的适应性	提升大数据处理框架的缓存能力和故障恢复速度	无法降低存储使用量,适用于特定硬件环境

9.1 访问速率友好的内存管理

内存使用压力增加不仅意味着更大的内存管理压力,同样意味着更大的数据处理量.现有优化工作一定程度上降低了海量数据对象带来的 GC 负担,但还没有充分考虑怎样管理内存能够提高数据的访问和处理速率.

(1) 使用堆外内存以规避 GC 无意义的扫描和迁移是使用普遍的优化技巧,如 Flink,Spark Tungsten 都选择在堆外内存存放内存页.尽管堆外内存在降低 GC 时间和提升数据 I/O 速度方面,相对堆内存具有一定优势,但 JVM 读写操作堆外内存的速度要低于堆内存[93],另外堆外内存的使用安全性并不能得到充分保障.对于操作频繁的大数据应用,未来研究工作需要权衡堆外内存带来的 GC 效率提升和对应用执行速率的影响,可以尝试的是让数据对象和内存页对象回到堆内存储,并使这些生命周期清晰的对象在堆内也能够规避 GC.

(2) 数据访问的空间局部性和时间局部性是当前 GC 算法和大数据处理框架都关注的问题.不论是应用线

程和还是 GC 线程,都需要按照一定的顺序遍历数据.如果数据在堆内存中凌乱分布,将不利于 CPU 缓存命中和 TLB 命中.当前的内存管理方法在申请和移动数据对象时还没有充分考虑这一点,对数据的排列顺序可能存在破坏.未来研究工作需要探究的是创造和维护数据内存分布局部性的方法,例如按照对访问速率有利的顺序在内存中布局数据,并在内存整理时避免破坏布局.

(3) 对于大量数据传输带来的序列化/反序列化问题,现有优化工作中,Skyway 直接传输对象将增加网络传输量,而 Gerenuk 直接传输数据值无法保证可用性.鉴于现有优化技术采用了内存页对象存储数据,未来研究工作可以提供内存页对象的传输途径,降低数据传输量和序列化负担,并保证数据在节点间传输后的安全可用.

9.2 内存使用模式感知的GC自适应调整

大数据应用内存使用模式的变化不仅体现在数据对象生命周期的增长,也体现在不同处理阶段对象使用量和对象生命周期的动态变化.现有优化工作一定程度上降低了长时间存活对象对 GC 算法的影响,但对如何使 GC 算法在运行时能够针对不同的内存使用模式进行自适应调整还需要研究.

(1) 大数据应用的内存使用模式在不同处理阶段是动态变化的,但现有 GC 算法在运行时的自适应调整,依据的是 GC 指标的历史统计结果.在这种基于历史的自适应调整策略下得到的 GC 参数,并不一定能够使适应未来的内存使用模式.另外 GC 参数在当前策略下往往需要经历数次 GC,才能适应内存使用模式的变化,具有一定滞后性.因而,大数据应用在运行时对 GC 参数的调整和干预是有必要的,但目前 JVM 并没有提供相应的接口.未来研究工作可以尝试提供更多运行时的调整接口,允许开发者在应用执行的不同阶段对 GC 参数做出改变.

(2) 由于 JVM 和大数据处理框架在运行时的信息交互有限,目前有关 GC 算法运行时自适应调整的优化只局限于堆大小调整和 GC 触发时机等方面,而有关大数据框架运行时自适应调整的优化也只局限于大数据框架的资源分配参数.除此之外,更多参数调整的工作集中于静态参数调整,这些工作在内存使用模式动态变化的大数据框架下并不完全适用.未来研究工作中,大数据框架可以尝试对不同处理阶段的内存使用模式进行预测,作为启发信息与 JVM 交互,JVM 结合启发信息和当前的堆内存使用状态,则能够及时有效地在运行时调整更多的 GC 参数.

9.3 轻量级的JVM运行时优化

现有的优化工作专注于降低 GC 暂停,JVM 冷启动时间等消耗,然而这些优化算法可能引入了更多的 CPU 竞争、内存占用和开发者负担.如何设计轻量级的优化技术,进一步降低副作用是值得探讨的.

(1) 在最新的 Java 版本中,高度并发 GC 算法 Shenandoah 和 ZGC 已经成熟可用,这些并发算法以降低 GC 暂停时间为目标,在暂停时间上相比 Parallel GC 有着较大优势.但为了维护 GC 线程和应用线程的一致堆视图,这些 GC 算法需要引入大量读写屏障,另外并发 GC 线程和应用线程之间存在着 CPU 竞争.因而在计算量较大的大数据应用下使用这些 GC 算法时,应用线程的执行时间会相对增加.鉴于目前大部分的优化工作是建立在 Parallel GC 算法上,未来研究工作可以将已有优化技术,如基于区域的数据对象管理,移植到这些并发算法上,通过降低处理负担,减少这些并发 GC 算法与应用线程的冲突.另外,开发专用的 GC 硬件加速器也是一个可以继续深入的解决方向[94-96].

(2) 现有优化工作 HotTub 通过重用热的 JVM 来缓解 JVM 冷启动时间过长的问题.但维持一个已预热的 JVM 池可能引入更多的内存开销,这使得这种优化方法难以应用于内存资源紧张的大数据环境.未来研究工作可以尝试的是 JVM 级别的检查点/回退(Checkpoint/ Rollback)技术[97-99],将 JVM 初始化后的映像持久化在磁盘上,在新的任务提交时,将 JVM 回退到预热好的映像以避免冷启动.另外代码缓存的跨节点共享也可以一定程度解决冗余的 JIT 编译.

(3) 为了协助 JVM 在大数据应用中识别数据对象,并判断数据对象的具体生命周期,现有优化工作如 Yak,DSA 等需要开发者在大数据处理框架和大数据应用的代码中做出人工注释,引入了显著的开发者负担.而在优化工作如 ROLP,POLM2 为了降低人工负担,则需要在 GC 算法的运行时加入对象存活时间的统计和预测器,引入显著的运行时开销.未来的研究工作可以尝试使用静态代码分析的技术,在应用执行之前完成数据对象

的识别,以及数据对象生命周期的预测工作,在不增加人力工作的情况下,转移 JVM 在运行时的负担.

References:

[1] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113.

[2] Isard M, Budiu M, Yu Y, et al. Dryad: distributed data-parallel programs from sequential building blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. 2007: 59-72.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets. HotCloud, 2010, 10(10-10): 95.

[5] Carbone P, Katsifodimos A, Ewen S, et al. Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4).

[6] Bu Y, Borkar V, Xu G, et al. A bloat-aware design for big data applications. Proceedings of the 2013 international symposium on memory management. 2013: 119-130.

[7] Nguyen K, Fang L, Navasca C, et al. Skyway: Connecting managed heaps in distributed big data systems. ACM SIGPLAN Notices, 2018, 53(2): 56-69.

[8] Lion D, Chiu A, Sun H, et al. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2016: 383-400.

[9] Xu L, Guo T, Dou W, et al. An experimental evaluation of garbage collectors on big data applications. Proceedings of the VLDB Endowment, 2019, 12(5):570-583.

[10] Luo L, Liu Y, Qian DP. Survey on in-memory computing technology. Ruan Jian Xue Bao/Journal of Software, 2016,27(8):2147-2167 (in Chinese with English abstract).

[11] Ji ZY, Pan W. Present research status and prospects of in-memory data management big data era. Computer Engineering and Design, 2014, 35(10): 3499-3506 (in Chinese with English abstract).

[12] Bruno R, Ferreira P. A study on garbage collection algorithms for big data environments. ACM Computing Surveys (CSUR), 2018, 51(1): 1-35.

[13] Zhang X, Lu L, Shi X H. In-Memory Data-Object Management in Distributed Data Processing System. Zte Technology, 2016, 22(2): 19-22 (in Chinese with English abstract).

[14] Cheng XQ, Jin XL, Wang YZ, et al. Survey on big data system and analytic technology. Journal of software, 2014, 25(9):1889-1908 (in Chinese with English abstract).

[15] Jones R, Hosking A, Moss E. The garbage collection handbook: the art of automatic memory management[M]. CRC Press, 2016.

[16] JVM Generations. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html.

[17] Detlefs D, Flood C, Heller S, et al. Garbage-first garbage collection. Proceedings of the 4th international symposium on Memory management. 2004: 37-48.

[18] Spark executor GC taking long. http://stackoverflow.com/questions/38965787/sparkexecutor-gc-taking-long

[19] Maas M, Asanović K, Harris T, et al. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. Acm SIGPLAN Notices, 2016, 51(4): 457-471.

[20] Wu M, Zhao Z, Yang Y, et al. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. 2020 USENIX Annual Technical Conference (USENIX ATC). 2020: 159-172.

[21] Maas M, Harris T, Asanović K, et al. Trash day: Coordinating garbage collection in distributed systems. 15th Workshop on Hot Topics in Operating Systems (HotOS). 2015.

[22] Navasca C, Cai C, Nguyen K, et al. Gerenuk: thin computation over big native data using speculative program transformation. Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 538-553.

[23] OOM error caused by large array allocation in G1. http://mail.openjdk.java.net/pipermail/hotspot-gc-use/2017-November/002725.html.

[24] Nguyen K, Wang K, Bu Y, et al. Facade: A compiler and runtime for (almost) object-bounded big data applications. ACM SIGARCH Computer Architecture News, 2015, 43(1): 675-690.

[25] Suo K, Rao J, Jiang H, et al. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. Proceedings

of the Thirteenth EuroSys Conference. 2018: 1-15.

[26] Yu Y, Lei T, Zhang W, et al. Performance analysis and optimization of full garbage collection in memory-hungry environments. ACM SIGPLAN Notices, 2016, 51(7): 123-130.

[27] Maas M, Asanović K, Kubiatowicz J. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. Proceedings of the 16th Workshop on Hot Topics in Operating Systems. 2017: 138-143.

[28] Ding MS, Chen SM. Helius: A Lightweight Big Data Processing System. Journal of Computer Application, 2017, 37(2): 305-310 (in Chinese with English abstract).

[29] Kedia P, Costa M, Parkinson M, et al. Simple, fast, and safe manual memory management. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017: 233-247.

[30] Herodotou H, Chen Y, Lu J. A survey on automatic parameter tuning for big data processing systems. ACM Computing Surveys (CSUR), 2020, 53(2): 1-37.

[31] Xu L, Li M, Zhang L, et al. Memtune: Dynamic memory management for in-memory data analytic frameworks. 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2016: 383-392.

[32] Wang G, Xu J, He B. A novel method for tuning configuration parameters of spark based on machine learning. 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2016: 586-593.

[33] Li M, Liu Z, Shi X, et al. ATCS: Auto-Tuning Configurations of Big Data Frameworks Based on Generative Adversarial Nets. IEEE Access, 2020, 8: 50485-50496.

[34] Zhang X, Zeng L, Shi X, et al. HCOpt: An Automatic Optimizer for Configuration Parameters of Hadoop. International Conference on Human Centered Computing. Springer, Cham, 2016: 599-610.

[35] Yu Z, Bei Z, Qian X. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 2018: 564-577.

[36] Zhu Y, Liu J, Guo M, et al. Acts in need: automatic configuration tuning with scalability guarantees. Proceedings of the 8th Asia-Pacific Workshop on Systems. 2017: 1-8.

[37] Jayasena S, Fernando M, Rusira T, et al. Auto-tuning the java virtual machine. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop. IEEE, 2015: 1261-1270.

[38] Singer J, Kovoor G, Brown G, et al. Garbage collection auto-tuning for Java mapreduce on multi-cores. Proceedings of the international symposium on Memory management. 2011: 109-118.

[39] Herodotou H, Dong F, Babu S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011: 1-14.

[40] Herodotou H, Lim H, Luo G, et al. Starfish: A Self-tuning System for Big Data Analytics. 5th Biennial Conference on Innovative Data Systems Research. 2011, 11(2011): 261-272.

[41] Tan Z, Babu S. Tempo: Robust and Self-Tuning Resource Management in Multi-tenant Parallel Databases. Proceedings of the VLDB Endowment, 2016, 9(10).

[42] Bao L, Liu X, Chen W. Learning-based automatic parameter tuning for big data analytics frameworks. 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018: 181-190.

[43] Zhu Y, Liu J, Guo M, et al. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. Proceedings of the 2017 Symposium on Cloud Computing. 2017: 338-350.

[44] Li M, Zeng L, Meng S, et al. Mronline: Mapreduce online performance tuning. Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. 2014: 165-176.

[45] Kunjir M, Babu S. Black or White? How to Develop an AutoTuner for Memory-based Analytics. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 1667-1683.

[46] Lu L, Shi X, Zhou Y, et al. Lifetime-based memory management for distributed data processing systems. Proceedings of the VLDB Endowment, 2016, 9(12): 936-947.

[47] Shi X, Ke Z, Zhou Y, et al. Deca: a garbage collection optimizer for in-memory data processing. ACM Transactions on Computer Systems (TOCS), 2019, 36(1): 1-47.

[48] Wang J, Balazinska M. Elastic memory management for cloud data analytics. 2017 USENIX Annual Technical Conference (USENIX ATC 17). 2017: 745-758.

[49] Nguyen K, Fang L, Xu G, et al. Yak: A high-performance big-data-friendly garbage collector. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2016: 349-365.

[50] Bruno R, Patricio D, Simão J, et al. Runtime object lifetime profiler for latency sensitive big data applications. Proceedings of the Fourteenth EuroSys Conference 2019. 2019: 1-16.

[51] Wang C, Ma H, Liu S, et al. Semeru: A Memory-Disaggregated Managed Runtime. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2020: 261-280.

[52] Wang C, Cui H, Cao T, et al. Panthera: holistic memory management for big data processing over hybrid memories. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019: 347-362.

[53] Jia D, Bhimani J, Nguyen S N, et al. Atumm: Auto-tuning memory manager in apache spark. 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC). IEEE, 2019: 1-8.

[54] Kolokasis I G, Papagiannis A, Pratikakis P, et al. Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O. 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage). 2020.

[55] Shi X, Chen M, He L, et al. Mammoth: Gearing Hadoop Towards Memory-Intensive MapReduce Applications. IEEE Transactions on Parallel and Distributed Systems, 2014, 1(8):2300-2315.

[56] Tang Z, Zeng A, Zhang X, et al. Dynamic memory-aware scheduling in spark computing environment. Journal of Parallel and Distributed Computing, 2020, 141: 10-22.

[57] Gog I, Giceva J, Schwarzkopf M, et al. Broom: Sweeping out garbage collection from big data systems. 15th Workshop on Hot Topics in Operating Systems (HotOS). 2015.

[58] Geng Y, Shi X, Pei C, et al. Lcs: an efficient data eviction strategy for spark. International Journal of Parallel Programming, 2017, 45(6): 1285-1297.

[59] Yang Z, Jia D, Ioannidis S, et al. Intermediate data caching optimization for multi-stage and parallel big data frameworks. 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018: 277-284.

[60] Murray D G, McSherry F, Isaacs R, et al. Naiad: a timely dataflow system. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 439-455.

[61] Fireman D, Brunet J, Lopes R, et al. Improving tail latency of stateful cloud services via gc control and load shedding. 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2018: 121-128.

[62] Shi X, Zhang X, He L, et al. MURS: Mitigating Memory Pressure in Service-Oriented Data Processing System. 2017 IEEE International Conference on Web Services (ICWS). IEEE, 2017: 428-435.

[63] Saha B, Shah H, Seth S, et al. Apache tez: A unifying framework for modeling and building data processing applications. Proceedings of the 2015 ACM SIGMOD international conference on Management of Data. 2015: 1357-1369.

[64] Nailgun: Insanely fast Java. http://www.martiansoftware.com/nailgun/background.html.

[65] Morisawa Y, Suzuki M, Kitahara T. Flexible Executor Allocation without Latency Increase for Stream Processing in Apache Spark. 2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020: 2198-2206.

[66] Cameron C, Singer J, Vengerov D. The judgment of Forseti: Economic utility for dynamic heap sizing of multiple runtimes. Proceedings of the 2015 International Symposium on Memory Management. 2015: 143-156.

[67] Simão J, Esteves S, Veiga L. Smartgc: Online memory management prediction for paas cloud models. OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Springer, Cham, 2017: 370-388.

[68] Hu Z Y, Shi X H, Ke Z X, et al. Estimating the memory consumption of big data applications based on program analysis (in Chinese). Sci Sin Inform, 2020, 50: 1178 – 1196 (in Chinese).

[69] Cohen N, Petrank E. Data structure aware garbage collector. Proceedings of the 2015 International Symposium on Memory Management. 2015: 28-40.

[70] Yang A M, Österlund E, Wrigstad T. Improving program locality in the GC using hotness. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020: 301-313.

[71] Yang A M, Österlund E, Wilhelmsson J, et al. ThinGC: complete isolation with marginal overhead. Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management. 2020: 74-86.

[72] Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[73] Flood C H, Kennke R, Dinn A, et al. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java framework: Virtual Machines, Languages, and Tools. 2016: 1-9.

[74] OpenJDK. ZGC - The Z Garbage Collector. https://openjdk.java.net/projects/zgc/, 2019.

[75] Suo K, Rao J, Jiang H, et al. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. Proceedings of the Thirteenth EuroSys Conference. 2018: 1-15.

[76] Li H, Wu M, Zang B, et al. ScissorGC: scalable and efficient compaction for Java full garbage collection. Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2019: 108-121.

[77] Yu Y, Lei T, Zhang W, et al. Performance analysis and optimization of full garbage collection in memory-hungry environments. ACM SIGPLAN Notices, 2016, 51(7): 123-130.

[78] Bruno R, Oliveira L P, Ferreira P. NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications. Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management. 2017: 2-13.

[79] Bruno R, Ferreira P. POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference. 2017: 147-160.

[80] Blackburn S M, Garner R, Hoffmann C, et al. The DaCapo benchmarks: Java benchmarking development and analysis. Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. 2006: 169-190.

[81] Huang S, Huang J, Dai J, et al. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010). IEEE, 2010: 41-51.

[82] Gidra L, Thomas G, Sopena J, et al. NumaGiC: A garbage collector for big data on big NUMA machines. ACM SIGARCH Computer Architecture News, 2015, 43(1): 661-673.

[83] 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.

[84] Khan M M, Alam M A U, Nath A K, et al. Exploration of memory hybridization for RDD caching in Spark. Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management. 2019: 41-52.

[85] Wu M, Chen H, Zhu H, et al. GCPersist: an efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2020: 1-14.

[86] Volos H, Magalhaes G, Cherkasova L, et al. Quartz: A lightweight performance emulator for persistent memory software. Proceedings of the 16th Annual Middleware Conference. 2015: 37-49.

[87] Akram S, Sartor J B, McKinley K S, et al. Write-rationing garbage collection for hybrid memories. ACM SIGPLAN Notices, 2018, 53(4): 62-77.

[88] Wu M, Zhao Z, Li H, et al. Espresso: Brewing java for more non-volatility with non-volatile memory. Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 2018: 70-83.

[89] Bruno R, Ferreira P, Synytsky R, et al. Dynamic vertical memory scalability for OpenJDK cloud applications. Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management. 2018: 59-70.

[90] Goumas G, Nikas K, Lakew E B, et al. ACTiCLOUD: Enabling the Next Generation of Cloud Applications. 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2017: 1836-1845.

[91] Costa C H A, Misale C, Liu F, et al. Optimization of genomics analysis pipeline for scalable performance in a cloud environment. 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). IEEE, 2018: 1147-1154.

[92] Anderson M, Smith S, Sundaram N, et al. Bridging the gap between HPC and big data frameworks. Proceedings of the VLDB Endowment, 2017, 10(8): 901-912.

[93] Stephan Ewen. Off-heap Memory in Apache Flink and the curious JIT compiler. https://flink.apache.org/news/2015/09/16/off-heap-memory

[94] Kotselidis C, Diamantopoulos S, Akrivopoulos O, et al. Efficient compilation and execution of JVM-based data processing frameworks on heterogeneous co-processors. 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE,

2020: 175-179.

[95] Jang J, Heo J, Lee Y, et al. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. 2019: 726-739.

[96] Maas M, Asanović K, Kubiatowicz J. A hardware accelerator for tracing garbage collection. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2018: 138-151.

[97] Bell J, Pina L. Crochet: Checkpoint and rollback via lightweight heap traversal on stock jvms. 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[98] Du Y, Shi X, Jin H, et al. FITDOC: fast virtual machines checkpointing with delta memory compression. 2014 IEEE 17th International Conference on Computational Science and Engineering. IEEE, 2014: 291-298.

[99] Wimmer C, Stancu C, Hofer P, et al. Initialize once, start fast: application initialization at build time. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-29.

附中文参考文献:

[10] 罗乐, 刘轶, 钱德沛.内存计算技术研究综述.软件学报,2016,27(08):2147-2167.

[11] 嵇智源, 潘巍.面向大数据的内存数据管理研究现状与展望.计算机工程与设计,2014,35(10):3499-3506.

[13] 张雄, 陆路, 石宣化. 分布式数据处理系统内存对象管理问题分析. 中兴通讯技术, 2016, 22(2): 19-22.

[14] 程学旗, 靳小龙, 王元卓, 等. 大数据系统和分析技术综述. 软件学报, 2014, 25(9):1889-1908.

[28] 丁梦苏, 陈世敏. 轻量级大数据运算系统 Helius. 计算机应用, 2017, 37(2): 305-310.

[68] 胡振宇, 石宣化, 柯志祥, 等. 基于程序分析的大数据应用内存预估方法. 中国科学: 信息科学, 2020, 50(8): 117